Lecture Scribing

Dr. Miodrag Bolic

Organized By:

Ismaeel Al Ridhawi - 3385278
Zeeshan Ansari - 2994576

TABLE OF CONTENTS

Lecture Section

Lecture #1a: NIOS II Architecture

Course: Computer Architecture III
Lecturer: Miodrag Bolic
Scribe: Li Zhang (3153125), Xing Yuan Wang (2804255)
Date: October 15, 2007

Introduction

The lecture covers the following topics: Stratix Altera Devices, NIOS II processor architecture, and how to design a system using NIOS II processors.

Explanation

Basic Description of Stratix Altera Devices

Stratix Chips
Stratix [1] is a family of Field-programmable Gate Arrays (FPGAs) designed and manufactured by Altera; FPGAs feature high-performance architecture, Digital Signal Processing blocks, clock management circuitry, external memory device interfaces, high-speed I/O interfaces, and NIOS II embedded processors.
Logic Array Block Placement

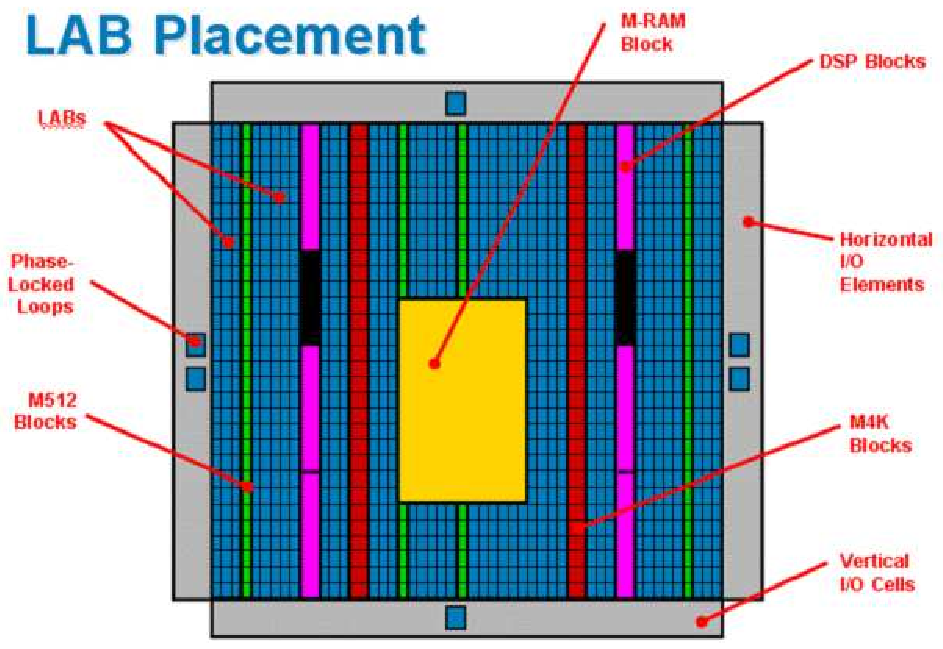


Figure 1 LAB Placement [6]

The placement graph is self-explainable except the TriMatrix[2] Memory structure. The structure composed of three sizes of embedded RAM blocks namely 512-bit M512 blocks, 4-Kbit M4K blocks, and 512-Kbit M-RAM blocks. The TriMatrix memory structure makes the Stratix devices ideal for memory-intensive applications. In the 3 types of RAM blocks, the bigger the memory size is, the lower the throughput is. The smaller M512 RAM blocks are good for First-In First-Out (FIFO) functions where memory bandwidth is critical. M-RAM blocks are good for large buffering applications such as image processing. The M4K blocks are ideal for medium-sized memory applications such as asynchronous transfer mode (ATM) cell processing. Among the Stratix Device family, EP1S10 that will be used in SITE lab has 2 M-RAM blocks, 138 M4K blocks, and 224 M512 blocks.

1.3 LAB and Logic Element [3]

Each LAB (Logic Array Block) contains 10 Logic Elements (LEs) and local interconnects that includes LAB input lines, LE feedback lines, and lines connecting a whole column of LABs. One of the design goals of a FPGA is making the interconnecting lines as short as possible. LE is a very important measurement of a FPGA's capacity.

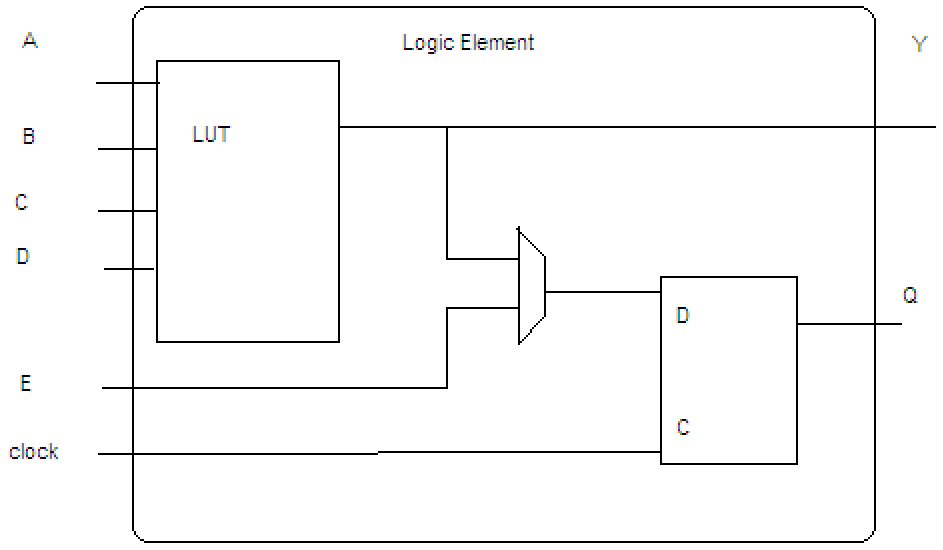


Figure 1 A Conceptual Logic Element
Figure2 shows that each LE composes of both combinational circuit and storage circuit (the D flip-flop in the figure). Among the combinational circuits, the Look-Up-Table (LUT) unit provides 16 possible outputs for the 4 inputs. The multiplexer is for choosing if the storage unit accepts input from outside of the LE or from inside result of the LUT. The real LE is much more complicated than the concepts. For example, the storage unit can be either D or JK flip-flop with or without control signals such as set, reset, and enable.

1.4 FPGA Configuration

Configuring a FPGA device is actually configuring the LEs, interconnections of the LABs, memory and DSP blocks. A FPGA device usually has thousands of LABs; although manually scripting to configure LEs and interconnections is possible, commercial software package such as Quartus II takes care of the detail that will usually produce optimized result.

2 NIOS II Processor Architecture

2.1 FPGA Use in Embedded Applications

A Field Programmable Gate Array (FPGA) is a semiconductor device containing programmable logic components and programmable interconnects. FPGAs are generally slower than their application-specific integrated circuit (ASIC) counterparts and draw more power. However, they have several advantages such as a shorter time to market, ability to re-program in the field to fix bugs, and lower non-recurring engineering costs[5].

For embedded applications, FPGA can be used as custom microcontroller, processor companion chip, or multi-processor system.

- Custom microcontroller that contains CPU, UART, Ethernet interfaces, RAM, SPI, and other custom function circuits moves functions from board to FPGA so that only necessary features are on the chip and thus reduces system cost, complexity and power consumption.
- Processor companion chip does not contain a CPU thus the FPGA has to rely on external CPU or DSP computational power while provides some extended system features and performance. This type of configuration usually adds some peripherals (such as PCI, Ethernet, SPI, UART) and custom logic functions in the FPGA.
- Multi-Processor system that contains multiple CPUs and other peripheral/custom functions in the FPGA boosts system performance and off-loads existing external processor.

2.2 Soft Processor Core and Firm Processor

NIOS II and Stratix device support processors as soft processor while other vendors may support certain kind of firm processor that is preprogrammed on FPGA. Some FPGA may implement fixed blocks as CPUs (hard processors); for example, some Xilinx FPGA chips may contain PowerPC hard processors. Hard processors have its advantage and disadvantages:

Advantage: the CPUs are implemented as fixed blocks that are optimized for high speed, best performance, and generally low power consumption. While soft processor may work at 200 MHz clock, hard processors on similar FPGA may work at as fast as 500 MHz.

Disadvantages:

- The placement is fixed.

- Cannot match the amount of CPUs required by the application.
- Not application specific. Some application may require special instruction set that a hard processor does not support.

2.3 NIOS II soft core Processor

NIOS II provide soft IP core processors to FPGAs. A soft-core processor is a microprocessor fully described in software, usually in a HDL, which can be synthesized in programmable hardware, such as FPGAs. NIOS II IP soft core features the following:

- Reduced Instruction Set Computer (RISC)
- Pipeline configurations may be zero, 5, or 6 stages
- Full 32 bit instruction set, data path, and address space
- 32 general-purpose registers
- Access to a variety of on-chip peripherals, and interfaces to off-chip memories and peripherals
- Software development environment based on the GNU C/C++ tool chain and Eclipse IDE

A simple Altera Cyclone device with 2910 logic elements may contain 1 NIOS CPU that is roughly 20% of the device while a complex Altera Stratix II EP2S180 device with 179,400 Logic Elements may contain up to 80 NIOS CPUs so that very powerful system can be built.

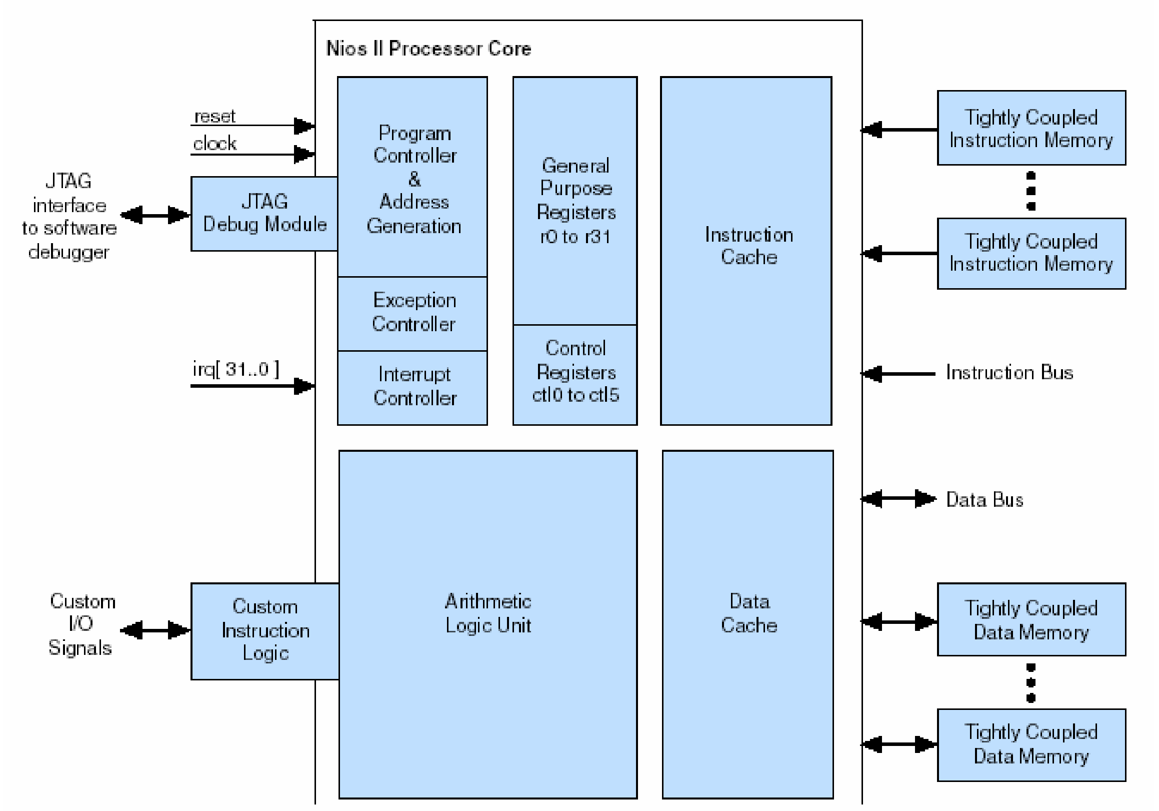

Nios II Processor Core
reset
clock
JTAG interface to software debugger
JTAG Debug Module
Program Controller & Address Generation
General Purpose Registers r0 to r31
Instruction Cache
Tightly Coupled Instruction Memory
Tightly Coupled Instruction Memory
Exception Controller
Interrupt Controller
Control Registers ctl0 to ctl5
irq[31..0]
Instruction Bus
Data Bus
Custom I/O Signals
Custom Instruction Logic
Arithmetic Logic Unit
Data Cache
Tightly Coupled Data Memory
Tightly Coupled Data Memory

Figure 2 NIOS II Processor Core Architecture

The NIOS II architecture describes an instruction set, not a particular hardware implementation. For example, a multiplication instruction can be implemented as a piece of hardware or a software routine that utilizes some simpler instructions. From Figure 3, one can see NIOS II soft core's features:

- JTAG debug module can be included or excluded for debugging codes.
- Arithmetic Logic Unit (ALU) supports arithmetic operations such as addition, subtraction, shift, and/or multiplications.
- Register files
- Custom Instruction Logic provides extra special instructions in addition to standard ALU instructions
- Tightly coupled memory provides guaranteed low-latency access. NIOS has up to 4 such memories. Tightly coupled memory is best used as fast data buffers, fast sections of code, fast interrupt handler and critical loop.
- Cache is useful for high latency external memories. NIOS 32 uses directly mapped cache and write-through policy

There are 3 versions of NIOS II soft-core processor provided: version e, s and f. The 3 versions of processor are in turn more complicated and powerful than the other; descriptions about these three versions are detailed in NIOS handbook. [4]

2.4 Pipelined Processor Review

Pipeline used in processors allows the parallel execution of two or more consecutive instructions from a nominally sequential stream; the processing elements are the logical circuits that implement the various stages of an instruction (address decoding and arithmetic, register fetching, cache lookup, etc.).

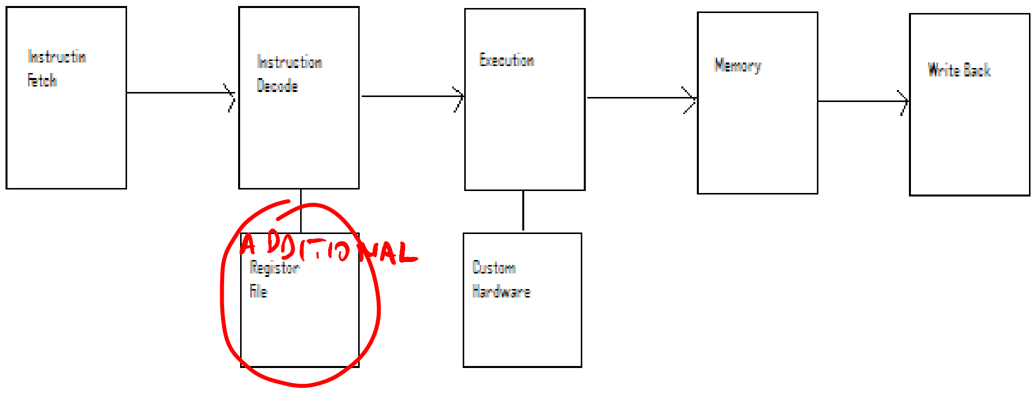


Figure 3 Pipelined CPU Structure

Version s NIOS II microprocessor implements a static branch prediction for pipeline algorithm; static branch prediction predicts the same direction for the same branch during the whole execution of program and consists of three possible means: predict always not-taken, predict always taken and backward branch predict taken, forward branch not taken.

3.1 SOPC Builder Design

Altera has developed many features towards System-On-a-Programmable Chip (SOPC). In order to make use of Stratix FPGAs, one can generally build custom hardware by using VHDL/Verilog on Quartus II IDE and download the resulting SOPC file???? Into the FPGA device; first designer can define the system in SOPC builder; SOPC contains detailed wizard that guides designer to generate a complete NIOS microprocessor that suits the requirements. Then other custom circuitry can be added along the NIOS soft-core to form the system; the system generated can then downloaded into the target FPGA. The software can be designed in parallel with the hardware design, because NIOS IDE provides Hardware Abstraction Layer (HAL) to isolate the hardware and standard APIs. Finally in NIOS II IDE one can download and debug C/C++ code on the customized hardware.

3.2 Hardware Abstraction Layer (HAL) [4]

Hardware Abstraction Layer (HAL) allows ANSI C access all hardware without knowing how the hardware is actually implemented. HAL somehow isolates the application software from hardware modifications; as a result, applications are device-independent because they abstract information of many systems from HAL APIs in ANSI C libraries. Each device driver interfaces with a specific peripheral and provides services for HAL APIs to communicate with hardware in NOIS core. Such systems include:

- Character mode devices: UART core, JTAG UART core, LCD display controller
- Flash memory devices
- Timer devices
- DMA controller core
- Ethernet MAC/PHY Controller

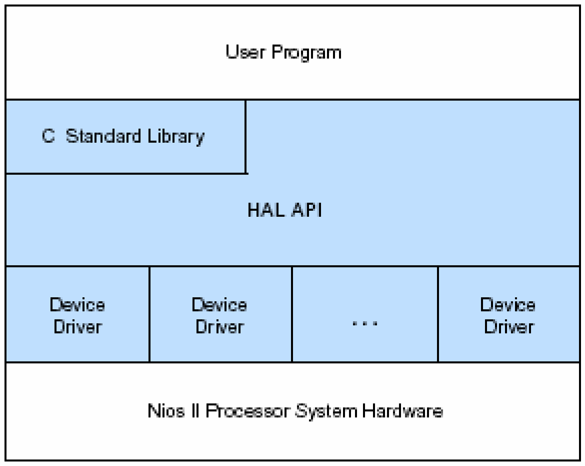


Figure 4 HAL API Structure [4]

NIOS II IDE generates a custom HAL system library to match a specific SOPC hardware system. The drivers can then be accessed within C programming. The external hardware devices are then directly controlled.

Summary

The lecture first explored the Altera Stratix family structures. A floor plan for LAB placement was presented and then TriMatrix memory structures were introduced. Generally speaking, the 3 memory structures are good for specific applications due to capacity and speed trade-offs. Within a LAB, LEs were studied in more detail. LE as basic unit of FPGA is one of the most important measurements for a FPGA.

The second part of the lecture explored NIOS II soft core CPUs. A comparison between soft and hard processors was given. Hard processor was better in performance with less flexibility. A pipelined CPU structure was then reviewed for the NIOS II processor architecture.

The close the lecture, a detailed instruction of how to use NIOS II and Quartus Integrated Development Environment (IDE) to construct customized hardware as well as program debug was illustrated.

Upon completion of the lecture, students gained a clear concept of what a FPGA is about and how to work on a configurable hardware platform.

Paper 1 System-on-chip: reuse and integration

Paper 2 Design Considerations for Soft Embedded Programmable Logic Cores (PLC)

Lecture #1b Introduction to Nios II Processor Architecture and Programming

Lecture #1: Wednesday, September 13, 2006
Lecturer: Miodrag Bolic
Scriber: Grant Yu (3413309) and Feng Xu (3144920)

Introduction

In the first lecture, the laboratory environment is introduced to the students: the physical environment of Altera Stratix devices including its floor mapping issues, memory hierarchy, logic array block (LAB) and logic element (LE); Altera NIOS II soft core processor including its placement issues, different versions, pipeline and expandability, implementation and design issue; and extended reading for applications and advantages of FPGA technology.

Lecture Scribing Section:

1. Introduction to Altera Stratix FPGA device

Explanation

1.1 Organization of the NIOS development board

The Altera NIOS development board, Stratix edition, provides a hardware platform for FPGA design project based on vendor designed NIOS II soft core processor. The board features [10]:

- A Stratix EP1S10F780C6 device
- 8 Mbytes of flash memory
- 1 Byte of static RAM
- 16 Mbytes of SDRAM
- On board logic for configuring the Stratix device from flash memory
- On-board Ethernet MAC/PHY device
- Two 5-V-tolerant expansion/prototype headers each with access to 41 Stratix user I/O pins
- Compact Flash connector header for Type I Compact Flash (CF) cards
- Mictor connector for hardware and software debug
- Two RS-232 DB9 serial ports
- Four push-button switches connected to Stratix user I/O pins

- Eight LEDs connected to Stratix user I/O pins
- Dual 7-segment LED display
- JTAG connectors to Altera® devices via Altera download cables
- 50 MHz oscillator and zero-skew clock distribution circuitry
- Power-on reset circuitry
-

The board organization is showing in Figure 1.1:

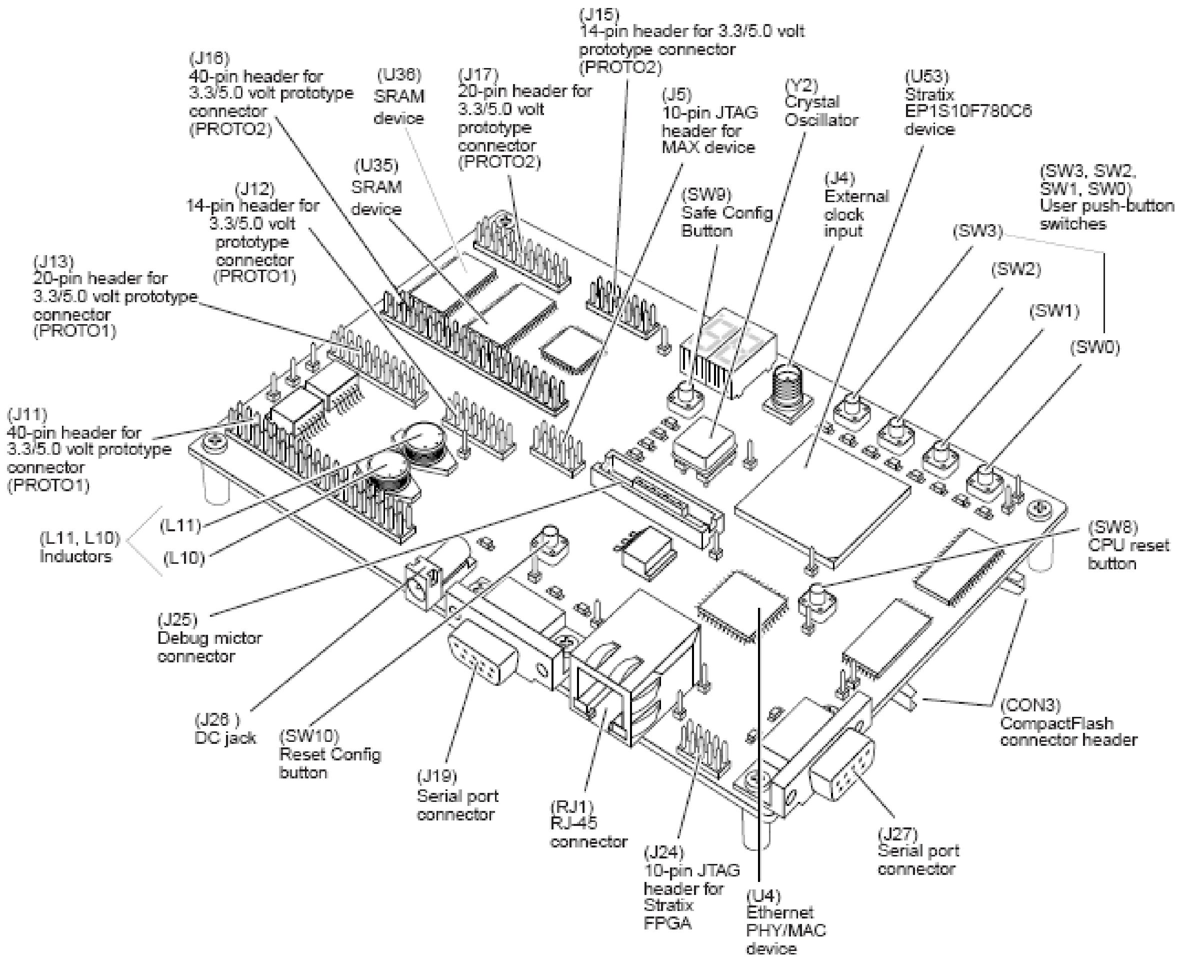

(J15) 14-pin header for 3.3/5.0 volt prototype connector (PROTO2)
(J16) 40-pin header for 3.3/5.0 volt prototype connector (PROTO2)
(U36) SRAM device
(J17) 20-pin header for 3.3/5.0 volt prototype connector (PROTO2)
(J5) 10-pin JTAG header for MAX device
(Y2) Crystal Oscillator
(U53) Stratix EP1S10F780C6 device
(U35) SRAM device
(J12) 14-pin header for 3.3/5.0 volt prototype connector (PROTO1)
(SW9) Safe Config Button
(J4) External clock input
(SW3, SW2, SW1, SW0) User push-button switches
(J13) 20-pin header for 3.3/5.0 volt prototype connector (PROTO1)
(SW3)
(SW2)
(SW1)
(SW0)
(J11) 40-pin header for 3.3/5.0 volt prototype connector (PROTO1)
(L11)
(L10)
(L11, L10) Inductors
(SW8) CPU reset button
(J25) Debug mictor connector
(J26) DC jack
(SW10) Reset Config button
(J19) Serial port connector
(RJ1) RJ-45 connector
(J24) 10-pin JTAG header for Stratix FPGA
(U4) Ethernet PHY/MAC device
(J27) Serial port connector
(CON3) CompactFlash connector header

Figure 1.1: NIOS development board [10]

The various components on board demonstrates not only the hardware programming capability within FPGA chip, but also the capability to expand memory to on-board SRAMs, communicate with peripherals with serial and parallel ports and display on 7-segments display and LEDs. These features enable us to project complex embedded applications.

1.2 Internal structure of Altera Stratix FPGA chip

The Altera Stratix EP1S10F780C6 FPGA chip features[11]:

- 10,570 logic elements

- 94 M512 ram blocks
- 60 M4K ram blocks
- 1 M-RAM block
- Total ram of 920,448 bits
- 6 DSP blocks
- 48 embedded multipliers
- 6 PLLs
- Maximum of 426 user I/O pins

These components can be visualized in Figure 1.2:

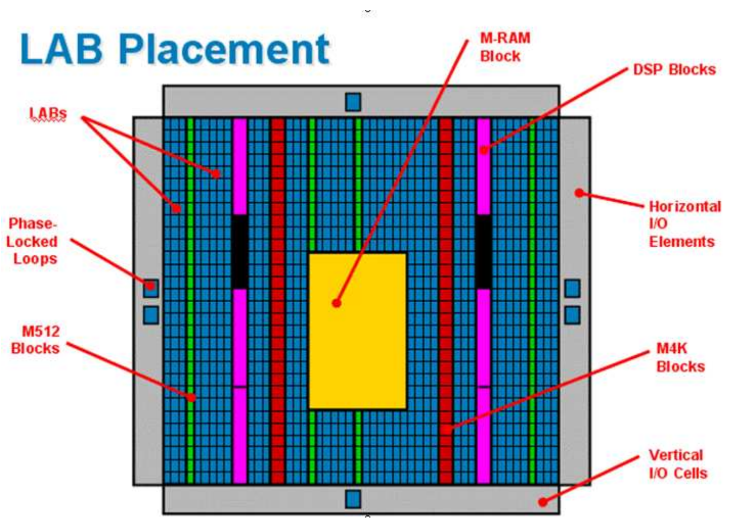


Figure 1.2: Internal structure of Altera Stratix EP1S10F780C6 [11]

High bandwidth buses are interconnecting logic array blocks (LAB) and memory elements.

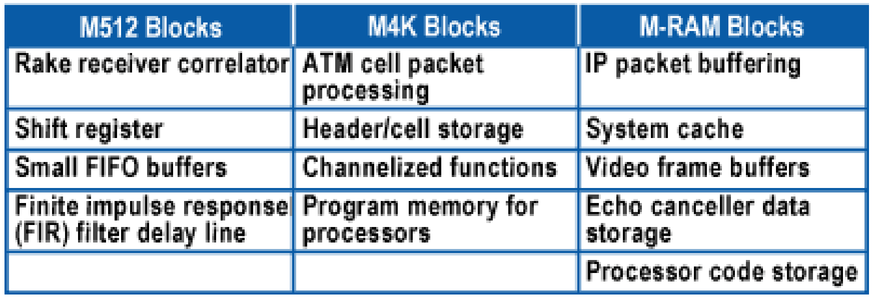
LABs, configured to function various digital logics, occupying most area in the chip. An IP core, for example the NIOS II soft-core processor, is downloaded to FPGA chip and mapped on LABs. An optimized design should keep its LAB placements closest-possible, avoiding occupying the bus of macro scale and reducing transmission delay. This optimization can be done manually, or by modern design environment like Altera Quartus II IDE.

Notice that smaller memory elements (M512 and M4K) are placed closer to the logic elements in order to achieve a higher access speed by adopting shortest-possible physical path, again avoiding complicated bus usage and reducing transmission delay, and therefore such memory access is more direct and can be achieved within one clock cycle. Such design made Stratix suitable for applications of intensive memory accessibility.

Several DSP blocks are facilitated to enhance performance on image processing, video compression or communication designs. These blocks are vendor-design, and therefore reduced design complexity. Another advantage of employing hard-cored DSP blocks is whether the LABs are 99% or 10% consumed, the DSP block will deliver the same high-performance.

1.2 TriMatrix Memory

TriMatrix memory is the on-chip memory. It is composed with RAMs of 3 sizes, including 512-bit M512 blocks, 4-Kbit M4K blocks, and 512-Kbit M-RAM blocks. Each of them can be configured for various applications of different memory access requirement, summarized in Figure 1.3.1.



M512 Blocks	M4K Blocks	M-RAM Blocks
Rake receiver correlator	ATM cell packet processing	IP packet buffering
Shift register	Header/cell storage	System cache
Small FIFO buffers	Channelized functions	Video frame buffers
Finite impulse response (FIR) filter delay line	Program memory for processors	Echo canceller data storage
		Processor code storage

Figure 1.3.1: Suggested usage of TriMatrix on-chip memory [12]

Mentioned in section 1.2, different types of on-chip memory element have different placements. It is a trade-off between memory size and access bandwidth, showing in Figure 1.3.2

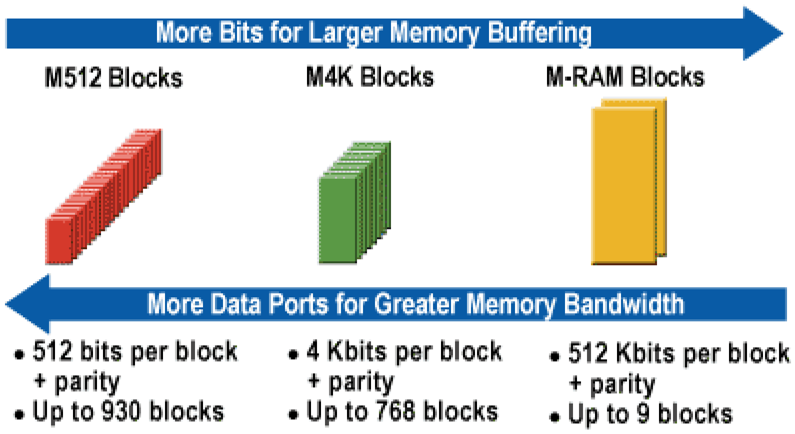

More Bits for Larger Memory Buffering
M512 Blocks
M4K Blocks
M-RAM Blocks
More Data Ports for Greater Memory Bandwidth
• 512 bits per block + parity
• Up to 930 blocks
• 4 Kbits per block + parity
• Up to 768 blocks
• 512 Kbits per block + parity
• Up to 9 blocks

Figure 1.3.2: Memory size versus access bandwidth [12]

Altera Stratix FPGA devices are produced in different versions with different pricing. Proper version is chosen according to design requirement and future expandability. One notable difference between versions is quantity and bandwidth of TriMatrix memory, summarized in Figure 1.3.3.

Device	Total RAM Bits	M-RAM Blocks	M4K Blocks	M512 Blocks	Maximum Bandwidth (Mbps)
EP1S10	920,448	1	60	94	1,245,024

EP1S20	1,669,248	2	82	194	2,096,928
EP1S25	1,944,576	2	138	224	2,894,400
EP1S30	3,317,184	4	171	295	3,750,192
EP1S40	3,423,744	4	183	384	4,384,800
EP1S60	5,215,104	6	292	574	6,762,528
EP1S80	7,427,520	9	364	767	8,784,720

Figure 1.3.3: TriMatrix memory specifications in Altera Stratix family [11]

1.3 Logic Elements and Logic Array

Logic gates are operations which act on one ore more logic inputs and produce a single logic outputs. All the logic components are mainly about the use of combination of AND, OR, NAND, NOR, NOT, XOR and XNOR gates.

In FPGA devices, logic gates functionality and syntax is constructed in Logic Elements (LE). In the first lecture, simplified LE with 4 inputs, 2 outputs, a Look-up Table (LUT) block, a multiplexer and a D flip-flop is introduced. LE actually has more inputs for timing and control signals and more outputs for complex functionalities.

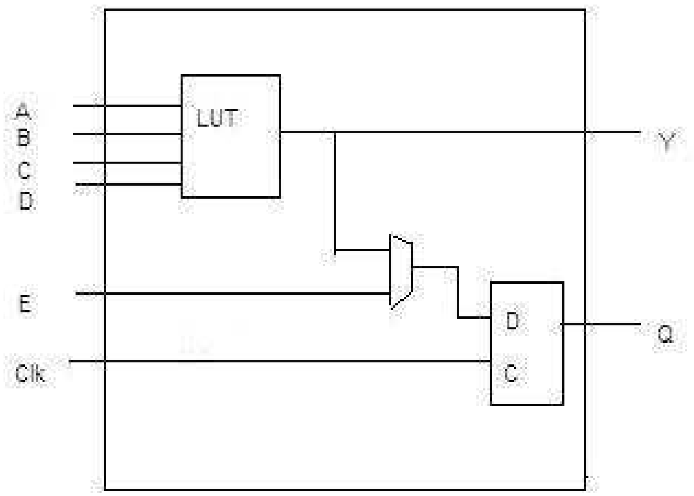


Figure 1.4.1: Logic Element structure

LUT takes inputs and generates an output according to control signals applied on pre-defined data structure. It can be configured to be a combinational logic.

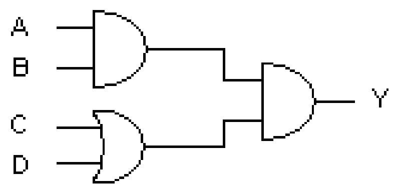


Figure 1.4.2: Configure LUT to be a combinational logic

It can also be configure to be a simple logic with large amount of inputs.

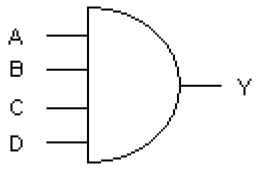


Figure 1.4.3: configure LUT to be simple logic

It is recommended to fully use the inputs of a LE. A configuration in Figure 1.4.4 is considered wasting resources of LE.

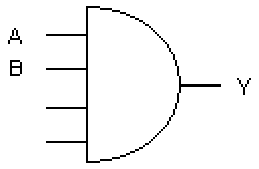


Figure 1.4.4: Configuration which wasting resources of a 4-input LE

Local interconnection of LEs form logic array block to construct complex logic, shown in Figure 1.4.5.

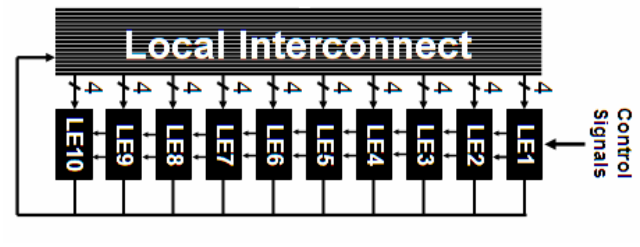


Figure 1.4.5 Logic Array Block [11]

The optimized placement allows LEs with close relation being located next to each other and therefore reduces the latency caused by interconnecting bus.

2. Overview to Altera NIOS II soft-core processor

2.1 FPGA Use in Embedded Application

During the lecture, three major trends of design, all employed a FPGA chip in it, are introduced, showing in Figure 2.1

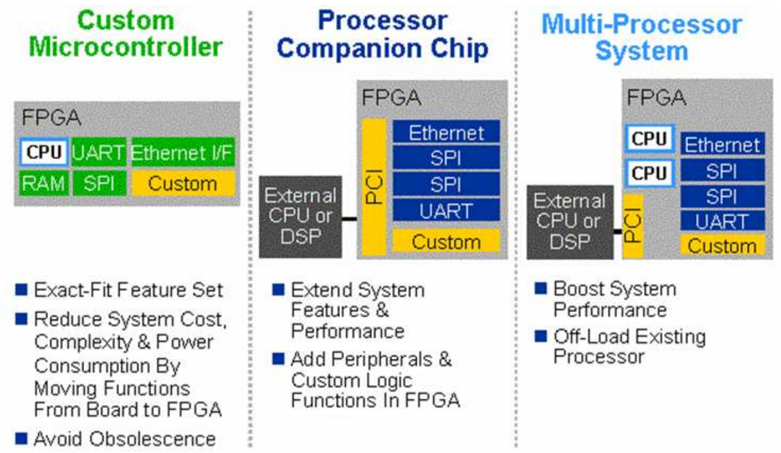


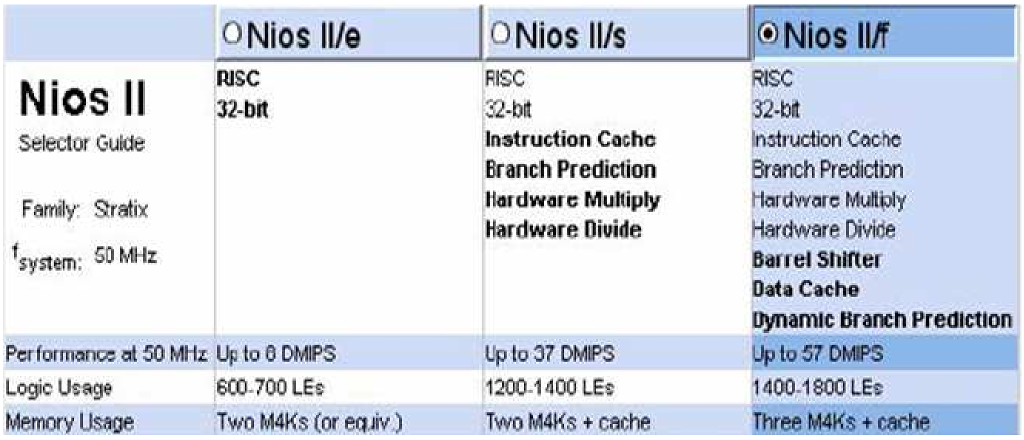
Figure 2.1: Different designs using FPGA [11]

A FPGA device may contain one or more CPUs beside controllers and on-chip memory. CPUs can be either hard-core (ASIC) or soft-core. An example of hard-core CPU in FPGA device is Xlinx Virtex 2 containing Power PC CPUs. NIOS II is a soft-core CPU built and distributed by Altera and can be implanted into FPGA device to provide CPU functionalities.

Advantages of the hard-core CPU are its fast clock speed and energy saving due to its ASIC nature. Soft-core CPU complements the disadvantages of the hard-core ones by its flexibility: it can be placed where it's optimized, closest to a controller, a memory block or a DSP block; as many as needed can be placed on same FPGA device when sufficient LABs are available; and each can be customized for different specialty since it is in fact a program?????.

2.2 Introduction to Altera NIOS II soft-core processor

Soft–core processor is a microprocessor fully described in software, usually in an HDL, which can be synthesized in programmable hardware, such as FPGAs. NIOS II from Altera is a soft-core processor with RISC architecture. It can be configured into no-pipeline (/e), 5 (/s) or 6 (/6) pipelines structure according to program specifications, showing in Figure 2.2.



Nios II Selector Guide Family: Stratix f_{system}: 50 MHz	○ Nios II/e	○ Nios II/s	◉ Nios II/f
	RISC **32-bit**	RISC 32-bit **Instruction Cache** **Branch Prediction** **Hardware Multiply** **Hardware Divide**	RISC 32-bit Instruction Cache Branch Prediction Hardware Multiply Hardware Divide **Barrel Shifter** **Data Cache** **Dynamic Branch Prediction**
Performance at 50 MHz	Up to 6 DMIPS	Up to 37 DMIPS	Up to 57 DMIPS
Logic Usage	600-700 LEs	1200-1400 LEs	1400-1800 LEs
Memory Usage	Two M4Ks (or equiv.)	Two M4Ks + cache	Three M4Ks + cache

Figure 2.2 Altera NIOS II configurations [11]

Altera NIOS II has a full data path, address space, 32 general-purpose registers, 32 external interrupt sources and supports a 32 bits instructions set distributed by the vendor. It can use a variety of on-chip memories and peripherals, and interfaces to off-chip devices. Software development environment is based on the GNU C/C++ tool chain and Eclipse IDE.

2.3 Scalability and placement

Scalability is a property of a system, a network or a process, which indicates its ability to handle the growing amount of work. One advantage of soft-core processor mentioned above is flexible placement in order to match certain requirement, thus soft-core processors such as NIOS II are highly scalable. It can suit a low-cost design which has one NIOS II CPU on Altera Cyclone FPGA chip; or a high-end design having several NIOS II CPUs on a Stratix II FPGA chip. In this course, multiple NIOS II processors will be implanted on a Stratix FPGA chip to achieve the parallel processing tasks.

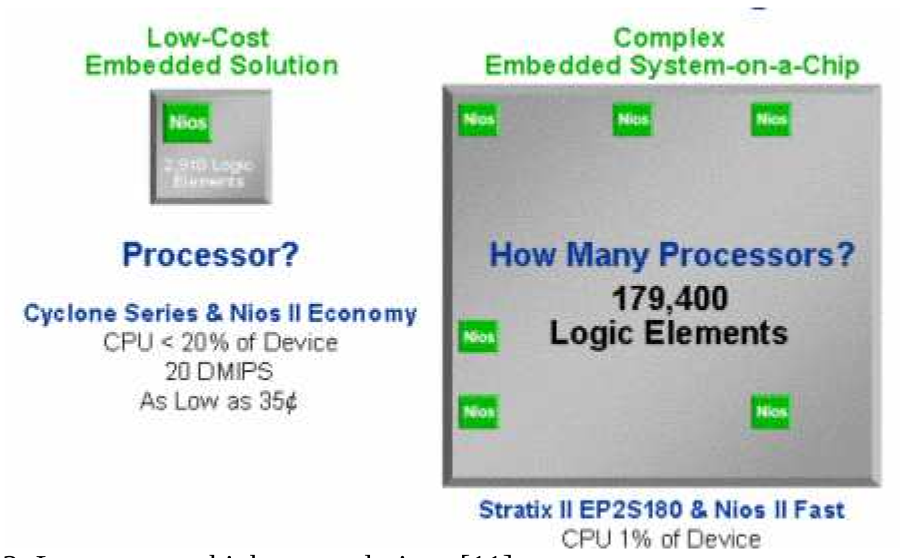


Figure 2.3: Low-cost vs. high-cost solutions [11]

Summary

The lecture presentation ends at slide 15. Next group of students will be responsible for further studying.

Paper 1 analysis Reconfigurable computing: architectures and design methods

Paper 2 analysis Network-on-chip architectures and design methods

Lecture #3: Cache Memory

Course: Computer Architecture III
Lecturer: Miodrag Bolic
Scribe: Samer Taweel 3405915, Alex Wong 3297818
Date: October 15, 2007

Introduction

Memory Hierarchy

Memory is a very important component of a computer, yet even more importantly is the type of memory in a computer's architecture. Having the wrong type of memory for the system can be costly.

There are many types of memory all of which must work together in a system to perform tasks efficiently. System efficiency is gained by having different levels of memory units that have different data transfer speeds. The top level there is a layer with slow memory access, this layer is normally a magnetic tape drives.

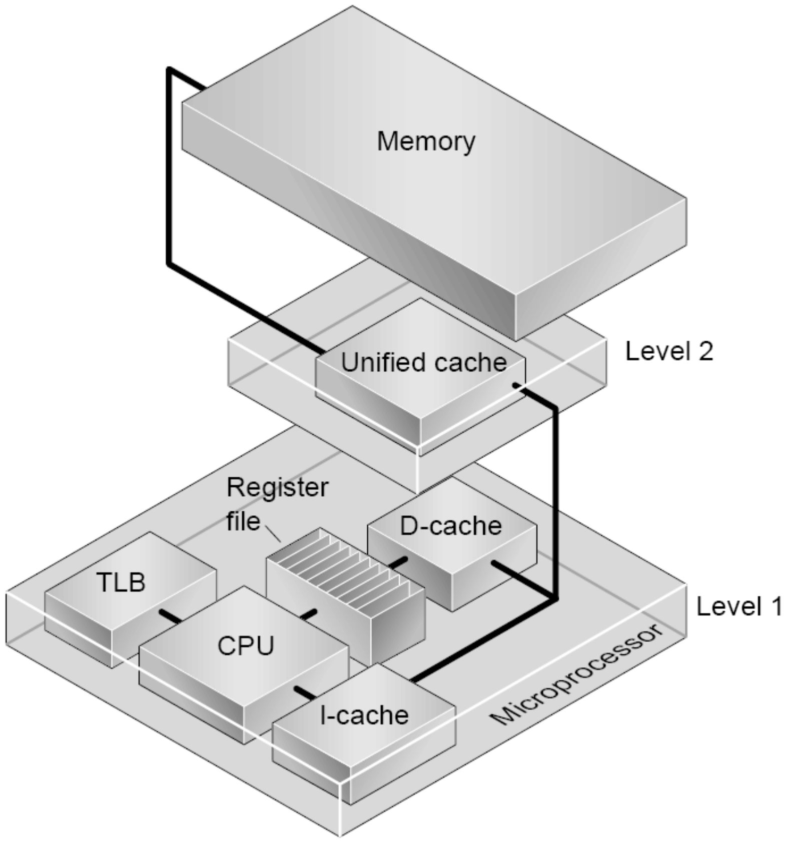


Figure 1. Memory Hierarchy [14] pp. 7

Magnetic drives although very slow at about 5,000,000 to 20,000,000 ns for a typical access time, are highly inexpensive ($0.5 to $2 per GB). This memory layer is the furthest from the CPU.

Level 2 has a faster level of memory access in comparison to magnetic tape drives. Typically this level is DRAM (dynamic ram) [15], which perform much faster then the pervious layer. DRAM has an average access time of 50 to 70 ns, but comes at a greater cost of $100 to $200 per GB of memory. In the memory hierarchy level 2 is closer to the CPU then the memory layer.

Level 1 in the memory hierarchy is the fasted memory. This memory is very close to the processor and has a very short access time. Typically SRAM (static RAM) [15] is located here because is has an access time of 0.5 to 5 ns but a cost of $4000 to $10,000 per GB. [17]

The memory hierarchy is designed this way because level 1 and 2 is almost always accessed before an instruction is preformed. If the data needed is located in these areas the instruction can be completed without a long fetch from the data memory. If the information is not in any of these layers, data must be transferred from the main memory which increases the wait time of the instruction [15]. To avoid this most cache based systems have two types of fast cache.

Instruction cache

The instruction cache is a cache that only holds program instructions. This will improve the speed of system because the instructions are not being fetched from the slow main memory. The instruction memory also holds important values such as branch locations and other instruction values that would slow the program down if it were to fetch them every time. [14] pp. 5

Data Cache

The other type of cache common in most cache based systems is the data cache. The data cache is used to store data that will be accessed multiple times. The job of the data cache is to provide the processor with the data as quickly as it can, by moving data for the program to the cache from main memory. When instructions are executed in a program their values are stored in the data cache. [14] Pp.5 the main memory will be corrected with the cached value at a later time.

Explanation

The Illusion of Fast Memory.

The ultimate goal of memory hierarchies is to present the end user with as much memory as is available in the cheapest technology, while providing access at the speed offered by the fastest memory. [16] pp. 461 When a memory system is designed using a hierarchical structure, the CPU has the perception that all memory blocks are stored in a faster, more expensive static RAM. In reality the memory blocks are stored in the slower, less expensive dynamic RAM. The CPU and end user are under this perception because the needed blocks get moved to from the main memory to cache, one way to achieve this is to use cache locality to improve hit rates. [19]

Locality

We can use one of two locality strategies for optimizing the use of cache. First we can use temporal locality. In this strategy contents that will frequently be used from main memory are transferred to cache memory, causing performance critical information to be closer to the CPU, taking advantage of a hierarchal design. An example of this would be systems that execute loops with data that is frequently accessed. Without cached memory the constants will have be fetched from the main memory every time an instance of the loop is preformed. This fetch takes time and cases the CPU to wait for a value. With cached memory the constants are close to the CPU and don't require a long fetch time.

A second locality strategy is spatial locality. In this strategy, the cache is filled with data that is likely to be used by a program before it is addressed by the program. The cache predicts what the program will need and stores it so that if the data is needed there will be a hit in the cache. This again causes information that will be used to be closer to the CPU. An example of this is having a loop that goes though an array of information. Once the first array element in an array is requested by the CPU the next few array elements are stored in the cache. This is done because there is a good chance that the next values of the array will eventually be requested by the CPU.

Memory design

There are many differences between cached memory and main memory in a system, other then the speed and cost. An important difference is the structural design of the two. Main memories have a basic structure with blocks of words, each block is broken into K words. Blocks are broken into words because frequently an instruction will want more then just a few bits of data. Having data stored in blocks causes the system to take more then what is needed incase more is needed. If not enough data is in a block, the result will be another miss, forcing a costly fetch. Each block of words is addressed with a linear approach. This linear approach takes a longer time to locate information. In figure 2 the main memory addresses are stacked on top of each other.

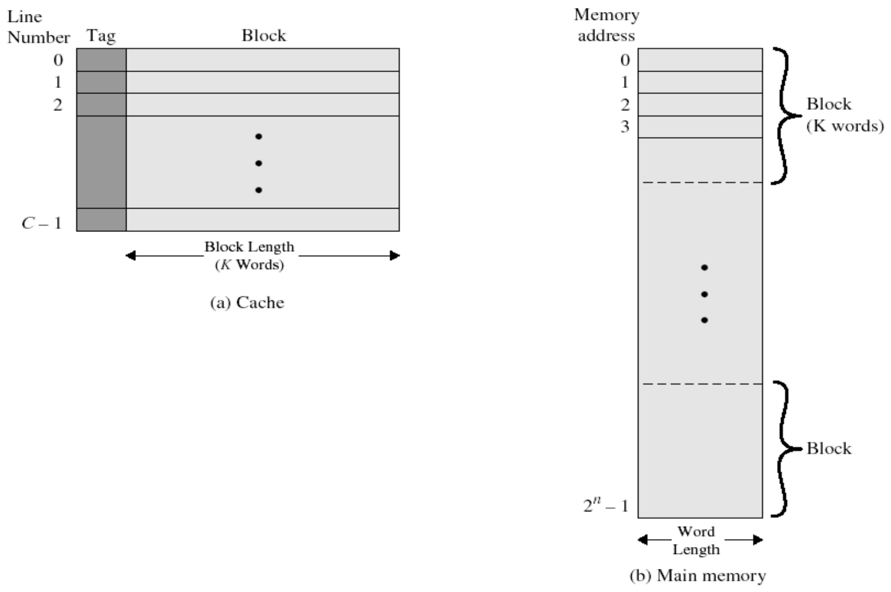


Figure 2. Cache and Main Memory Organization [17]

Much like main memory, cache memory is organized with address and blocks. But in most cases of cached memory the blocks are arranged in a non-linear structure (non-linear referring to set associative designs)   There are three ways to transfer data from main memory to cache.

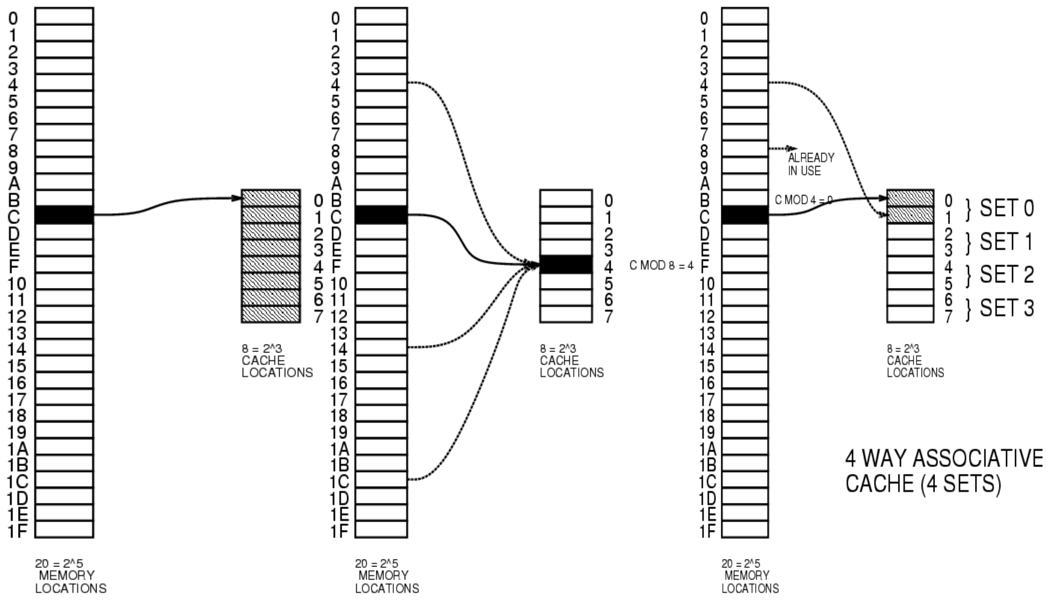


Figure 3.a Fully associative

Figure 3.b Direct mapping

Figure3.c Set associative

Figure 3. Different Cache Mappings [18]

Fully Associative mapping

In this memory placement method the block entering the cache can be placed anywhere. Figure 3.a, shows the basic structure of a fully associative mapped cache. Cached data can be placed in any block within the cache. Tags will later be used to relocate the cached data. This structure is very easy to implement but has a longer fetch time because all the data tags must be compared to the tag in the address field in order to relocate and use the data.

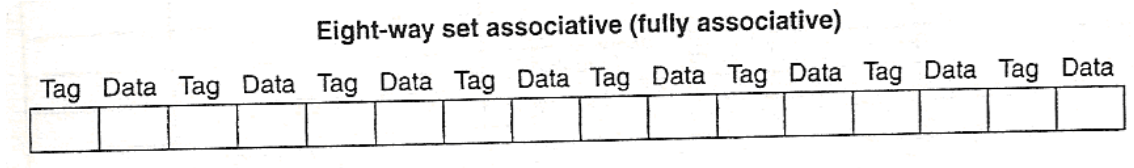


Figure 4 Fully Associative Mapping on an 8 block cache [16]

Direct mapping

In direct mapping each block of data can only be placed into one block in the cache. Figure 3.b shows that multiple main memory elements can be stored in the same cache location, but they can never be stored at the same time. If two main memory elements have to be stored in the same cache location, we overwrite one of them. This location is given to us be applying the formula:

Block number = (block address) MOD (Number of blocks in cache)

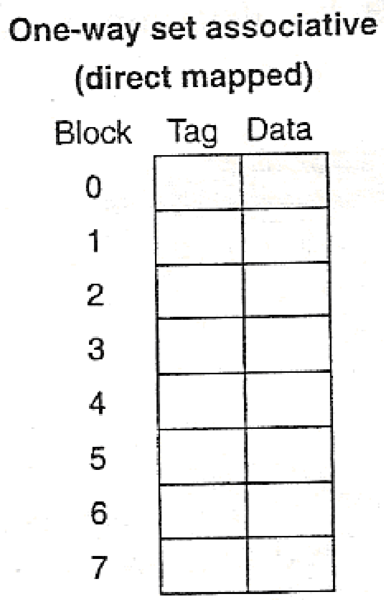


Figure 5 Direct Mapping on an 8 block cache [16]

When applying the block number formula for a direct mapped cache to a memory location, take the block address from the main memory and perform a modulus to the number of blocks available in the cache. In this case we have to use 8 as the "number of blocks in cache" value. The cache will use this formula to relocated stored data.

As you can see by Figure 3.b this method of block placement causes some problems because there is a possibility of the same cache location wanting to be used by different memory addresses. If for example block addresses 1C and 14 want to be used in order from the main memory. Applying the block number for a cache formula will result in block number 4. This will cause a miss because main memory address block 1C will be in cache memory block 4 when main memory block address 14 needs to be used. This problem can be avoided using a set associative mapping.

Set Associative Mapping

In a set associative data placement method the pervious problem cannot be fixed but helped. Figure 3.c shows a 4 way set associative cache, in set associative caches the block numbers are divided into sets. 4 way set associative caches are divided into 4 sets (Figure 6.a), while 2 way set associative caches are divided into 2 sets (Figure 6.b) The following formula is used to figure out which block set the data entering the cache will be placed into.

Set Number = (Block address) MOD (number of sets in Cache)

The set number will be the location in the cache that the data will be stored. This is derived by taking the block address from the main memory and performing a modulus to the number of sets available in the cache. For a four-way set associative cache figure 6.a we have to use a value of 4 for the "number of sets in cache" while for a two-way set associative cache (figure 6.b) we have to use a value of 2.

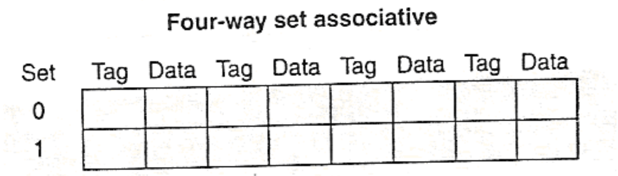


Figure 6.a Four way set associative [16]

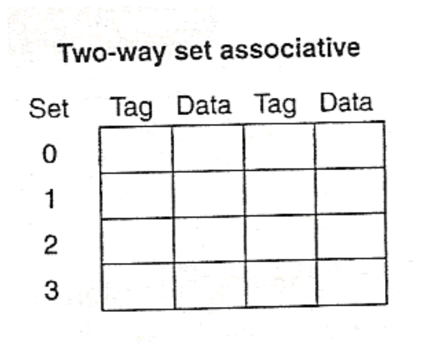


Figure 6.b Two way set associative [16]

When using direct mapping if there are two addresses pointing to the same location in cache, the old data will have to be replaced by the new data when it is needed.

With set associative mapping there can be more then one memory address pointing to the same set in the cache but there are multiple locations in the set that data can go into. This causes multiple memory addresses that would normally point to the same cache location point to different locations. In the pervious example, both 1c and 14 will be able to be stored in the same set. In a two-way set associative cache, a third address block pointing to the same place will have to over write one of those locations. In a four-way set associative cache a third and fourth block address from the main memory would be able to be stored in the cache.

Cache Design

The cache is designed in such a way that you can easily find saved information in it. There are 4 fields in a cache [17].

- Data: normally 32 bits of data, this is the contents that was in the main memory before entering the cache. The size is dependent on the system, but is normally 32 bit because most processors currently take in 32 bits of data at a time. The future trend will be to extend this to 64 bit processing.

- Tag: this part of the cache is used to compare with the address to ensure that this is the same data that is being requested. If the tags from the address field and the cache are different then they are not the same data. The size of the tag field is given by:

  Size of Tag = address size – ($\log_2$(Memory size) + offset size)

  Or by : **Size of Tag = address size – index size – offset size**

  The address side refers to the size of the address field. The index size is always be given by $\log_2$(Memory size) because this will give you the number of bits needed to index the cache. The offset size is used to multiplex what part of the data should be set to the processor. Both formulas produce the same output.

- Valid bit: the valid bit is a single bit that is initialized to 0 until the content of the cache is changed. Once the contents of that cache location is valid the bit changes to 1.

- Dirty bit: this bit indicates if the contents of the cache are still the same as what's in the memory (by having a value of 0), or if the cache contents have changed from what's in the main memory (by having a value of 1). This is very important in write strategies, but the bit will not be shown in figure 7.

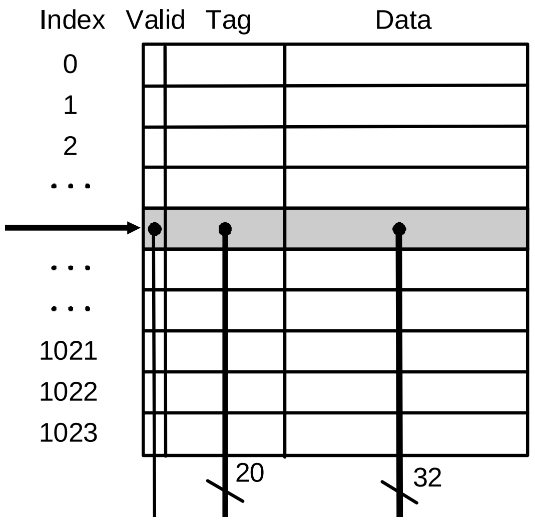


Figure 7 Cache fields [17]

Address Fields

The address field is used to locate data in the cache. Here are the following fields (figure 8)

- Block offset: the block offset is used to select which part of data is needed from a block and the size is calculated by

$$\textbf{Offset} = \log_2(\textbf{block size})$$

- Block Address: is denoted by the tag and the Index.
    - The index gives the address within the cache. This will tell the computer where to look and it’s size is calculated by:

      $$\textbf{Index Size} = \log_2(\textbf{number of blocks / associativity})$$

    - The Tag in the address field is the same as in the cache.

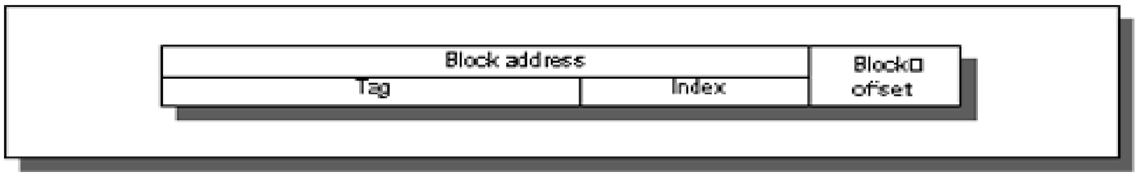


Figure 8 Cache Address Field [17]

Locating data

When data is needed for the CPU to perform an instruction there are a set of steps that are followed [16]:

- CPU requests contents of memory location
- Check cache for this data
    - This is done by comparing the Tag from the address field and the tag that is in the cache at location declared by the index
    - If the tags are the same. And the valid bit is set to 1 (valid data) then the data is present.
- If present, get from cache (fast). When data is in the cache it is called a hit.
- If not present, read required block from main memory to cache, this is called a miss
    - In the case of a miss there must be some kind of replacement strategy that is in place to replace information in the cache.
    - **Least recently used (LRU)** removes the least recently used block of data from the cache and replaces it with the new data from memory. The theory is that if the data hasn't be used in a while it wont be used again.
    - **Most recently used (MRU)** removes the most recently used block of data from the cache and replaces it with the new data from memory. The theory behind this is that if the data was used already it won't be used again.
- Then deliver from cache to CPU
- Cache includes tags to identify which block of main memory is in each cache slot

If an operation is done and there changes made to the cache contents then a write strategy must be put into place. We've learned about two different strategies

Example of locating data with a four way set associative cache

After the processor requests a given memory location, the cache will perform several tests to determine if the cached value is correct. The first step is to check which set to check within the cache. The follow formula is applied:

Set Number = (Block address) MOD (number of sets in Cache)

Once the set is located in the cache, the tags of all 4 blocks are compared with the tag in the address field. If nothing matches, we have a miss and must obtain the data from main memory. If there is a match, the valid bit is check to ensure nothing has changed from what's in the main memory. If the valid bit is a 0 there has been a change and we have a miss. Otherwise if the valid bit is a 1, we have a hit.

Assuming a miss has occurred, the data must be fetched from the main memory and a loading strategy must be applied. This will place the correct data in the cache. Once the data is in the cache, the same process applies for a hit. The offset bits will multiplex the needed data from the cache to the processor.

Write back

In a Write back strategy the changed information is only written to the cache memory. This is when the dirty bit is needed. If the dirty bit is 0 then the data in the cache is the same as in the main memory, if the dirty bit is 1. Then the data in the cache is different from what's in the main memory. In this case the cache block is only written to memory when the content in the cache is being replaced. This saves many cycles and memory traffic. But it is a harder system to implement in a design. If you are building a system that relies on speed then write back is a better solution. If speed is not an issue for the system then write thought might be a better strategy.

Write Through

In a Write through strategy there is no need for a dirty bit because all the information that is changed in the cache is automatically changed in the main memory. This takes any confusion away from what the memory locations hold because they are always the same. The down side is that it is a slow procedure and takes time every time a value is changed.

Performance

Here are some formulas that are used for measuring the performance of a CPU [17]

CPU time = (CPU execution clock cycles + Memory stall clock cycles) × Clock-cycle time

Memory stall clock cycles = Read-stall cycles + Write-stall cycles

Read-stall cycles = Reads/program × Read miss rate × Read miss penalty

Write-stall cycles = (Writes/program × Write miss rate × Write miss penalty) + Write buffer stalls

The CPU time is the overall time it takes to run a given portion of code. This is important to know to for timing and analysis of systems. The "CPU execution clock cycles" is the number of clock cycles it will take to execute an instruction of code. This value is added to the number stalled cycles that the system will have to perform to keep the program running efficiently, stalls refer to all the overhead of having a system. There must be clock cycles taken away from the operation of the program to load, store and read data. . Finally the summations of the clock cycles are multiplied by the clock time to get the amount of time needed to perform the executing code.

Summary

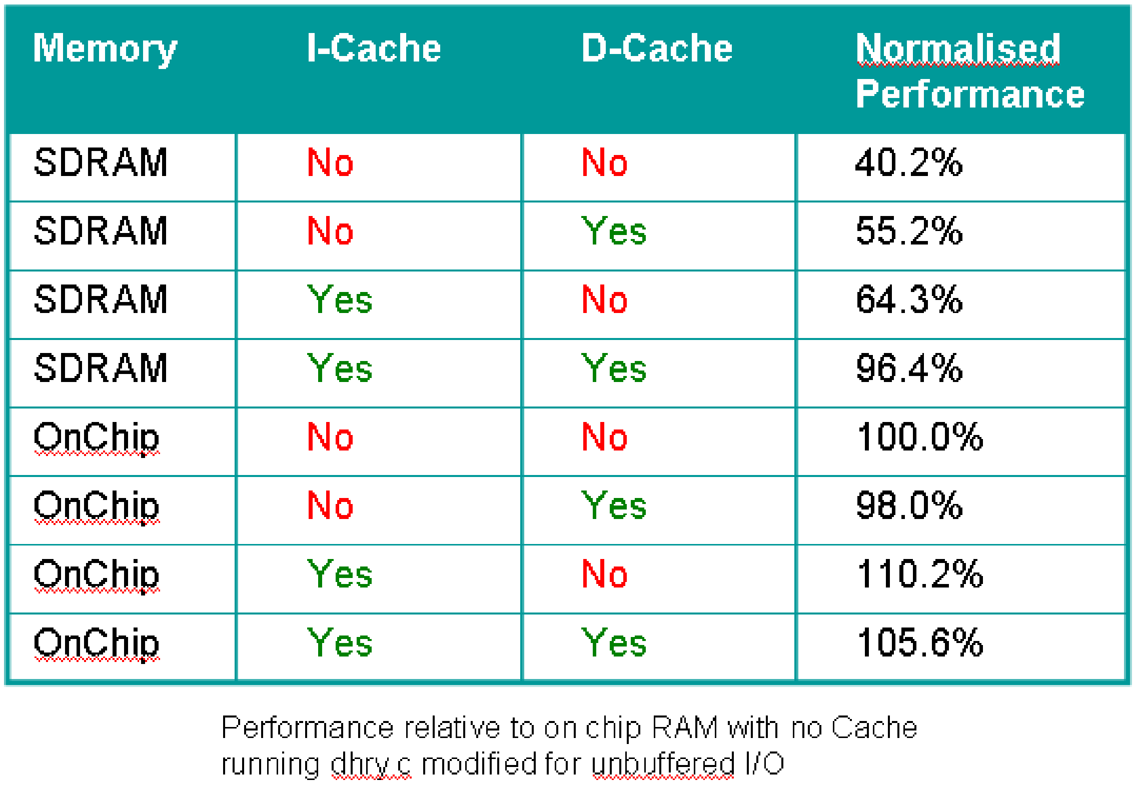
NOIS II Cache Memories

NIOS II processors support two different types of directly mapped cache memories. The first is instruction cache and the second is data cache. The cache is located on chip and provides a decrease of memory access. The configurable caches are managed using software and NOIS II has a complete instruction set to do so [20]. An example of this instruction set is the stio function, which will bypass the data cache and go directly to a specified address.

Using cache memories in a NOIS II processor will only affect the performance if the main memory is located off chip. This is because the access time of an off chip memory is much longer then that of an on chip memory. Cache memories will also improve performance if the largest, performance critical instruction loop is smaller then the instruction cache, or if the largest block of performance critical data is smaller then the data cache [20].

The NIOS II soft core processors are configurable to the exact requirement of the designed system. The cache size can be altered to affect the performance of the system. Smaller cache sizes are useful for conserving on-chip memory resources but cause slower system, due to larger miss rates. Larger cache sizes allow for a decrease in miss rate, but take a larger toll of no-chip resources [20].

Figure 9 displays results of a test program that was run using different cache and memory settings. There is a very small increase of performance when using on chip memories with data cache or instruction caches. This is because there will be an unlikely gain in performance for memory that is already on chip. When using off chip memories cache will improve the performance. This increase of performance will happen because the off chip memory is slow and accessing the on chip cache will increase the performance [17].



Memory	I-Cache	D-Cache	Normalised Performance
SDRAM	No	No	40.2%
SDRAM	No	Yes	55.2%
SDRAM	Yes	No	64.3%
SDRAM	Yes	Yes	96.4%
OnChip	No	No	100.0%
OnChip	No	Yes	98.0%
OnChip	Yes	No	110.2%
OnChip	Yes	Yes	105.6%

Performance relative to on chip RAM with no Cache
running dhry.c modified for unbuffered I/O

Figure 9 Cache performances on a NOIS II processor [17]

Paper 1 The Processor-Memory Bottleneck

Paper 2 Cache Performance Analysis of Algorithms

Lecture #4: Performance Analysis of Multiprocessor Architectures

Course: Computer Architecture III
Lecturer: Miodrag Bolic
Scribe: Kenneth Loo 3019619, Manpreet Singh 2619107
Date: September 22, 2006

Introduction

Performance analysis is the investigation of a program's behavior using information gathered as a program runs, as opposed to static code analysis. The usual goal of performance analysis is to determine which parts of the program to optimize for speed or memory usage. [27] Determining the performance of these multiprocessor architectures requires a defined standard of performance analysis, these analyzing schemes are standard across the industry so that each architecture knows where it stands when compared to each other. This type of analysis is known as Benchmarking. When analyzing the performance of multiprocessor architectures, parallel processing, efficiency, speedup and scalability must be taken in to account. Converting from serial processing to parallel processing great improvement in performance can be made to programs that can be executed in parallel. Those programs that can only be executed in sequential form can still be improved but in a different manner.

Explanation

Performance Measurement
Speedup[26]

Speedup is a ratio which compares the execution time of a chosen algorithm in parallel to that of the best corresponding execution time in serial. Generally, this is the comparison of a single processor system versus a multiprocessor system. The formula for calculating the speedup ratio is $S_n = T_1/T_n$. Where S_n is the speedup ratio, T_1 is the excution time of the single processor system and T_p is the excution time of the p-processor system.[32]

The formula for calculating the ideal speedup is $S_n = n$. Usually the ideal speedup of a p-processor system is p, where p is the number of processors in the system. There are cases where speedup exceeds the number of processors. This is called Super Linear Speedup. [25]

Efficiency

Efficiency is the measure of the speedup that is achieved per processor added when converting a single processor system to a multiprocessor system.

The formula for calculating the efficency is $E_p = S_n/n$. Where E_n is the efficency of the n-processor system and S_n is the speedup of the n-processor system.
p is the number of processors. [31]

An efficiency of 1 means that linear speedup has been achieved, while an efficiency of 0 means that no parallelism could be achieved. In the case of Super Linear Speedup, an efficiency of greater than 1 can be achieved through parallelism and the effective use of different cache sizes. The cache effects come into play when the size of the sub-domain becomes small and the variables accessed frequently fit into the cache. [23]

Scalability[31]

Scalability is defined as the proportional performance increase realized from the addition of processors while maintaining the same efficiency.

This can be explained more easily through the use of the following example. The addition of *m* numbers is done using *n* processors and communication and computation take one unit of time.

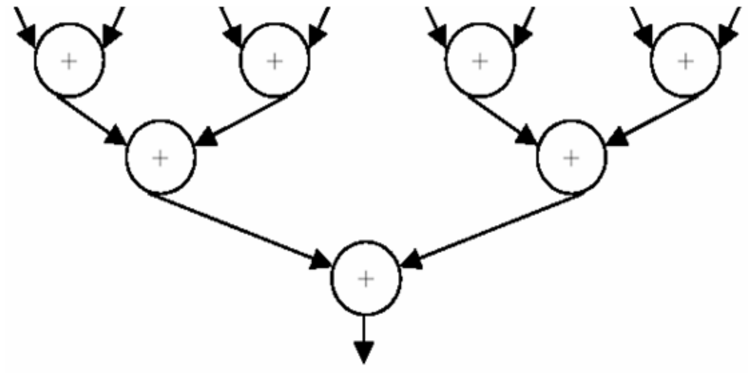


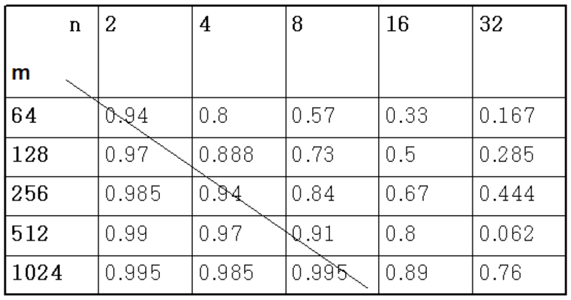
Figure 5: Scalability Example

Each processor adds two numbers as is shown in Figure 1 and the results are added again by another processor and so on.

Using the equation $S = m / m / n+2\log_2 n$ we can determine the speedup for a number of cases as shown in Figure 2.

m	n=2	n=4	n=6	n=16	n=32
64	1.88	3.2	4.57	5.33	5.33
128	1.94	3.55	5.82	8.00	9.14
256	1.97	3.76	6.74	10.67	14.23
512	1.98	3.88	7.31	12.8	19.70
1024	1.99	3.94	7.64	14.23	24.38

Figure 6: Speedup for different values of m and n



n / m	2	4	8	16	32
64	0.94	0.8	0.57	0.33	0.167
128	0.97	0.888	0.73	0.5	0.285
256	0.985	0.94	0.84	0.67	0.444
512	0.99	0.97	0.91	0.8	0.062
1024	0.995	0.985	0.995	0.89	0.76

Figure 7 Efficiency for different values of m and n

From the values of speedup we can determine the values for efficiency as shown in Figure 3. The line on Figure 3 shows where efficiency remains the same and demonstrates scalability. For 4 processors to maintain the same level of efficiency, 256 numbers must be added, compared to 2 processors and 64 numbers.

There are advantages and disadvantages to scaling up a system. While scaling up a system you could increase speedup and efficiency but the amount of hardware and synchronization required to coordinate the extra processors could actually result in slowing down the system.

Parallel Computing[28]

"Parallel computing is the simultaneous execution of the *same task* (split up and specially adapted) on multiple processors in order to obtain results faster. The idea is based on the fact that the process of solving a problem usually can be divided into smaller tasks, which may be carried out simultaneously with some coordination."[1]

[1] http://en.wikipedia.org/wiki/Parallel_computing

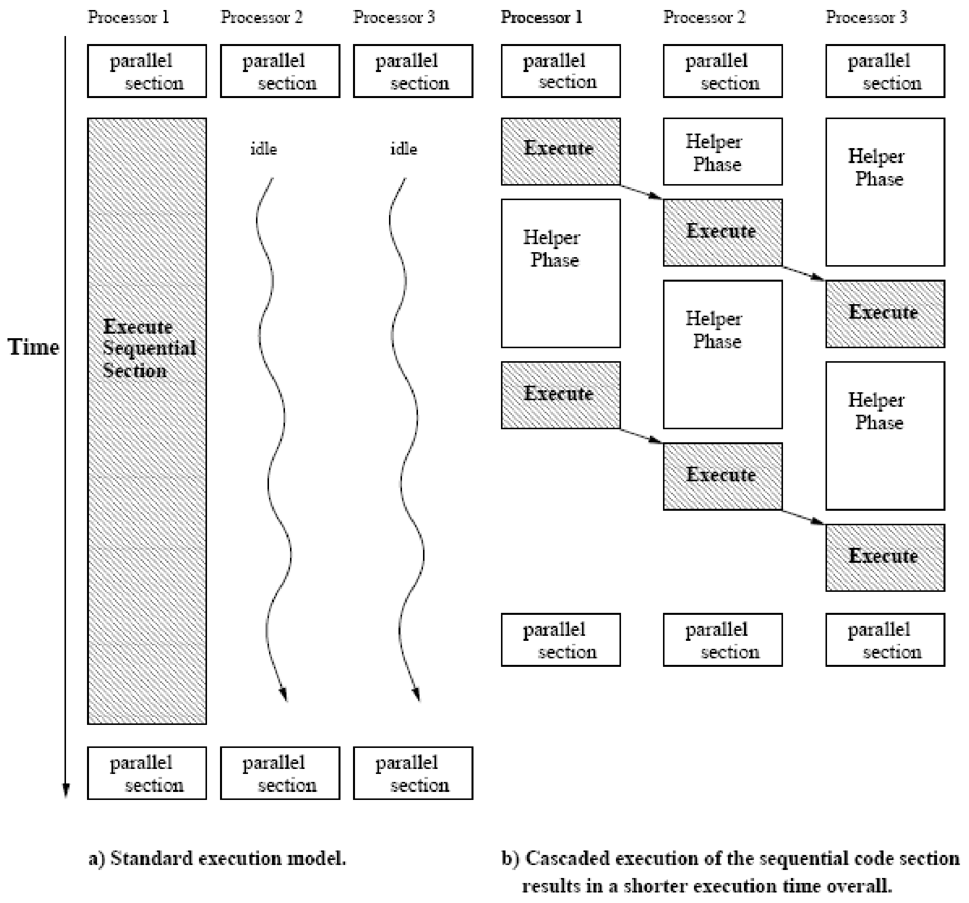

Processor 1
Processor 2
Processor 3
Processor 1
Processor 2
Processor 3
parallel section
Time
Execute Sequential Section
idle
idle
Execute
Helper Phase
a) Standard execution model.
b) Cascaded execution of the sequential code section results in a shorter execution time overall.

Figure 8 Standard Execution vs. Cascaded Execution

As shown by Figure 4, the standard execution model only benefits when there are parallel sections of code. Sequential sections are run by one processor only. Using cascaded execution, these sequential sections can be run on multiple processors reducing run time.

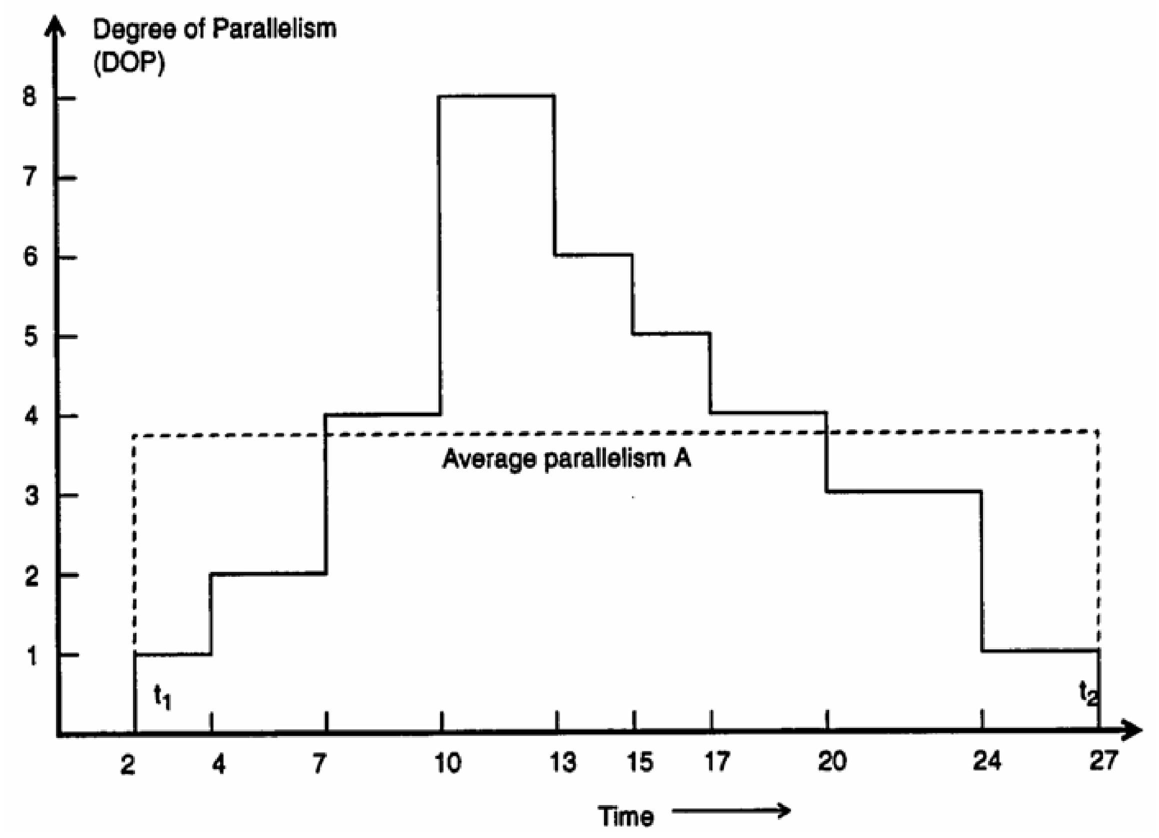


Figure 9 Degree of Parallelism Achieved

Figure 5 shows the average parallelism achieved by the system given a certain program. At times where 8 degrees of parallelism are not achieved, the other processors would be idle in contrary to the cascaded example.

Amdahl's Law[31]

Amdahl's Law states that the speedup from a faster mode of execution is limited by the amount of time the faster mode can be used. In the case of parallel processors, this faster mode is where execution can be done in a fully parallel mode.

Recall the speedup formula, $S_n = T_1/T_n$. Let us define β as execution in pure sequential mode and 1-β the probability of a fully parallel mode using n processors.

This gives us that $T_n = T_1\beta + T_1(1-\beta)/n$. $T_1\beta$ measures the sequential part execution time, while $T_1(1-\beta)/n$ measures the time required for the parallel part execution time given n number of processors.

Plugging back into the speed up equation, we get that $S = 1/\beta+(1-\beta)/n$ which can be rewritten as $S = n/\beta n + (1-\beta)$.

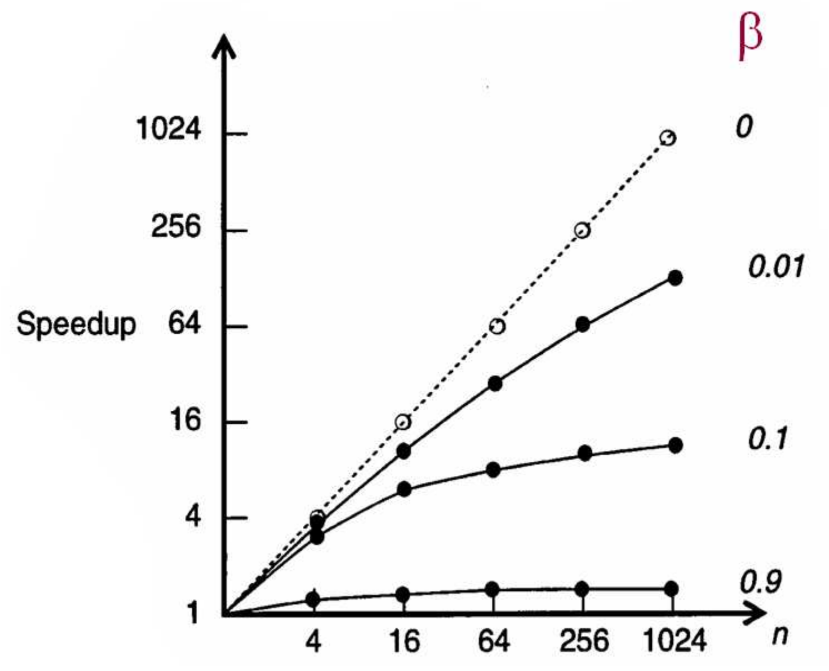
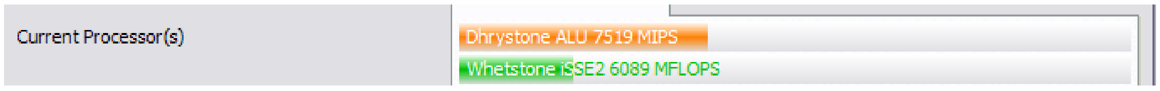


Figure 10: Speedup versus number of processors

Figure 6 shows the speedup versus the number of processors given the amount of pure sequential execution.

Benchmarks

Benchmarks are a standard used to measure the performance of a system. These results can then be used to evaluate the performance of the said system and compared to others. Some benchmarks are geared to test only one part of the system, like Dhrystone, while some test the overall performance of the system, like SPEC. Dhrystone benchmarking consist of all integer operation and no floating point operations. The inverse of this is known as the Whetstone benchmarking where all the operations consist of floating point operations. The Dhrystone and Whetstone number is translated to the number of iterations of the main code loop per second. Both of these benchmarks are synthetic meaning that they are simple programs that were carefully designed to mimic very common programs.



Current Processor(s)

Dhrystone ALU 7519 MIPS

Whetstone iSSE2 6089 MFLOPS

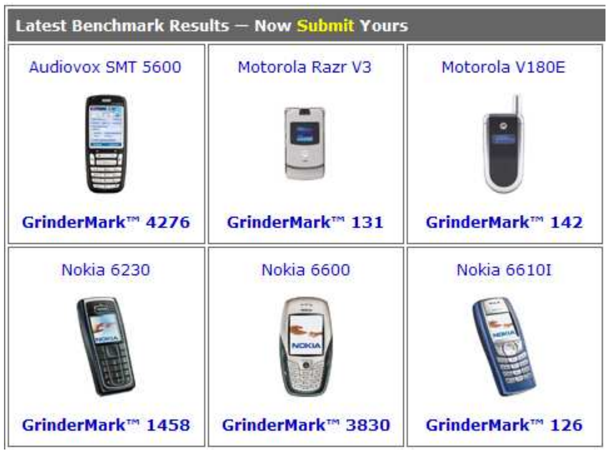
Figure 11 Dhrystone Performance

Figure 7 shows the Dhrystone and Whetstone measurement on a 2.1GHz computer. The problem with synthetic benchmarks is that the system can be optimized specifically for one particular benchmark.

SPEC stands for Standard Performance Evaluation Corporation which is a non profit organization that provides a mean of comparison between industry leading standards. SPEC benchmarks are usually coded in C or Fortran and the end company may use whichever compiler they wish for their platform without changing the code. There have been instances where manufacturers optimize their compilers for better performance from various SPEC benchmarks. The benchmarks aim to test "real-life" situations. For example take SPEC CPU tests CPU performance by measuring the run time of several programs such as the compiler gcc and the chess program crafty. The various tasks are assigned weights based on their perceived importance; these weights are used to compute a single benchmark result in the end. This is the result you see in the specifications when buying a computer or any other products.[29][29]

Summary

Benchmarks also exist for embedded applications. EEMBC stands for the Embedded Microprocessor Benchmark Consortium which is a non profit organization that aims to develop meaningful performance benchmarks for hardware and software which are used in embedded systems. [30]



Latest Benchmark Results — Now Submit Yours

Audiovox SMT 5600	Motorola Razr V3	Motorola V180E
GrinderMark™ 4276	GrinderMark™ 131	GrinderMark™ 142
Nokia 6230	**Nokia 6600**	**Nokia 6610I**
GrinderMark™ 1458	GrinderMark™ 3830	GrinderMark™ 126

Figure 12 Mobile Java Benchmark Results[33]

In Figure 8, each cell phone is given the same set of mobile applications to run and is timed to see how long it takes to finish. The higher the number, the less time was required to complete the suite. EEMBC benchmark scores are submitted to the certification lab before any benchmark scores can be released to the public. You must be a member of EEMBC to be able to use their exclusive benchmarks and you must be a member to be able to publish these EEMBC scores with your end product. [30]

Paper 1 Cascaded Execution: Speeding Up Unparallelized Execution on Shared-Memory Multiprocessors [23]

Paper 2 The Consequences of Fixed Time Performance Measurement [24]

Lecture #5: Parallel Computer Models

Lecturer: Miodrag Bolic
Scribe: Michel Proulx (2622193)
Darya V. Shapka (2897501)
Date: Friday, September 22 2006

Introduction

This lecture gives the overview of different classification systems used to describe the parallel computer architectures. For example, Flynn's classification system that is based on instruction and data streams. The classification systems can also be based on memory management, interconnection networks etc.

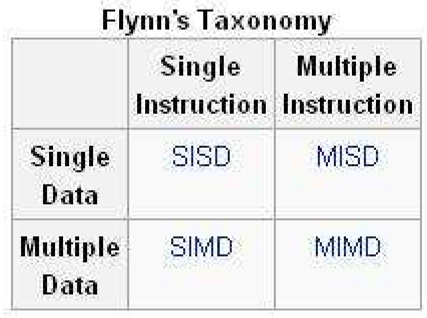
Some of the classification systems that will be discussed are also models. Models are constantly used in the design to provide the necessary level of abstraction. For example, continuous timing model used in the hardware design is a powerful model used for low-level design (i.e. working with transistors); yet, it is too detailed for a successful high-level design.

This lecture does not give details for different classifications. It is, rather, introduces students to different parallel computer architectures that will be discussed in details later in the course.

Explanation

1 Flynn’s Taxonomy

Flynn’s taxonomy is the most popular, universal and most used classification of computer architectures. It was introduced by Michael J. Flynn in 1966[34]. This classification is based on two types of information that flows into a processor: instructions (instruction memory to controller unit) and data streams (data memory to processor). According to Flynn’s classification, this information can either be single or multiple.



Flynn's Taxonomy

	Single Instruction	Multiple Instruction
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

Figure 13-Flynn’s Taxonomy classification table [34].

1.1 SISD ARCHITECTURE

The SISD model (Single Instruction Stream, Single Data Stream) is a sequential computer that exploits no parallelism in either the instruction or data streams. Some examples of this architecture are conventional single-processor machines like a PC or old mainframe.

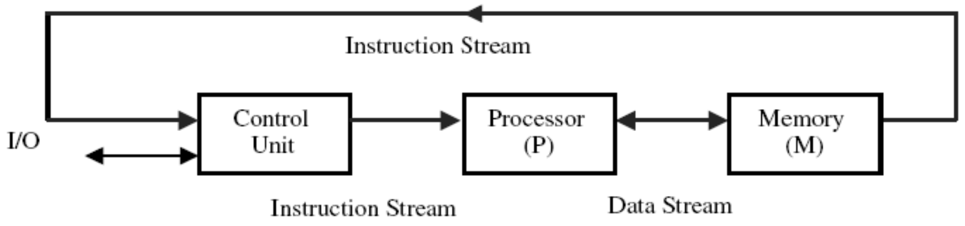


Figure 2 - SISD architecture [34]

1.2 SIMD ARCHITECTURE

The SIMD model (Single Instruction stream, Multiple Data stream) exploits multiple data streams against a single instruction stream to perform operations which may be naturally parallelized. There may be multiple processor units, and the control unit for this model is usually a typical serial processor. In order to use multiple processors to work on different data stream with one single instruction stream, the lock-step mode is used. This mode will determine if a specific processor will work with others on the same instruction stream, or if it simply will not do anything for that instruction set.

Example:

```
Inst 1
Inst 2
If (a>b)
    Inst 3
Inst 4
```

In this example, each processor in the system executes the same sequence of operations. i.e. all of the processors execute *inst 1* and *inst 2*. When the processors reach conditional statements, processors that do not need to perform *inst 3* will wait until *inst 3* is completed by other processors. Only after all of the processors are ready to execute *inst 4*, all of the processors will proceed with the execution.

This model is used for application with lots of data, like picture manipulation, where it divides data among processor units.

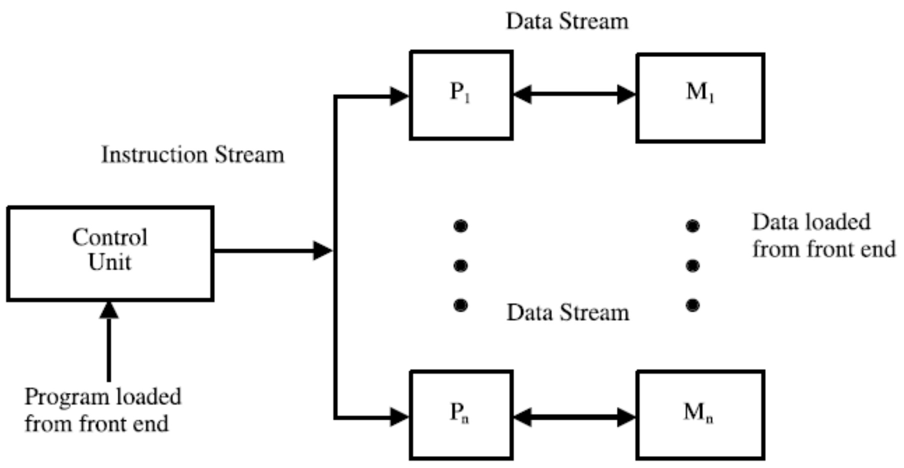


Figure 3 - SIMD architecture [34]

1.3 MIMD ARCHITECTURE

The MIMD model (Multiple Instruction Stream, Multiple Data Stream) is the most common architecture, in which it uses multiple processors in a parallel architecture, executing a single shared memory space or distributed memory spaces connected via some interconnecting network. This model employs independent instruction streams and independent data streams, where each processor usually has a unique control unit in order to process the independent instructions. The central controller can also be used to synchronize the operation of different processors. This controller will act as a master and will distribute the operations between other processors (slaves).

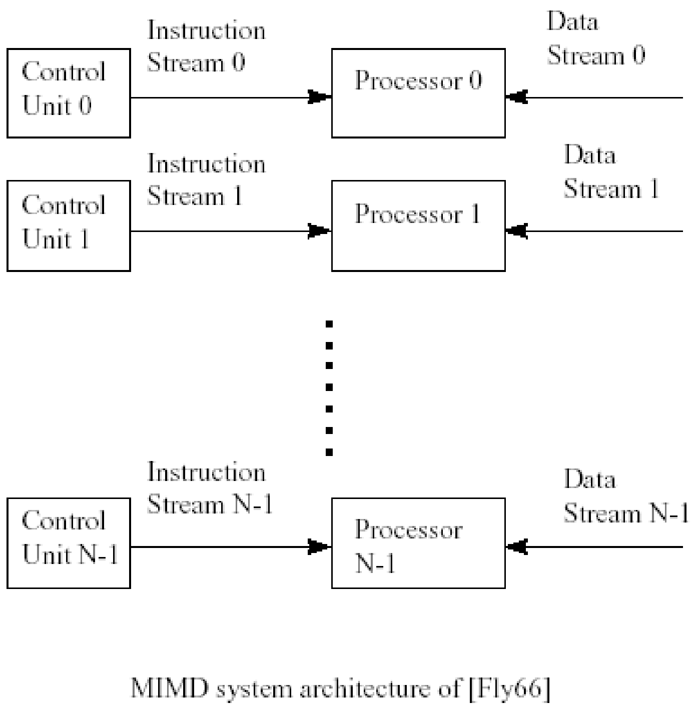


Figure 4 – MIMD Architecture [36]

Processors exchange information through their central shared memory in shared memory systems and exchange information though their interconnection network in message passing systems. We can see in figure 5 the two different types of memory architectures used. We will discuss these two memory systems in more detail in a further section.

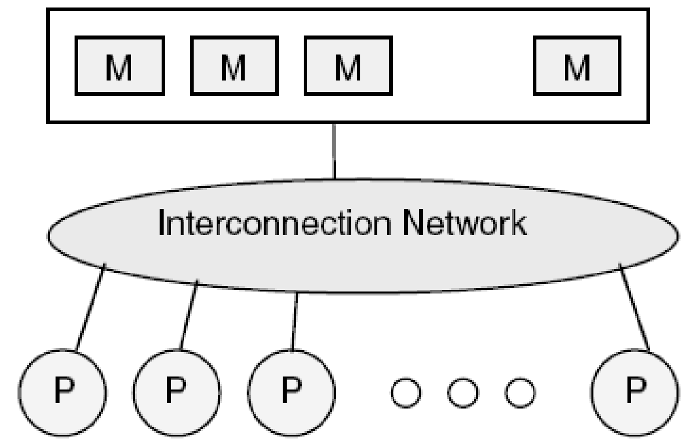


Figure 5a – Shared memory MIMD Architecture

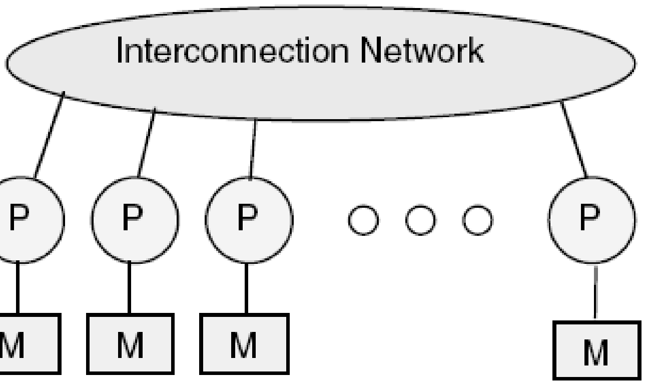
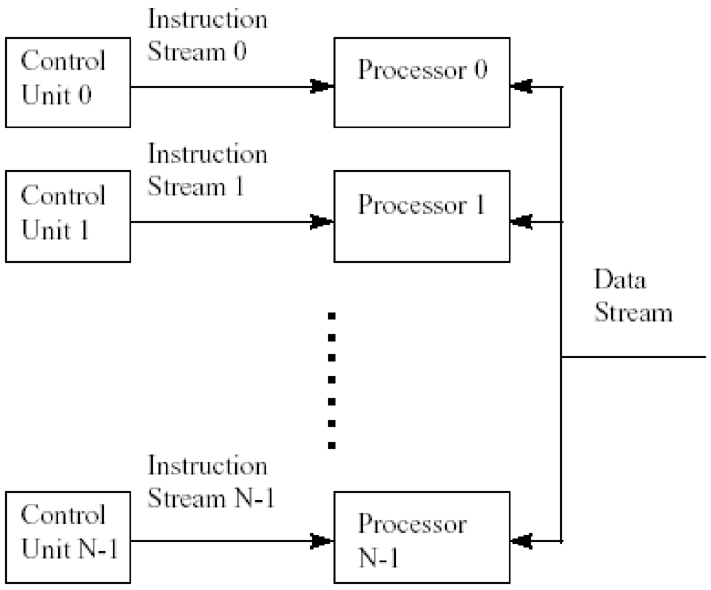


Figure 5b – Message Passing MIMD Architecture

Figure 5 – Shared memory versus message passing architectures [34]

1.4 MISD ARCHITECTURE

In the final model, MISD (Multiple Instruction Stream, Single Data Steam), the same stream of data flows through all processors executing different instruction streams. This architecture is not common and is not really known to have entered in mass production. One use for this model would be to test different algorithms on the same set of data.



MISD system architecture of [Fly66]

Figure 6 – MISD Architecture [36]

2 Shared-Memory Multiprocessors

A shared memory model is one in which multiple processors in a computer architecture communicate amongst themselves by writing and reading information in a shared memory space (See Figure 5a). This method can typically be seen as a bulletin board where you leave a message for everyone to see, but nobody else can post a new message at the same place on the board until the previous message is removed. The shared memory system in a large multi-processor system loses its efficiency, since the access to the memory might get congested and blocked creating a bottle-neck affect. One common and well-known computer system that uses shared memory architecture is Dual-Pentium system by Intel.

There are three different types of interconnection networks used with shared-memory: uniform memory access (UMA), non-uniform memory access (NUMA), and cache-only memory architecture (COMA).

2.1 The UMA Model

In the UMA system, a shared memory is accessible by all of the processors through an interconnection network in the same way a single processor accesses its memory. The UMA interconnection network can use different type of network link arrangements, such as a single bus, multiple buses, a crossbar, or a multi-port memory. All of the processors see the memory at the same way; thus, the memory access time is the same for all processors. It should be noted that interconnection network can create an access-time delay, yet this delay will not be caused by the computer architecture.

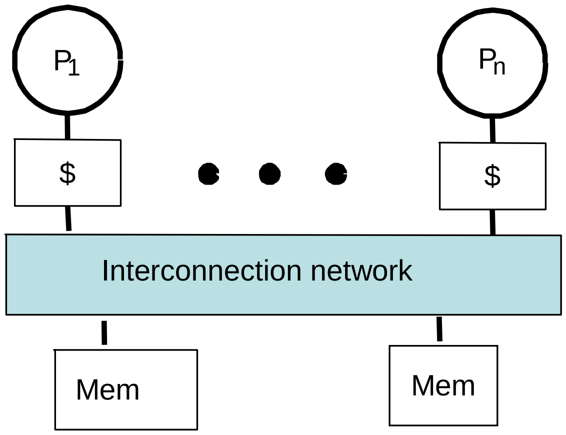


Figure 7 – UMA Model [36]

The processors in UMA model can be described as a synchronous and asynchronous. Synchronous architecture implies that all of the processors are the same and can perform similar functionality. In asynchronous architecture, the processors vary based on the functionality that they can perform. For example, master processor acts as a control processor, while other processors perform operations assigned to them by master processor.

2.2 The NUMA Model

In the NUMA system, memory does not have to be in the same location, therefore each processor may be associated to a memory block. All memory blocks form a global address space which is accessible by all processors. Thus, shared memories are distributed to local memories. There is different access times depending on where the memory block is located comparing to the accessing processor. Therefore, parallel systems might not be advantageous if there is a lot of remote memory access due to increased access time.

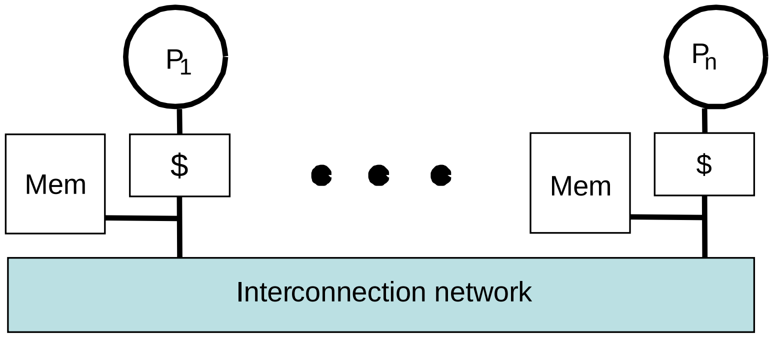


Figure 8 –NUMA Model [36]

2.3 The COMA Model

The COMA model is similar to the NUMA model, as each processor has part of the shared memory. But in this case, the shared memory consists of cache memory. Data must be migrated to the processor requesting it. There is a cache directory that helps in remote cache access.

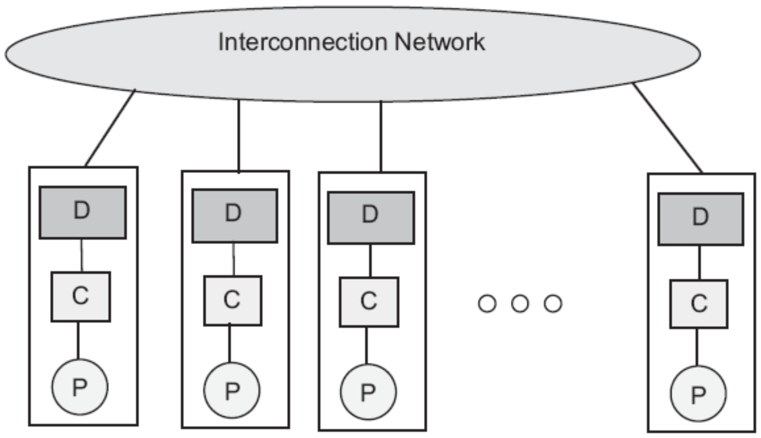


Figure 9–COMA Model [34]

3 Distributed-Memory Multiprocessors

In the distributed memory multiprocessors systems, communication is done through interconnecting networks by sending and receiving messages. There is no use or need of global memory (See Figure 5b). This method, in general, can be compared to sending mail using regular postal service, where you send a message to a specific destination and wait for the response. In this system, a large number of processors can be added since no blocking in accessing shared content will occur. This system is commonly used in network applications and large mainframes. Since there is overhead to be considered in message-passing, messages tend to be larger.

In terms of access time and delays, two things must be considered: link bandwidth and network latency. Link bandwidth is defined as the number of bits that is sent per unit of time (bits/s), where latency is seen as the time it takes to transmit a message through the network from sender to receiver. The size of a process in a message passing through the system can be defined by the process granularity. This can be formulated as follows.

$$\text{Process Granularity} = \frac{\text{computation time}}{\text{communication time}} \quad [37]$$

Although message passing generally uses medium or coarse granularity, there are three types of granularity that are defined as follows.

1. Coarse granularity: Each process holds a large number of sequential instructions and takes a substantial amount of time to execute.
2. Medium granularity: Since the process communication overhead increases as the granularity decreases, medium granularity describes a middle ground where communication overhead is reduced.
3. Fine granularity: Each process contains a few sequential instructions (as few as just one instruction). [4]

It is evident that message passing systems are the only way to efficiently increase the number of processors managed by a multiprocessor architecture and thus has an edge over a shared memory system.

4 Interconnection Networks

The interconnection networks can be classified based on their topology, mode of operation, switching techniques, and control strategy.

4.1 Topology of the network

Networks can be differentiated as static or dynamic.

In static networks, processors and memories are connected in advance and this connection does not change. Figure 10 shows different static topologies.

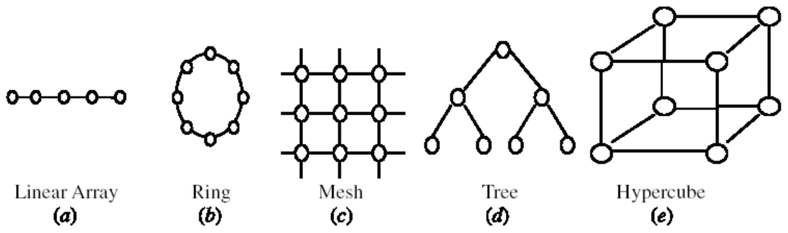


Figure 10 – Examples of static networks [36]

There are few disadvantages to static topology. To increase the performance of the network, current connections have to be removed and new ones installed. In some cases, it is very expensive or not even possible. It is also expensive to add new processors or memory units to the system. There is also a limit on the number of processor/memory units that can be added. In some static topologies, the processors will be too far-removed to provide an efficient parallel computing, as the access time between them will be too large.

Dynamic networks are formed as needed. The examples of dynamic networks include single-stage, multistage and crossbar switch. Figure 11 illustrates a crossbar switch.

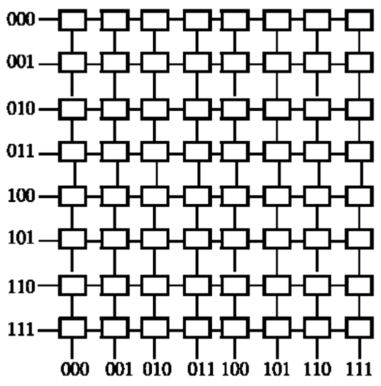


Figure 11 – Crossbar switch [36]

The dynamic networks are used to provide the fastest possible communication path. They can also allow for more than one communication paths to operate simultaneously. Besides being efficient, dynamic networks are also more reliable. New connection can be formed if the original communication path fails. As can be seen from Figure 11, dynamic network infrastructure can be expensive to set up. Multiple binary switches are required to connect each source with every destination. Dynamic networks do not have the same limit on performance as static ones. Although, crossbar switch might not be feasible to set up for multiple processors, multistage connection networks are widely used. For example, information in the Internet travels through dynamic multistage interconnection network.

4.2 Mode of operation

Mode of interconnection network operation can be synchronous and asynchronous. Synchronous networks use global clock signal to synchronize communication. The operation of the network is in lock-step mode. Asynchronous networks use hand-shaking techniques to coordinate communication. Due to the lock-step mode of operation, synchronous networks are slower; yet, they are more reliable i.e. they are race and hazard-free. Distributed systems used asynchronous networks, since it is not possible to have a global clock.

4.3 Control Strategy

Control strategy can be characterized by being centralized and decentralized. In the centralized control strategy, the central control unit oversees the operation of all of the network components. This can greatly affect the performance and reliability of the system. Decentralized control strategy spreads the control of the system among different components.

4.4 Switching Techniques

There are two switching techniques used, circuit and packet switching. Circuit switching established the communication path between source and destination which exists until the end of communication. The message is sent as a whole through this pre-established path. Packet-switching technique divides the message into smaller parts, packets, and sends each packet independently. In packet switching, the provision for packets arriving at different times and out of order has to be made. Packet switching also is more efficient on use of network resources, especially for large messages.

5 Classification Based on the Kind of Parallelism

Figure 12 shows the classification system based on the kind of parallelism. Parallel architectures are based on the information that is being used in parallel. It can be either data or function.

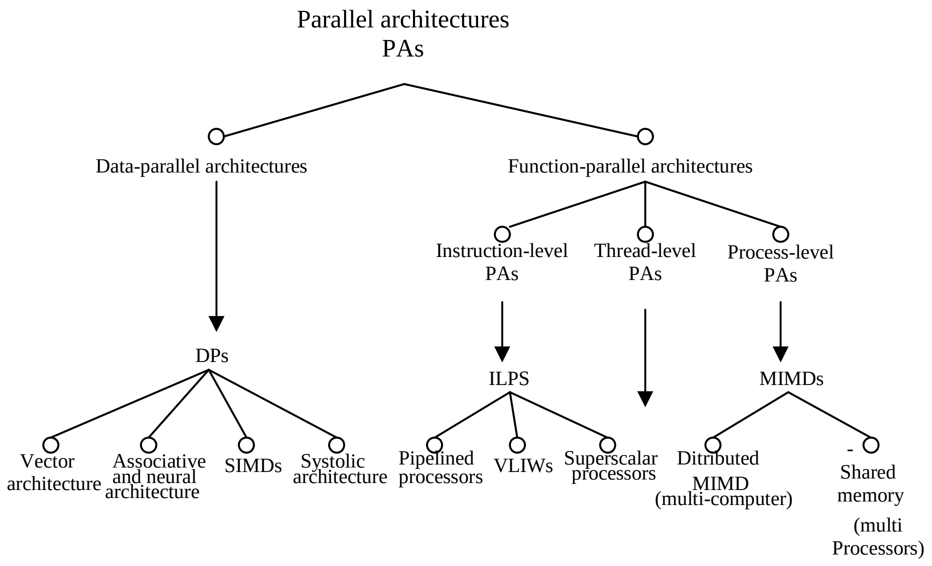


Figure 12 – Parallel Architectures [36]

In data-parallel architecture, as the name implies, data is used in parallel. One of the examples of data-parallel architecture is SIMD from Flynn's taxonomy. Other examples are vector, associative and neural, and systolic architectures. These architectures execute single instructions on multiple sets of data in various ways.

In function-parallel architectures, various functions can be performed in parallel. The functions can be a simple instruction, a thread (combination of few instructions) or a complete process.

Pipelined processors, VLIW (Very Long Instruction Word) and superscalar processors are examples of instruction-level parallel architectures. Pipelined processors execute sub-sets of instructions simultaneously (i.e. performing fetching for one instruction, decoding for another, memory-write for yet another etc.). VLIW codes very long instructions that specify the operation for each sub-unit of the processor [38]. Superscalar processors have multiple processor units on one machine.

Thread-level parallelism is very common in software development (e.g. Java threads). The thread-level hardware parallel architectures are starting to emerge.

Process-level parallel architectures are essentially MIMD architectures according to Flynn's taxonomy. As mentioned earlier, MIMDs can be classified as distributed memory MIMDs (i.e. message-passing) and shared-memory MIMDs.

Summary

This lecture has introduced different classification systems for parallel computer architecture. The most commonly used is Flynn's classification system that is based on instruction and data streams. Although, being a very powerful system, Flynn's taxonomy does not take interconnection networks, I/O or memory in consideration.
When memory is considered, multiprocessors can be divided into shared-memory and message-passing systems. In turns shared-memory multiprocessors can be classified as Uniform Memory Access (UMA), Non-Uniform Memory Access (NUMA), and Cache-Only Memory Architecture (COMA) systems.

Interconnection networks can be classified based on their mode of operation, control strategy, switching techniques or topology.

All of the above classifications do not provide the complete description of the system. It is common to describe the system using few of the mentioned classifications. For example, parallel computing system can be described as shared-memory MIMD. The classification used to describe the system depends on the required level of abstraction and application.

Paper 1 A Survey of Parallel Computer Architectures

Paper 2 Performance of Multiprocessor Interconnection Networks

Lecture #6a: Dynamic Interconnection Networks - Buses

Course: Computer Architecture III
Lecturer: Miodrag Bolic
Scribe(s): Anuj Shah and Saurabh RattiDate: October 15, 2007Introduction

Introduction

Interconnection Networks (INs) are an important concept in all aspects and levels of designing computer systems. An IN is the architecture that connects components in a computer system, determining the manner in which communications are carried out. In multiprocessor systems, the IN connects the processors and memories, allowing them to communicate with each other.

INs can be classified into two major groups: static and dynamic. In static INs, “direct fixed links are established among nodes to form a fixed network, while in dynamic networks, connections are established as needed” [39]. Dynamic networks can be further divided into bus-based and switch-based systems. Bus-based systems are the most common IN in computer systems, and are the focus of this paper [40].

Explanation

1 Single and Multiple Buses

The defining concept of a bus interconnection network is that components are connected, and communicate, via a shared communications line. Figure 1 shows an example of a simple single bus structure that allows communication between generic peripherals.

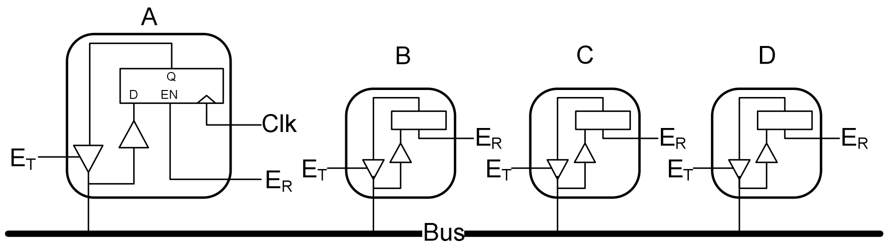


Figure 1: Single bus with 4 generic peripherals [40]

The communication link between the peripherals is the bus, which may be comprised of several wires (bit-lines). An internal register is used to store data read off the bus when the Enable Receive (E_R) signal is set. The signal from the bus to the register is sent through a buffer for signal conditioning. The register’s output is transmitted to the bus when the Enable Transmit (E_T) signal is set. If a peripheral’s E_T signal is not set, the tri-state buffer prevents the register’s output from reaching the bus. This is necessary since the bus is a shared resource, and only one peripheral can transmit at a given time. The E_R and E_T signals are controlled by an arbiter, which is discussed in Section 3.

The bus-based IN structure can also be comprised of multiple buses. When using multiple buses, many processor and memory connection schemes are possible, two of which are illustrated in Figure 2.

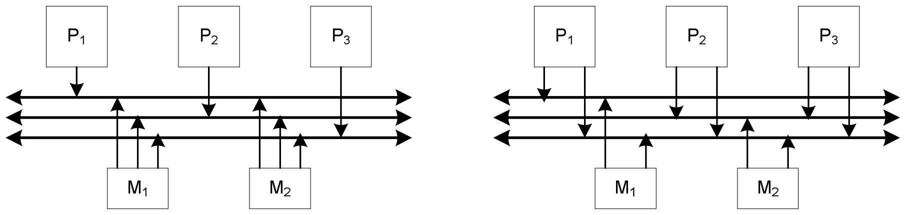


Figure 2: Two different multiple bus connection schemes

When considering whether to use single or multiple based buses, it is important to note the advantages and disadvantages associated with each. In single bus INs the network complexity remains the same with the addition of peripherals, as they are simply connected to the single bus. However, since access requests for the shared bus also increase, contention and blocking become issues to consider. Multiple buses can address this problem, as the connection scheme can be optimized for the system behavior. In addition, there is the possibility of constructing redundant buses for fault tolerance. However, with these advantages come the caveats of requiring more resources to implement them and increased complexity.

2 Bus Communication Styles

Communication over a bus is composed of several elements:

Cycles:	The smallest unit of time that is measured on the bus.
Message:	A unit of information sent from one peripheral to another, often sent within a cycle. However, a message may require a number of cycles in order to be sent from sender to receiver over the bus.
Transaction:	A transaction is an exchange of information between two peripherals. A transaction consists of a sequence of messages.

For example, consider a processor reading a location from memory. The read transaction consists of two messages. The first message, from the processor to the memory, contains a request to read and an address. The second message, from the memory to the processor, contains the data from the memory location. All these messages can be sent within one cycle if the bus width is the same size as the address and memory data. However if the bus only has 8 lines for communication, but 32 bits of data must be transferred, a total of 4 cycles are required to complete the data message.

2.1 Synchronous

If all peripherals on the bus are clocked at the same speed, it is possible to use the same strict timing protocol for intercommunication. This means that all parts of the transaction are synchronized and executed according to distinct clock edges.

Synchronous timing schemes are very simple to control and reduce the overhead associated with ensuring that the correct data is available when needed. However, this scheme is best suited when designing systems composed of peripherals with similar operational speeds. An example of this is Altera’s Avalon bus architecture. The Avalon bus, which is synchronous, is used in the NIOS II processors for Altera’s FPGAs. All the components within the NIOS II processor, such as CPUs and memory, run off a shared clock supplied to the processor [41]. This allows each component that knows the bus timing scheme to communicate over the bus without timing issues arising. The Avalon bus is further discussed in Section 5.

Figure 3 illustrates the sequence of events during a synchronous timing scheme.

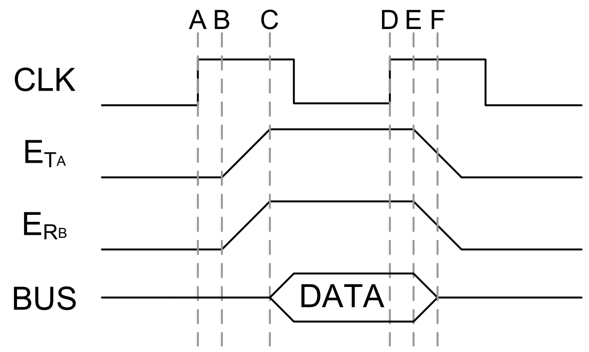


Figure 3: Synchronous timing sequence [40][40]

A:	Arbiter logic begins to compute which master and slave will be communicating.
B:	Arbiter logic begins to assert the E_{TA} and E_{RB}.
C:	E_{TA} is set and data begins to flood the bus.
D:	Register in B reads the data off the bus, and arbiter logic computes the next cycle.
E:	Arbiter logic begins to de-assert the E_{TA} and E_{RB}
F:	E_{TA} and E_{RB} signals become low; A stops transmitting to bus and B stops reading.

2.2 Asynchronous

Asynchronous timing schemes rely upon handshaking protocols in order to complete a transaction. This method offers the ability to communicate without a common clock. Instead, it relies upon the assertion of signals between entities to communicate when portions of the transaction have been completed. Again, take Altera’s Avalon bus as an example. While it is synchronous bus when internal to a NIOS II processor, it provisions for the possibility that the NIOS II processor may be connected to slower peripherals that do not run off the shared NIOS II clock or use the same timing scheme, such as off-chip memory devices. Once an asynchronous handshaking protocol is defined between processor and peripheral, communication can be achieved [41].

To illustrate this, take a single master and single slave with different clock speeds on a single bus. In order to communicate without a common clock, a handshaking protocol must be used to accomplish a read transaction. Figure 4 illustrates the sequence of events during the transaction.

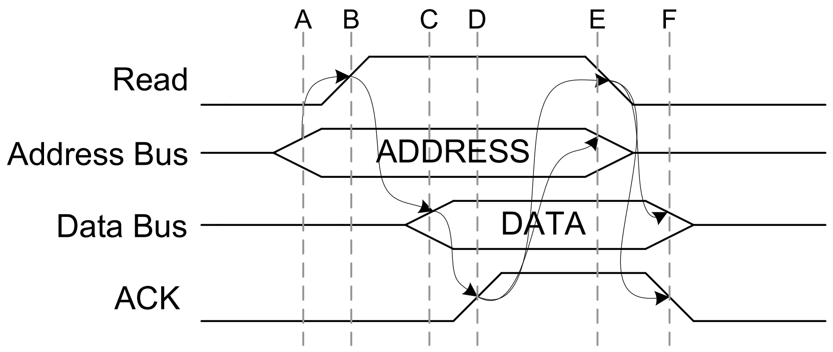


Figure 4: Asynchronous timing sequence

A:	Master puts the read address onto the address bus.
B:	Once address value on bus stabilizes, read signal is put high by master.
C:	Slave retrieves data from the address specified, and places it on the data bus.
D:	Once data value on bus stabilizes, ACK signal is put high by slave.
E:	Master reads data bus, de-asserts the read signal and address bus.
F:	Once read signal is low, slave de-asserts ACK signal and data bus.

3 Bus Arbitration

As more and more peripherals are added and the size of the system grows, more devices will be requesting access to the shared bus. In order to eliminate the possibility of multiple peripherals accessing the bus simultaneously, it is the task of the arbitration unit to manage and grant access.

However, for the arbitration scheme to be truly efficient, two criteria must be taken into account: the first being that high priority peripherals must gain access often; the second being that low priority tasks should not be indefinitely blocked from accessing the resource. In dealing with these constraints, arbitration schemes have been developed, a few of which are discussed here.

Daisy Chain Arbitration

In daisy chain arbitration, priority of a peripheral is based upon its proximity to the arbitration unit.

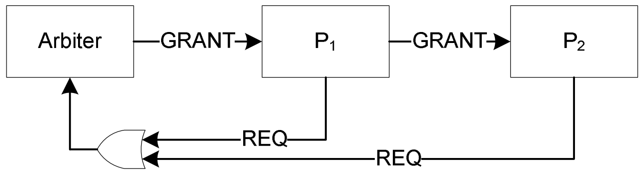


Figure 5: Daisy chain arbitration

In Figure 5, if P_2 requests bus access from the arbiter, the arbiter will grant access to P_1. Since P1 did not request access, it will pass the grant onto P_2. If both processors P_1 and P_2 request access from the arbiter simultaneously, the arbiter will grant access to P_1. If P1 continually requests access, P2 will never receive the grant. This method is not used nowadays, since some peripherals may never be granted access.

Arbitration with Independent Request and Grant

Here the arbitration is handled in a module separate from the peripherals. The decision as to who is granted access can be based upon a number of techniques.

Round-Robin	The most recent request to be granted will have the lowest priority in the next round of arbitration[40].
TDMA	Each peripheral is given a fixed time slot in which to execute[40]. This method is very simple to implement but allocates time slots to peripherals that may not require access.
Unequal-Priority	Each processor is assigned a unique priority, and access is assigned to the requesting processor with the highest priority [40].

Hybrids of the above 3 methods are also possible, such as with the Avalon bus in Section 5.

3.3 Distributed Arbitration

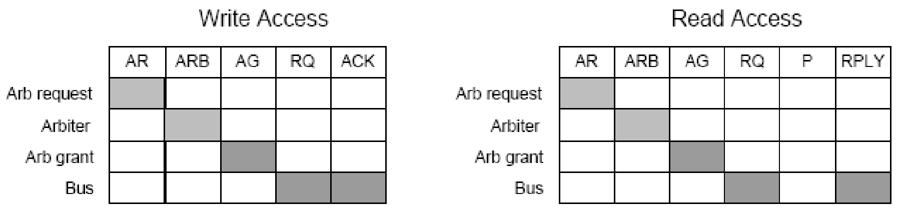
With distributed arbitration no one specific arbitration unit exists. Instead, arbitration is handled individually by each peripheral as access is required. If two peripherals require access to the data transfer bus at the same time, there are several techniques for delegating the resource, the first being arbitration by self-selection. In the self-selection arbitration scheme the devices requiring the resource assert their identities to all the other devices. Each device then examines these identities and determines independently whether they, or another device, have priority access to the bus. The second distributed arbitration scheme is collision detection. In this scheme, any device requiring the bus monitors it until it is free, and then transmits onto the bus. If the device detects that the data on the bus is different from what it is transmitting, meaning another device is also transmitting, both devices stop using the bus and wait a random amount of time until they try again. This collision handling scheme is used in other networks as well, such as Ethernet, and works well in practice. [42] The advantage of distributed arbitration is that there is no central arbiter to be redesigned to accommodate the addition or removal of peripherals from the design. However, in distributing the arbitration, there is also duplication of the arbiter "module" in every peripheral, possibly increasing the overall hardware requirements.

4 Advanced Bus Techniques

4.1 Bus Pipelining

Since all the cycles in a given transaction may not be utilizing the bus, it is possible to pipeline the execution of multiple transactions on a synchronous bus, similar to pipelining the execution of multiple instructions in a processor. This method improves efficiency since the bus resource is used where it would not be otherwise, thereby increasing the utilization of the bus.

Figure 6 shows message sequences for write and read transactions, with Table 1 outlining the details of the different messages of the transactions.



Write Access

	AR	ARB	AG	RQ	ACK
Arb request	▨				
Arbiter		▨			
Arb grant			▨		
Bus				▨	▨

Read Access

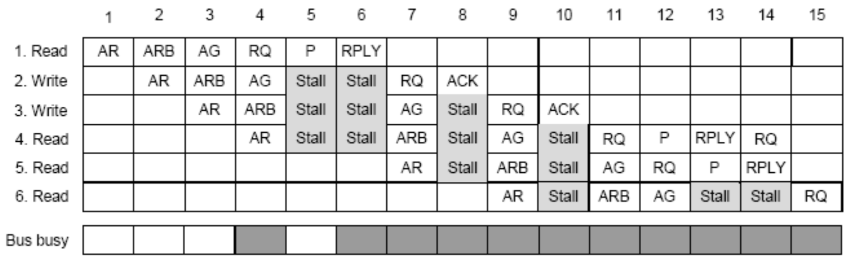
	AR	ARB	AG	RQ	P	RPLY
Arb request	▨					
Arbiter		▨				
Arb grant			▨			
Bus				▨		▨

Figure 6: Message sequences for write and read operations [43]

cles not using bus		Cycles using bus	
AR	Arbitration request,	RQ	Request signal is set
ARB	Arbiter processing cycle	RPLY	Reply from the memory or I/O
AG	Grant signal is set	ACK	Ack. of write from memory or I/O
P	Pause while data is retrieved		

Table 1: Transaction message details

Now with these transactions, take an example transaction sequence of Read, Write, Write, Read, Read, Read, as shown in Figure 7.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1. Read	AR	ARB	AG	RQ	P	RPLY									
2. Write		AR	ARB	AG	Stall	Stall	RQ	ACK							
3. Write			AR	ARB	Stall	Stall	AG	Stall	RQ	ACK					
4. Read				AR	Stall	Stall	ARB	Stall	AG	Stall	RQ	P	RPLY	RQ	
5. Read							AR	Stall	ARB	Stall	AG	RQ	P	RPLY	
6. Read									AR	Stall	ARB	AG	Stall	Stall	RQ
Bus busy				▨		▨	▨	▨	▨	▨	▨	▨	▨	▨	▨

Figure 7: Pipelined bus usage [41]

When pipelining transactions on a bus, it is necessary to insert stall cycles into a transaction while another is being completed, in order to avoid overlapping bus usage by multiple transactions. In order to not have the ACK of transaction 2 (write) overlap with the RPLY of transaction 1 (read), transaction 2 is stalled at cycle 5 and 6.

4.2 Split Transactions

The purpose of split transactions is to handle the event of a peripheral taking an extended period of time to reply to complete a transaction. In the previous example, pausing for a peripheral to reply (ACK or RPLY) requires the bus to be unused. Rather than wasting this pause time, the transaction can be split into request and reply transactions, thus allowing for the bus resource to be used by other peripherals in that interim. Figure 8 shows the advantage of split transactions, with three operations of various wait times.

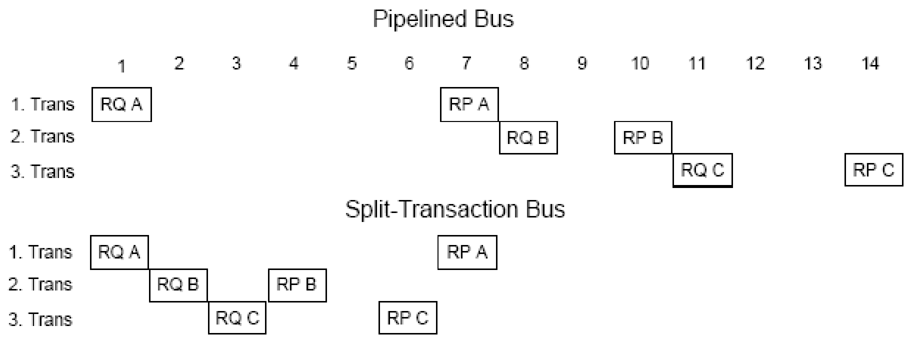


Figure 8: Pipelined bus usage versus split-transaction bus usage [43]

All three transactions (A, B and C) consist of a request (RQ) message, a wait time and a reply (RP) message. The wait times for transactions A, B and C are 5 cycles, 1 cycle and 2 cycles, respectively. In the pipelined bus, the transactions are atomic and cannot be interrupted by other transactions, forcing them to queue up and take 14 cycles to complete. In the split-transaction bus, the RQ and RP messages are transactions in themselves. This means that the wait time between related RQs and RPs can be used for the other transactions, reducing the overall execution time to 7 cycles.

4.3 Burst Messages

Bus messages can be viewed as having two main elements: control overhead and data. Since the main purpose of a transaction is often the transfer of data, the efficiency of a transaction is the ratio between the amount of overhead needed to transfer the data versus the amount of data transferred. Burst messages attempt to increase efficiency by including higher amounts of data per message, reducing the overhead to data ratio in the message, as shown by Figure 9.

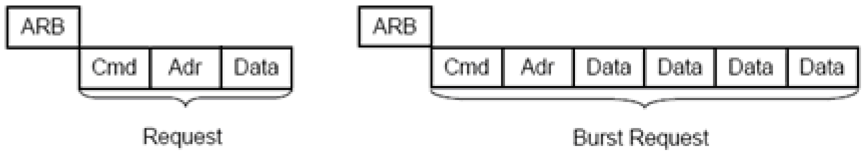


Figure 9: Regular message versus Burst message [43]

In the regular message, only a fourth of the message contains data. In the burst message, this ratio is increased to one half, making the message more efficient than the other. However, the disadvantage of this is that other devices queuing up to use the bus must wait longer as the length of messages has increased. [43]

5 Avalon Bus

The Avalon bus is a proprietary bus specification developed by Altera [41] for use with their NIOS family of embedded processors [44]. The goals of the bus scheme are to provide address decoding, data-path multiplexing, wait-state insertion, and arbitration for multi-master systems [40].

The specification interconnect strategy allows designers to connect master and slave peripherals together. Rather than creating redundant signals and hardware, only the required signals are generated to support the designer's need[41]. The behavior of these signals and the types of transfers supported by these signals are also defined. The Avalon bus also provides a mechanism for decoding address logic, meaning only a single address must be provided to the Avalon bus to accomplish inter-module communication. Avalon internally decodes the address to select the relevant slave, as well as handle requirements for different address widths.

5.1 Bus Timing

The Avalon bus operates in synchronous mode with the clock supplied to it. Any signal placed on the bus is read at the rising edge of the clock, and thus all signals to be read must be stable on the rising clock edge. As long as this remains true, signals may be changed at any point between clock edges without fear of disrupting communications between any modules using the bus synchronously. [41]

The synchronous property of the bus does not preclude asynchronous communication between peripherals using the bus, such as off-chip memory devices [41]. However, it does require the designer to be aware of the specifications of the Avalon bus in order to overcome any timing issues.

5.2 Arbitration Placement

Another important aspect of the Avalon specification is that arbitration is handled slave side of the bus rather than the master [40]. This technique removes unnecessary arbitration in portions of the hardware that does not require it, and only connects what is needed. For example, compare the architectures of the traditional multi-master system in Figure 10 with that of the simultaneous multi-master bus used by Avalon in Figure 11.

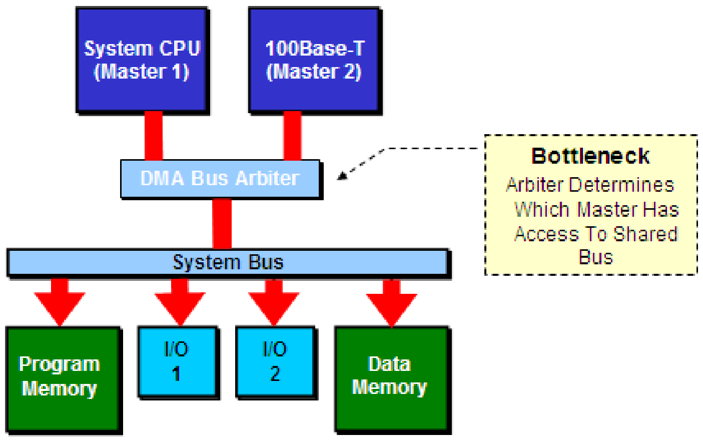


Figure 10: Traditional multi-master bus [40]

With the traditional multi-master bus, the arbitration unit must determine whether Master 1 or Mater 2 can access the System Bus. If Master 1's task requires it to communicate with all the slave devices, and Master 2 only requires communication with the Data Memory slave device, the arbiter can only grant access to one of the masters at time, even in the event that both masters require access to different modules. Therefore, the communication bottleneck occurs at the arbiter.

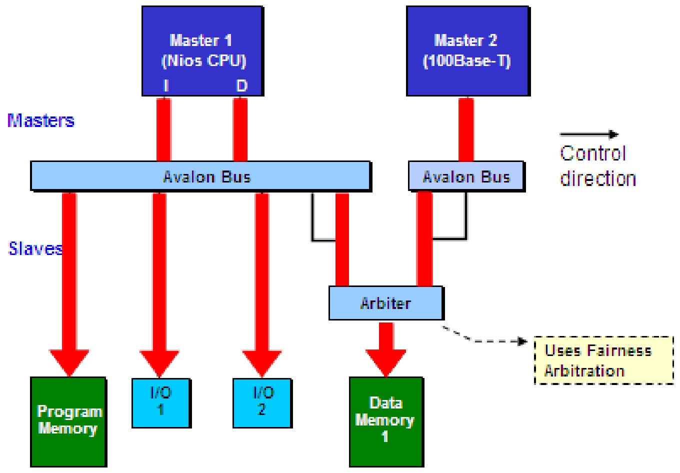


Figure 11: Simultaneous multi-master bus [40]

However, the simultaneous multi-master bus uses two Avalon blocks to make the connections from master to slave and the arbiter is placed on the slave-device end. Therefore, Master 1 can connect to the Program Memory without Master 2 aware of the connection. At the same time, Master 2 can connect to the Data Memory. If both Master 1 and Master 2 request access to the Data Memory at the same time, the arbiter would then determine who is to be granted access. Thus communication is only constrained when two simultaneous requests of one module occur.

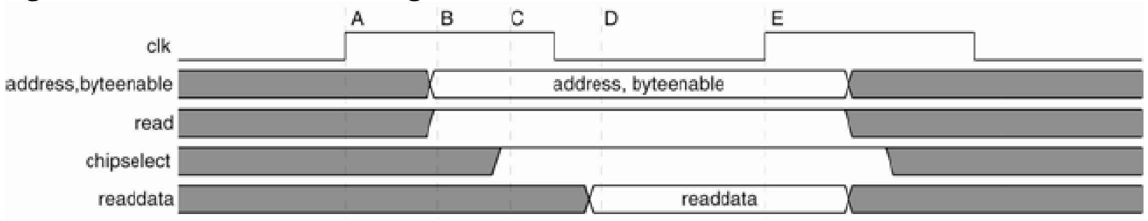
This multi-master bus also utilizes a fairness arbitration scheme, a hybrid of round-robin and unequal priority. This method involves assigning integer "shares" to each master slave pair. If a conflict is encountered on the bus, the master with the highest number of shares is granted access until these shares are used up. The master with fewer shares then takes the bus until its shares are used up. This mechanism alleviates the problem of all masters continuously requesting access to the bus, since each will be given access for a percentage of time equal to the percentage of total master shares that they own. [40]

5.3 Transfer Specifications

The transfer specifications of the Avalon bus is split into the perspective of the masters and slaves connected to the Avalon switch fabric.

5.3.1 Slave Transfer

Figure 12 shows the basic timing of a slave read transfer.



(A) First cycle starts on the rising edge of `clk`.
(B) `address` and `read` from Avalon switch fabric to slave port are valid
(C) Avalon switch fabric decodes `address` and asserts valid `chipselect`.
(D) Slave port returns valid data on `readdata` during the first cycle.
(E) Avalon switch fabric captures `readdata` on the next rising edge of `clk`, and the read transfer ends.
The next cycle begins here, and could be the start of another transfer.

Figure 12: Slave read transfer [41]

The slave read transfer in Figure 12 shows a read transfer that takes a single cycle. If a longer response time be necessary, the slave asserts a signal named waitrequest, and the time between (C) and (D) will extend over the necessary amount of cycles to complete the read with wait-states inserted onto the bus while wait request is high. This is illustrated in Figure 13. [41]

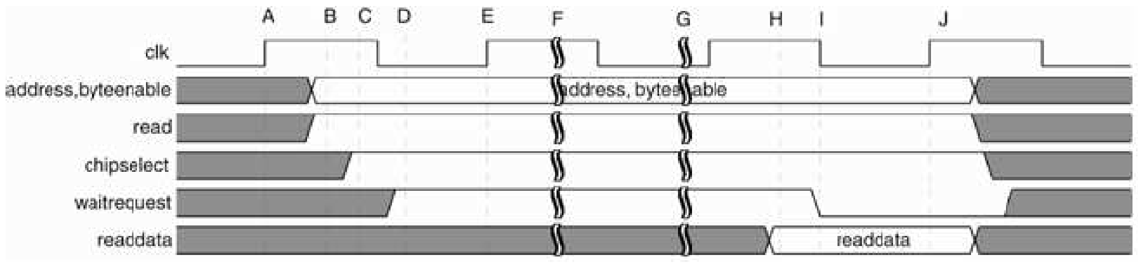
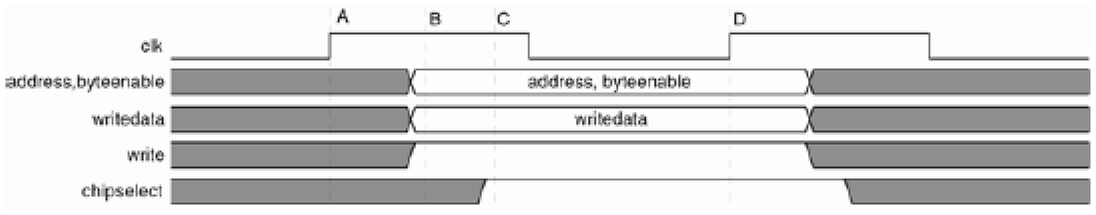


Figure 13: Slave read transfer with wait states [41]

Figure 14 shows the basic timing of a slave write transfer.



(A) First cycle starts on the rising edge of `clk`.
(B) The Avalon switch fabric asserts valid `writedata`, `address`, `byteenable` and `write` signals.
(C) Avalon switch fabric decodes `address` and asserts valid `chipselect` to slave.
(D) Slave port captures `writedata`, `address`, `write`, `byteenable` and `chipselect` on the rising edge of `clk`, and the transfer terminates. The next cycle begins here, and could be the start of another transfer.

Figure 14: Slave write transfer [41]

The slave write transfer in Figure 14 shows a write transfer that takes a single cycle. If a longer response time be necessary, the slave asserts another signal named waitrequest, and the Avalon switch fabric inserts a wait-state onto the bus at every clock cycle edge that it is high, (F) and (G) in Figure 15. Once the slave can complete the write, it deasserts waitrequest and the transfer completes at (I). [41]

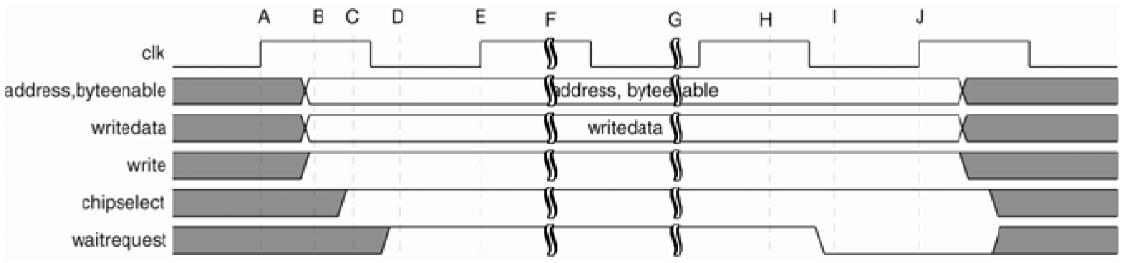
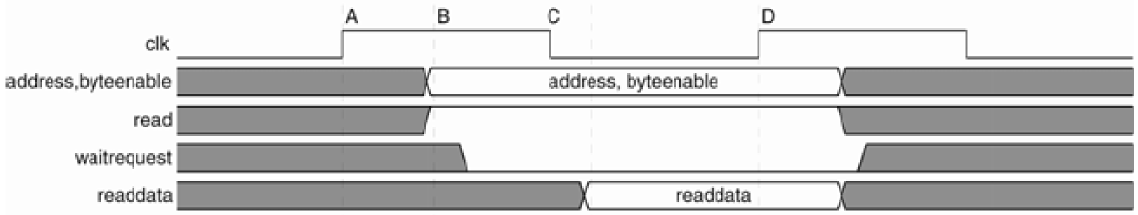


Figure 15: Slave write transfer with wait-states [41]

5.3.2 Master Transfer

Figure 16 shows the master read transfer timing specification.

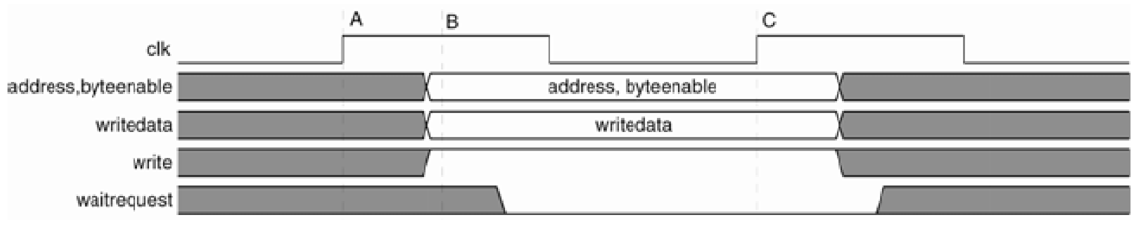


(A) First cycle starts on the rising edge of `clk`.
(B) Master port asserts valid `address`, `byteenable` and `read`.
(C) Valid `readdata` returns from the Avalon switch fabric during first cycle.
(D) Master port captures `readdata` on the next rising edge of `clk` and deasserts all its outputs. The read transfer ends here and the next cycle could be the start of another transfer.

Figure 16: Master read transfer [41]

If the slave device asserts the waitrequest signal, the master continues to hold the read, address and byteenable values on the bus for the slave device. Once waitrequest is deasserted, the master then reads the readdata value off the bus and deasserts its outputs at the next clock cycle. [41]

Figure 17 shows the master write transfer timing specification.



(A) Write transfer starts on the rising edge of `clk`.
(B) Master asserts valid `address`, `byteenable`, `writedata`, and `write`.
(C) `waitrequest` is not asserted at the rising edge of `clk`, so write transfer terminates. Another transfer could follow on the next cycle.

Figure 17: Master write transfer [41]

If the slave device asserts the waitrequest signal, the master continues to hold the writedata, address and byteenable values on the bus for the slave device. Once waitrequest is deasserted, it indicates the write is completed and the master deasserts its outputs at the next clock cycle. [41]

Summary

5.3.3 Pipelined Transfer

The pipelining of the slave and master read transfer timings in Figures 12, 13 and 16 can be accomplished over the Avalon bus. This increases the bandwidth of reads between synchronous peripherals as new transfers can be started before data is returned by the slave for the previous request. The throughput of the bus is limited by the amount of time the read and address values must be held on the bus for the slave peripheral to read. Once again, the use of the waitrequest signal from the slave allows the Avalon bus to handle variable response times. [41]

5.3.4 Burst Transfer

The Avalon bus also supports burst transfers between a master and a slave. A burst is the execution of multiple transfers as a unit, which is advantageous for slave peripherals that are designed for high efficiency when handling multiple data units. Burst transfers are initiated by a master, which locks the slave to communicate it, without interruptions, for the duration of the burst. [41]

Paper 1 Design and Analysis of Arbitration Protocols

Paper 2 Architecture of a System on a Chip with User-Configurable System Logic

Lecture #6b: Dynamic Interconnection Networks: Buses

Course: Computer Architecture III
Lecturer: Miodrag Bolic
Scribe: Dominic Lalonde, 3258780 and Marc-André Schiffo, 3403515
Date: September 27 and September 29, 2006

Introduction

Today, computer systems are using multiprocessor architectures. In order to keep a good performance, the interconnection network between the processors and the memory must be designed carefully. This is the case because they are usually the limiting factor in systems.

Multiprocessor systems consist of multiple processors and sometimes, multiple memory units, that are connected together through an interconnection network. Those systems can use multiple communication styles. The first style is seen when processors use a shared memory space to communicate between each other. To communicate, a processor deposits it's information in a designated memory space, and then, this same memory space is accessible and read by the other processor in order to complete the information sharing. The second style is named message passing, because to communicate, one processor sends the information directly to the destination processor. Those communication styles both require interconnection networks, and those networks have a high influence on the communication speed.

In general, interconnection networks are said to be static or dynamic. Static networks include 1D(e.g. linear), 2D(e.g. mesh) and HyperCube(e.g. Super Computers). Dynamic networks can be seen as Switch based or Bus based networks. Switch base include single stage, multiple stage and Crossbar. The bus based networks, which are the main topic of this document, can be divided into Single bus networks and Multiple bus networks. The following sections will include single and multiple bus interconnection networks, addressing and timing, arbitration, advanced bus allocation techniques and Avalon bus.

Explanation

Single and Multiple Bus Interconnection Networks

If someone would want to build a system that requires between two and fifty [47] processors, he/she could use a single bus interconnection network for his task; the performance of the bus would depend on the traffic (the traffic is the amount of bus utilization) of each processor and the bus bandwidth (the bus bandwidth is the maximum rate at which it can transfer data). This would surely be the simplest way to implement the interconnection network, but it would not be the best performing design. Having a lot of traffic on a single bus can cause a lot of delays, and that is really harmful for the performance. The delays would occur because a bus has a limited bandwidth and also, a bus can be used by only one processor at a time, meaning that only one value can be present at a given time. In order to know what processor has access to the bus, an arbitrator module is used, which will be described in more details in the following sections. The arbitrator's task becomes more complex and time consuming when there are a lot of processors that want to communicate at the same time and it has to decide what processor has the priority to the bus.

To overcome the single bus problems, the use of multiple bus architectures is studied. Having multiple buses for communicating between processors and memory units only consists of having multiple single buses in parallel. When using multiple buses interconnection networks, we can use different designs for the memory access, and they are well illustrated in Figure1.

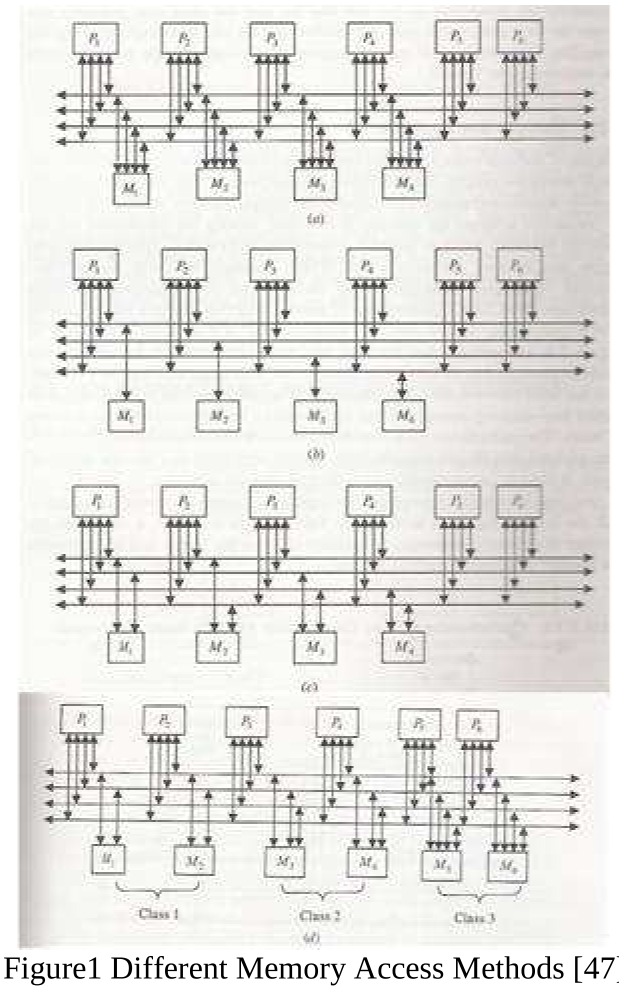


Figure1 Different Memory Access Methods [47]

In Figure1(a), all memory units are accessible by all buses. It is called multiple bus with full bus-memory connection (MBFBMC). In Figure1(b), the memory units are only accessible by a specific bus. This is called multiple bus with single bus-memory connection (MBSBMC). In Figure1(c), memory units are accessible by a subset of buses. It is called multiple bus with partial bus-memory connection (MBPBMC). In Figure1(d), we have memory classes specified in the system, and we can conclude that the “Class 3” memory units are the most used in this system. This is the multiple bus with class-based memory connection (MBCBMC).

Having multiple buses is good to have a better performance. Also, if one bus fails, at least, the communication between processors can still be done. In order to prevent bus contention, we would need more available buses than there are processors or memory units in the system. On multiple buses architectures, the arbitrator’s decisions are easier to make in most of the cases because it has more available buses.

Addressing and Timing

To communicate with another processor, a processor has to know the corresponding address. Data transmission can then be done. Sometimes, a processor needs to send the same data to multiple processors or memory units. For example, in order to prevent 50 repetitive transmissions (if there are 50 other components), a processor could put his data on the bus and the other processors/memory units could accept this data at the same time. This method is called “Broadcasting”. Figure2 is a good example of simple bus communication where broadcasting can be achieved.

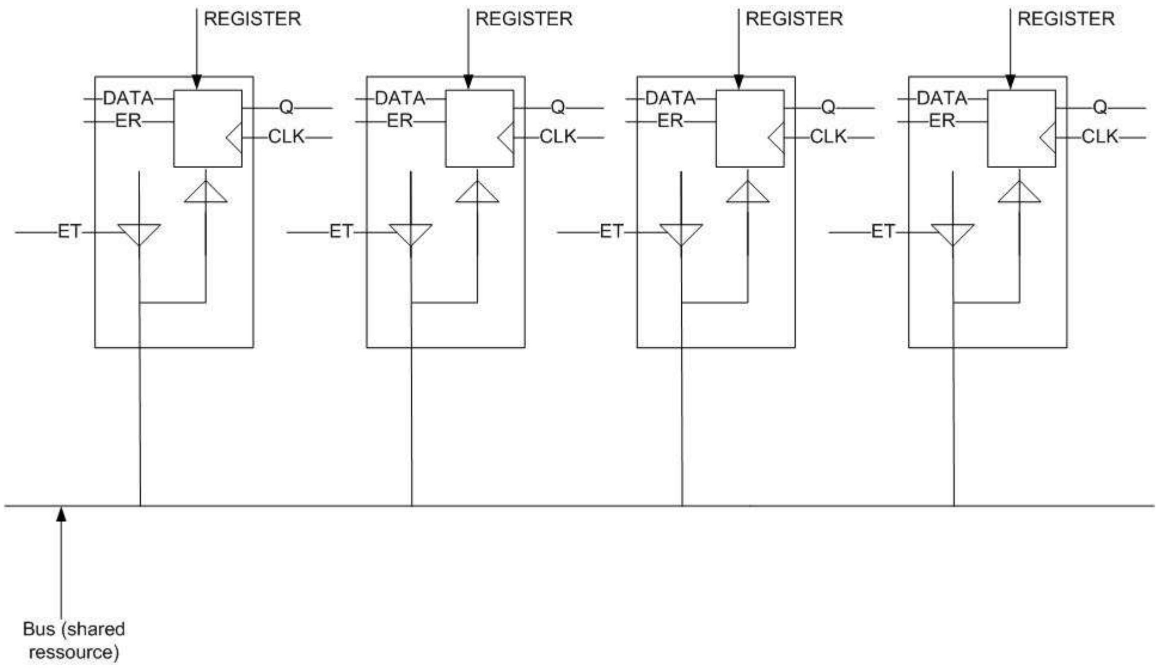


Figure2 Single Bus Communication

In Figure2, one bus is shared by four independent modules. Only one of the modules can put data on the bus at a given time, by setting its ET (Enable Transmit) bit. Then, once the data is available on the bus, the three remaining modules can set their ER (Enable Reception) bit if they want to read the bus’ data. In order to receive data from the bus, the modules have to be equipped with registers that are controlled by the clock and the ER bit. Here, the modules either represent processors or any other peripherals.

Bus timing can either be synchronous or asynchronous. The synchronous timing is simpler to control and is better used when the devices almost have the same speed. The simpler control comes from the fact that the execution time of synchronous systems is computable, so the transaction time is known. The asynchronous timing method is based on the handshaking principle and is better used when a system's components are not all of equal speed. It gives the opportunity to faster components to execute other tasks instead of waiting for slower components.

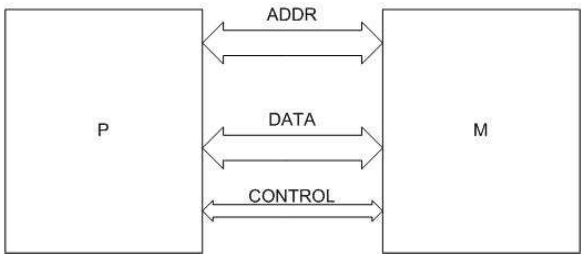


Figure3 Transaction Cycle of a message

If the data bus is the same size at the processor P and at the memory unit M, then the message transaction only takes one clock cycle if the message holds data which is not bigger than the size of the processor's data bus. This is the case in Figure3. Sometimes, the size of the output of P could be different than the size of the input at M. In this case, the transaction is broken into multiple smaller messages (in order to accommodate the input size at M) and therefore, it takes more clock cycles.

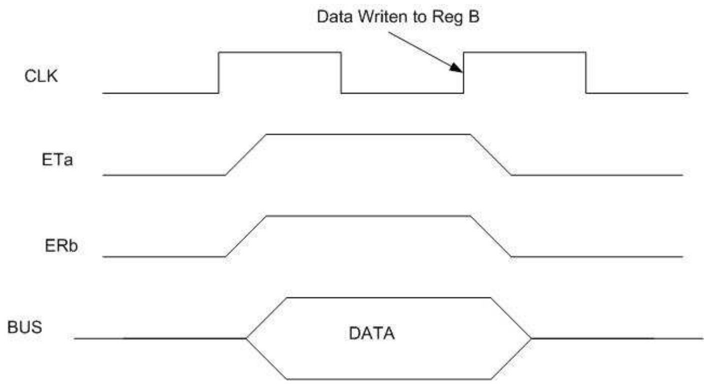


Figure4 Synchronous Transaction

Synchronous transactions can be illustrated by Figure4. We can see that at the first rising edge of the clock, data is sent on the bus and the control bits are set appropriately at the sender and receiver. When the second rising edge occurs, the data is then transferred in the receiver's memory.

Figure5 demonstrates an asynchronous transaction. The arrows present in the drawing represent handshaking. In this example, we have A who will send data to B. As the first step, the ET bit of A needs to be enabled. When A's ET bit is set, A sends the data on the bus and creates a request that is sent to B. When B receives that request, it takes the data from the bus and sends an acknowledgement to A. Then the request message is terminated and A tells the bus that the data is not needed anymore.

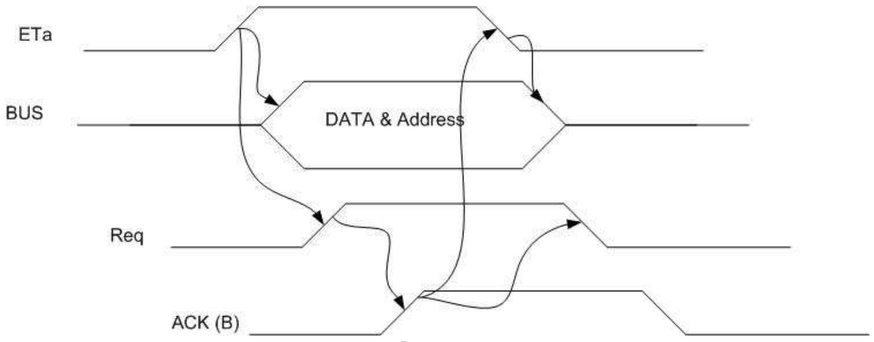


Figure5 Asynchronous Transaction

Most of the buses today use a synchronous timing. Asynchronous timing in buses was quite popular a while ago.

Arbitration

The arbitration module is a hardware and/or software block which decides who can use the bus. Logically, this module is only needed when there are several masters in the system. In a multiple master environment, when a master wants to make use of the bus, it has to send a request to the arbitrator. Then, the master has to wait the grant of the request sent by the arbitrator before it can use the bus. When the master is finished using the bus, it needs to signal it to the arbitrator.

The arbitrators use different techniques in their decision making but they have to try to balance priority and fairness. Some devices in a system are more important than others and therefore, the arbitrator needs to take that factor into account. Also, the arbitrator should take into account that lower priority devices should not be forbidden to use the bus. Doing so, it should prevent higher priority components to use the bus forever even though they need it.

Arbitration can be implemented using different architectures. The first one studied is the “Daisy Chain Arbitration” which is not really used nowadays but gives a good idea

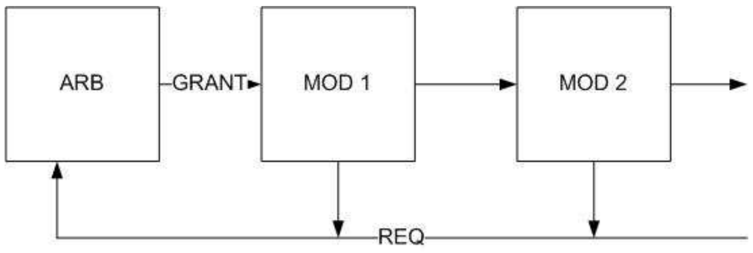


Figure6 Daisy Chain Arbitration example

of the arbitration principle. All the grants sent by the arbitrator passes through the closest module. If this module needs the bus, it will intercept the grant that was sent to another module. Because it does not provide fairness, this architecture is not used anymore. Also, if two requests are sent at the same time, the arbitrator will only see one. Another type of arbitration implementation is “Independent request and grant”. In that architecture, each master has a direct connection with the arbitrator to send its request messages and to receive the grant from the arbitrator. In that way, the arbitrator can have the request of all the masters at the same time, compute his decision and send the grant to whoever he wants. This arbitration type is more flexible than the “Daisy Chain” and is said to have a faster arbitration time. The drawback is that it needs a lot of arbitration connections. Another type of arbitration is “Distributed Arbitration”. In this architecture, there is no arbitrator. Every unit has a corresponding priority and the decision is only based on that priority. A huge drawback of this method is that fairness is not achieved. The highest priority unit could use the bus forever, and the bus utilization may not be as fast if there is more than one processor who needs the bus at the same time because a decision has to be made before a processor can use the bus..

To make the arbitrators intelligent, bus allocation techniques are needed. An arbitrator could use a random algorithm, which would grant bus access to a random unit for a random time, but this is not recommended. It could use the “Round-Robin” technique, where the arbitrator grants access to the bus to each unit, one after the other. If a unit does not need the bus, it skips it and goes to the next. The round-robin technique is harder to implement because it has to know if the master needs the bus. The TDMA (Time Division Multiple Access) technique gives a fixed time to each master whether or not they need the bus. This could waist bus utilization time when a certain master does not need the bus. The arbitrator could also use the LRU (Least Recently Used) technique, which permits the master that did not use the bus for the longest to use it if he needs it. The last technique studied is “Unequal-priority protocol”, where each unit has it’s priority, and then, the arbitrator decides how much time it can be allocated depending on other priorities requests.

Advanced Bus Allocation Techniques

The advanced techniques to share bus utilization include Bus Pipelining, Split Transactions and Burst Messages. The resources are not used most of the time, and pipelining can help to improve the performance. Also, if you pipeline every request for the bus, you can achieve higher utilization of the bus (resources) and diminish the overall communication time.

The split transaction technique separates the transactions in two parts: the request-transaction and the reply-transaction. Figure7 shows that we can pipeline multiple requests while waiting for replies. This technique is good when there are problems with very slow peripherals because masters don’t have to wait for the reply of a slower unit.

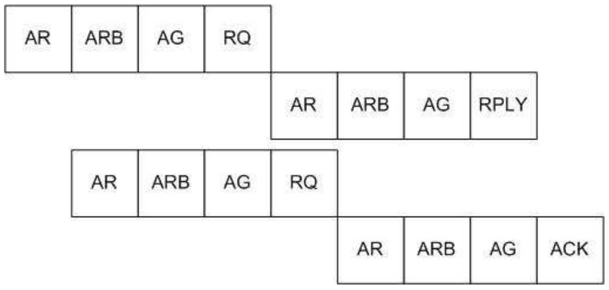


Figure7 Split Transaction

Here is an explanation for Figure7: First, we define the acronyms, Arbitration Request (AR), cycle for processing inside the arbitrator (ARB), grand signal set (AG), request signal set (RQ), reply from memory or IO (RPLY), acknowledgement from memory or IO (ACK). The messages are split and pipelined in order to prevent the <pause> cycle from the pipelining method (the <pause> was used to wait for the reply), so the processor does not have to wait for the reply anymore. In Figure7, we have 4 rows, the first and third are the same and the second and fourth are the same, they are requests with their corresponding replies. In the first and third row, we find the steps for a request to a memory or IO for data. In the second and fourth row, we find the steps for a reply from the memory or IO to the corresponding processor.

The burst message technique is used to reduce the time consumed by the arbiter. Instead of passing through the arbiter each time a master wants to use a bus, it can get access to the bus first, and then he can send multiple data transactions one after the other without having to ask the arbiter again. The arbiter allows this for a certain time only, to keep fairness in the bus allocation.

Avalon Bus

The Avalon bus was designed specifically for Altera applications because in an FPGA, there are more possible interconnections. Altera designed this bus keeping in mind that they directly connected some of the units together to prevent useless traffic on the bus. The principle goals found in the design of this bus were: Address Decoding, Data-Path Multiplexing, Wait State Insertion and Arbitration for Multi-Master Systems. The Avalon has several types of transfer capabilities which are: Slave Transfers, Master Transfers, Pipelined Transfers and Burst Transfers. Figure8 shows the Avalon bus used in a Nios II processor design.

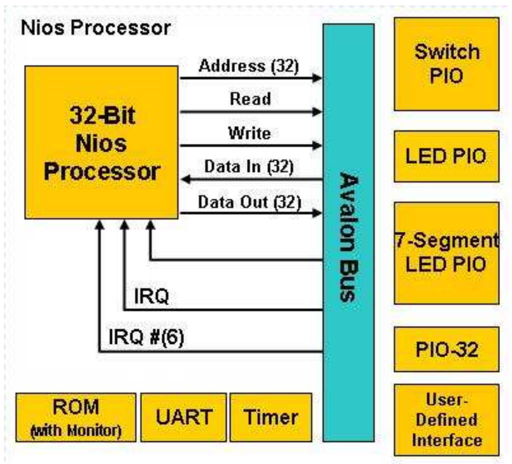


Figure8 Avalon bus in an example design

This bus uses a slave based arbitration method, where an arbiter is placed near and in front of the slaves only when several masters can access that slave. It uses a round-robin combined with priority allocation technique. The Avalon bus also supports advanced bus allocation techniques such as pipelining and burst messages. The bus has different lines for data, address and control signals. The Avalon bus uses synchronous timing and the connection between the masters and slaves are not direct; they pass through an Avalon interface (or Avalon blocks).

The Avalon bus uses a fairness arbitration scheme. It consists of assigning an integer to each pair of Master/Slave, called “shares”, which is the same principle of priority. Then, the master with the most shares can access the bus until he uses all his shares. Then, the next master having the most shares uses the bus, and so on.

Now, we will discuss on the Avalon bus master reads and slave reads. In order to do a master read, the master must provide the correct address and the read signal. Then, the Avalon switch fabric needs to raise a wait signal until the data is ready. When the data is ready at the Avalon switch fabric, the wait signal drops and the master can read the data through the Avalon switch fabric. For a slave read, the Avalon switch fabric needs to provide the address, decodes it and trigger the chipselect signal. Then the Avalon switch fabric raises the read signal and the slave port presents the readdata. The Avalon switch fabric then takes that data. [48] The master read implies that the master reads some data from the Avalon switch fabric and the slave read implies that the Avalon switch fabric reads data from a slave.

Summary

This document introduced the single bus and multiple bus interconnection networks. During this section, it was clear that a multiple bus systems brought a better performance, but it is more costly in terms of hardware. Then, the next section showed how processors could communicate using multiple addressing modes and the differences between synchronous and asynchronous timing. After, the study of the arbitrator’s functionality and how it takes its decisions was done. Also, advanced bus allocation techniques were studied, like bus pipelining, split transactions and burst messages.

Finally, the document included an example of a bus using the theory seen in the previous sections, the Avalon bus.

To improve a system's performance that is using a bus based interconnection network, the physical implementation of the bus or buses must be designed carefully. Also, the arbitrator should be implemented so it can make his decisions in the most efficient way, taking into account priority and fairness. The use of advanced bus techniques also have to be implemented to see a change in the performance of a bus. Maybe it would even require to do some research for a customized advanced bus allocation technique.

Today, increase in performance caused by improving physical designs of processors or other units (e.g. size/number of transistors on a chip) has almost reached its limits. While other technologies are not released, we will have to work on interconnection network's designs if we want to increase some performance measures.

Paper 1 **[49]** Performance of multiprocessor interconnection networks

Paper 2 **[50]** Multilevel Bus Networks for Hierarchical Multiprocessors

Lecture #7: Dynamic Interconnection Networks

Lecturer: Miodrag Bolic
Scribes: Aaron Sadhankar, Ming-I Lu
Date of Lecture: October 4, 2006

Introduction

This lecture provides a general description of Dynamic Interconnection Networks used in multiprocessor systems. Interconnection Networks (INs) are required in multiprocessor system architectures in order for communication between processors, and the sharing of resources such as shared memory. There are two types of interconnection networks used, static and dynamic. Static network interconnections are established in design, whereas dynamic networks establish connections during operation. This lecture focuses on the dynamic interconnection network, and the various topologies that can be used.

Dynamic interconnection networks are further classified as either bus-based, where the method of interconnection is a bus, or switch-based, where switching elements are used for interconnection. This lecture focuses on the latter method of interconnection. Switch-based networks from an architecture and design perspective can follow three different structures:

1. Single-stage,

2. Multistage, and
3. Crossbar.

The various interconnection network designs have differences related to cost, complexity, and performance. A comparison between these identifies the advantages and disadvantages of each design. This is also directly related to the application a design will be used for.

Explanation

Switch-Based Networks

Switches are simple devices used to route inputs, which can be physical lines, to outputs. Each type of switch-based interconnection network uses switches, with the operation and number of inputs and outputs for the switches dependant on the design. The basic element of switch based networks is a 2×2 switch, which has four different settings that can be used. A *straight* switching setting sends an input directly to the corresponding output (input 0 to output 0). An *exchange* setting sends an input to another output (input 0 to output 1). The terminology of "upper" and "lower" can be used, so for example if input 0 and output 0 represent the upper switch input and output, then sending input 0 to output 0 is an upper to upper switch transfer. In addition to straight and exchange switch settings, a *broadcast* setting, either the upper input to both lower outputs or the lower input to both upper outputs, can be used. Figure 1 illustrates these four different switch settings.

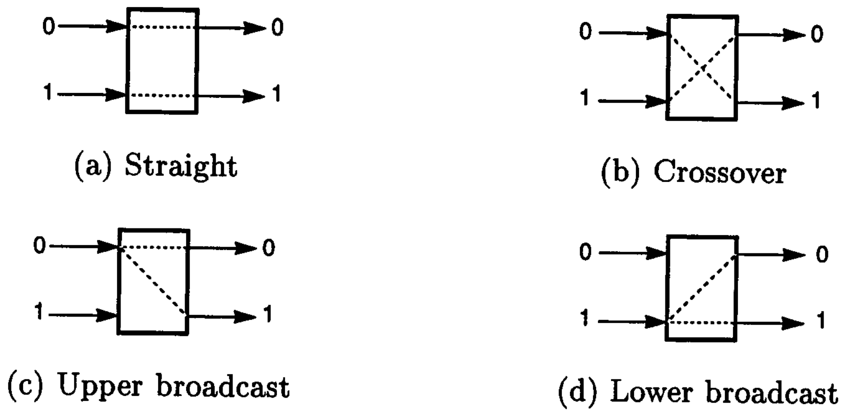


Figure 14: Settings for a 2 x 2 switch element [51]

The layout and use of switches determines the type of network topology. In a *single-stage* network there is only one set, or stage, of switching elements between the inputs and outputs. In a *multistage* network, more than one stage of switches is used between the inputs and outputs of the network. In *Crossbar* networks, all input lines intersect with all output lines, and consist of switches at each of these intersections.

Single-stage Networks

In a single-stage network the inputs are sent to the outputs through a single set of switches. Methods are needed to change the interconnection pattern so that a given input is sent to the desired output. One common method is to circulate the input and output connections until the desired destination is reached. This is called *Shuffle-Exchange*, since a shuffle operation (on the input number) is performed first followed by an exchange operation. For shuffle, the bit representation of an input (i.e. Input 3 = 011_2) is cyclically shifted to the left by one bit (011_2 -> 110_2). Then for exchange the least significant bit is inverted (110_2 -> 111_2), and this is the output ($111_2 = 7$). However, the first Shuffle-Exchange operation might not give the desired input-output connection, so another operation is done until the connection is reached. This method is the most common method for a simple single stage network; however there are other methods to determine the interconnection pattern of a single stage network. These methods, which were not explored in the lecture, provide different network layouts and certain methods may be better suited depending on the desired design.

Multistage Networks

Multistage Interconnection Networks (MINs) were introduced to improve the performance over single-stage networks while maintaining an acceptable cost. Multistage interconnection networks use the method of routing data in Inter-stage Connection (ISC) patterns [56]. These inter-stage connection patterns are kept between stages of switches. Each stage of connections is always the same, with switches changing dynamically depending on the desired routing.

There are various configurations for multistage interconnection networks. They all follow the same basic concept, with the pattern of inter-stage connections being different. Each pattern can be better than others in different applications, depending on the layout of the system and how the particular application is designed.

Blocking

MINs allow an interconnection network to have the property that a given input can be routed to a given output through more than one path. This allows an input/output pair to be connected even though other input/output pairs are connected at the same time. However, there is an extent to which this is able to occur. If at a certain point an input/output pair is unable to be connected because all available connections are currently in use, then this is referred to as *blocking*. In terms of blocking, MINs can be classified into three categories [56]:

1. Blocking Networks,
2. Rearrangeable Networks,
3. Nonblocking Networks.

Blocking Networks have the property that for a certain connection situation in the MIN, an input/output pair will not be able to establish a connection. *Nonblocking Networks* are

MINs that simply do not have this property, so any a given time an input/output pair not currently connected can establish a connection. *Rearrangeable Networks* satisfy the nonblocking property by rearranging connections so that all requested simultaneous connections are satisfied. The category a MIN falls under is dependent upon the design type of the network, which is in turn dependent on the ISC pattern that is used.

Shuffle-Exchange ISC

In the *Shuffle-Exchange Network* (SEN) the shuffle-exchange function, as used in the single-stage network, can be used up in a multistage network also. By using stages of switches, data can get to the desired destination faster. This method is the easiest in routing data and is the most commonly desired among the multistage interconnection networks. An example of a SEN is shown in Figure 2. Note that the number of stages depends on the number of inputs/outputs. If N is the number of inputs/outputs, $\log_2 N$ stages are required. Here a 4×4 SEN has $\log_2 4 = 2$ stages. Any input can be routed to any output, and in Figure 2 it is shown how input 00 is routed to output 11, and input 10 is routed to output 00. It can also be seen that multiple simultaneous paths can be established (in this particular example blocking is not encountered for the requested connections).

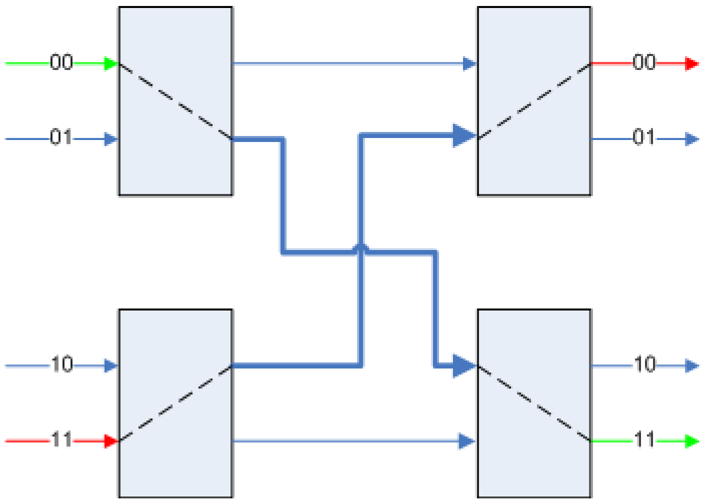


Figure 15: Example of a 4×4 Shuffle-Exchange Network

Omega Network

The *Omega Network* is another well known MIN type [56]. It is made out of elements of SENs. A size N Omega network has $\log_2 N$ elements consisting of SENs. Each stage has a column of N/2, and 2×2 switches whose inputs are shuffle inputs. The method is to take the bit representation of the output number to determine the routing path. In essence the design is based on single-stage SENs, one for each stage of the network. Switches change settings according to the bit numbers. The rule used by the switches is that the upper output is used whenever a 0 bit is encountered, and goes to the lower output whenever a 1 bit is encountered. This type of network allows the input/output paths to be independently determined by the switches; however a blocking situation can still occur in the Omega network.

Baseline Network

A *Baseline network* is a design type for dynamic MINs where the inter-stage connections are created recursively using the method of cyclic bit shifting. The bit representation of an input is shifted right by 1 bit to make the output number. However between stages the next stage is always cut into 2 separate sections. So the second stage would have 2 sections and the third stage would have 4 sections. In each section the connection setting stays the same to shift right 1 bit of the inputs. The switch rules are similar to Omega networks. Because of this recursive method a desired network can be obtained by an existing Baseline network by rearranging the switches.

Crossbar Networks

The *Crossbar* is a special type of dynamic interconnection network. The advantage of using this type of network is that there will always be many routes to get to the desired destination directly, eliminating blocking. It uses *N* square switches in an *N*-by-*N* network configuration. Each of the switches is placed at the intersection of data lines coming directly from the inputs and outputs. This allows every input and every output to be connected simultaneously. Figure 3 is an example of a 4×4 crossbar network. The behaviour of the switch elements can be seen here with two switch settings that are used, either a "diagonal" setting or a "straight" setting.

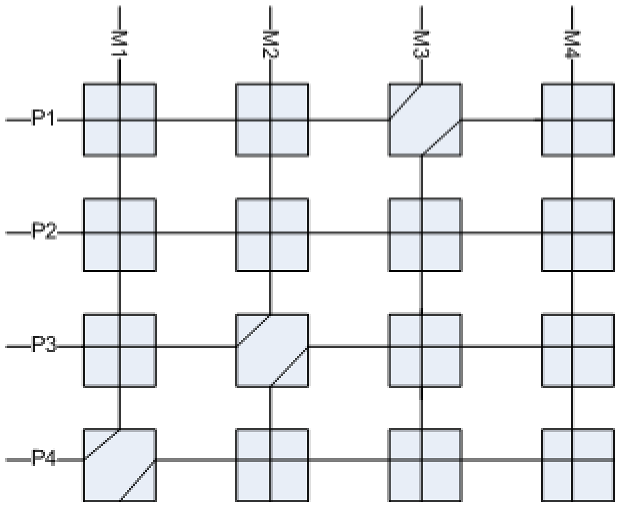


Figure 16: An example of a 4×4 crossbar network

A crossbar network offers a performance advantage over other networks. Due to the fact that there are so many data routes, if the processors would like to access different shared resources, such as memory, blocking is not an issue (the crossbar is a nonblocking network). The trade-off here is a crossbar design requires more switching elements than other network types, increasing cost.

Summary

Applications in Modern Systems

The design and implementation of interconnection networks in modern parallel processing systems are heavily dependent on cost while trying to maximize performance. These dictate which interconnection network configuration is best suited for use. Mass market consumer products will have much different purpose and design implementation than highly specialized computer architectures. On one end of the parallel computing spectrum there are consumer-level processors that have been recently introduced which have independent processing cores on a single chip. Intel Corporation’s consumer “multi-core” processors are an example of this. As mentioned in [52], current consumer “dual-core” processors utilize interconnection through shared memory and a bus. This is a simple design compared to more complicated switch-based networks many processors. As this technology progresses and the number of independent processing cores increase, however, more advanced interconnection techniques may have to be used.

On the other hand, highly specialized performance-oriented applications, such as supercomputers, require an entirely different design methodology. This becomes much more apparent for designs with a large amount of parallel processors. An example of this is the University of Illinois at Urbana-Champaign’s current development of the *ILLIAC 6* supercomputer. The interconnect design methodology here is to use an N-by-N crossbar for each level of the system [53]. It is classified as a *bidirectional shuffle exchange network*, and because each elementary component functions as crossbar networks an interconnect hierarchy is formed. The primary considerations here are on programming model, performance, error and tolerance recovery, and packaging. Figure 3 is a diagram of the ILLIAC 6 chassis, which contains a total of 32 processor boards (containing 32 processors each, for a total of 1024 processors for each of the 64 chassis). Each board can function as a 16-by-16 crossbar, with the overall chassis functioning as a bidirectional 4-by-4. Because of this configuration, any boards can communicate with each other even in the event of a board failing.

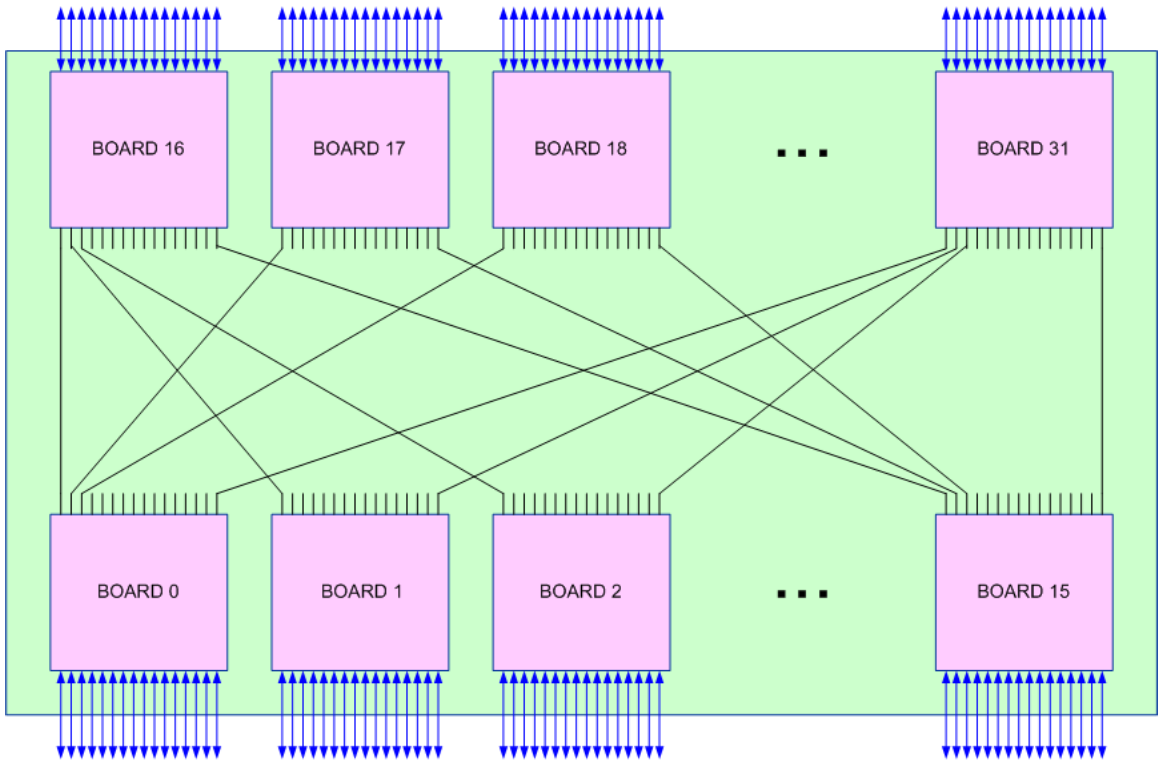


Figure 17: The ILLIAC 6 Chassis

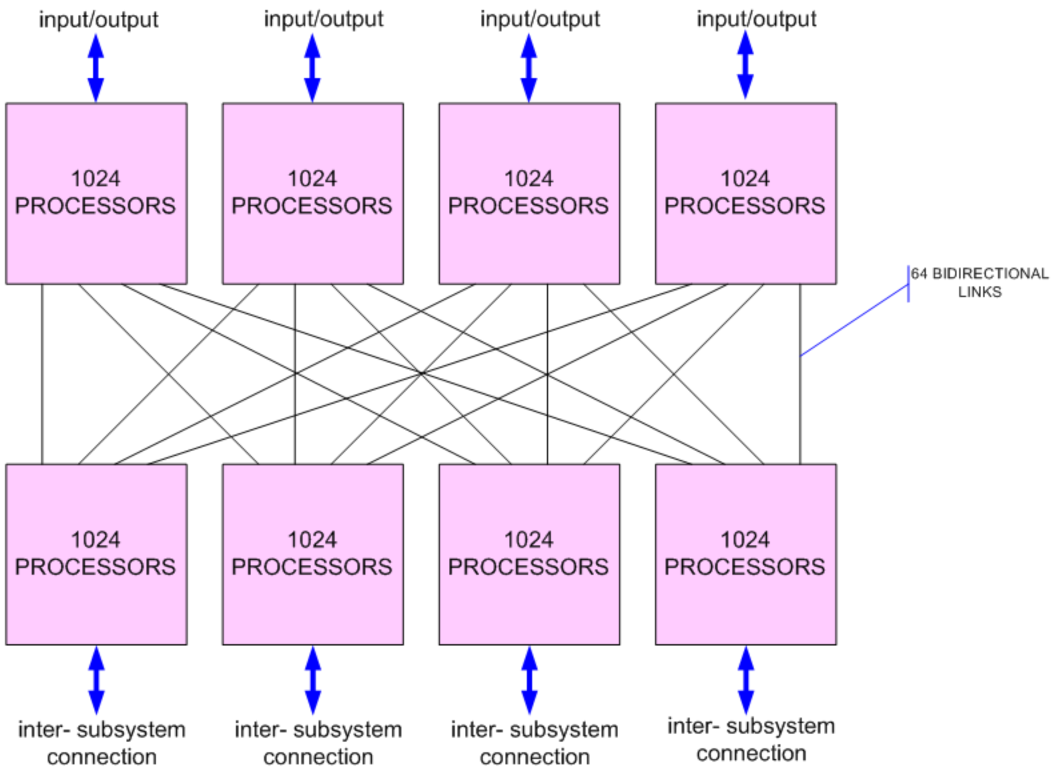


Figure 18: The ILLIAC 6 Subsystem (consisting of 8 chassis units)

As seen in Figure 4, the planned ILLIAC 6 architecture is a highly parallel system, containing a total of 65536 processors. This shows how different the various parallel processing system architectures can be, and as a result of this the interconnection methodology must be designed appropriately for each.

Paper 1: Architectural Choices in Large Scale ATM Switches
Paper 2: Performance Analysis of Future Shared Storage Systems

Lecture #8: Dynamic Interconnection Networks

Course: CEG 4131 – Computer Architecture III
Lecturer: Professor Miodrag Bolic
Scribe: Edmond Goon (3021795) and Gilles Poitras (3035209).
Date: October 4th, 2006

Introduction

The theme of this lecture deals with dynamic interconnection networks. The lecture will outline various topologies for interconnection networks and other issues regarding this subject. The interconnection networks (INs) that will be examined are the following: omega networks, baseline networks and crossbar networks. In order to better understand the functionality of these topics, a review of switches will also be outlined. Lastly, a brief performance analysis of different interconnection networks will be reviewed.

Explanation

1 Switches

The various interconnection networks examined in this lecture all utilize switching elements (SE) in various configurations. Thus it is important to understand the functionality of these elements. In switch-based interconnection networks, the switch elements are utilized to establish the connections between processors and memory modules. [58] These switches are available in four different configurations. Using the different configurations of SEs, we can create INs with different permutation connections and legitimate states. These two values determine the number of ways inputs can be connected to outputs and vice-versa according to the number of switches used in the design. The four configurations are outlined in the figure below. [57]

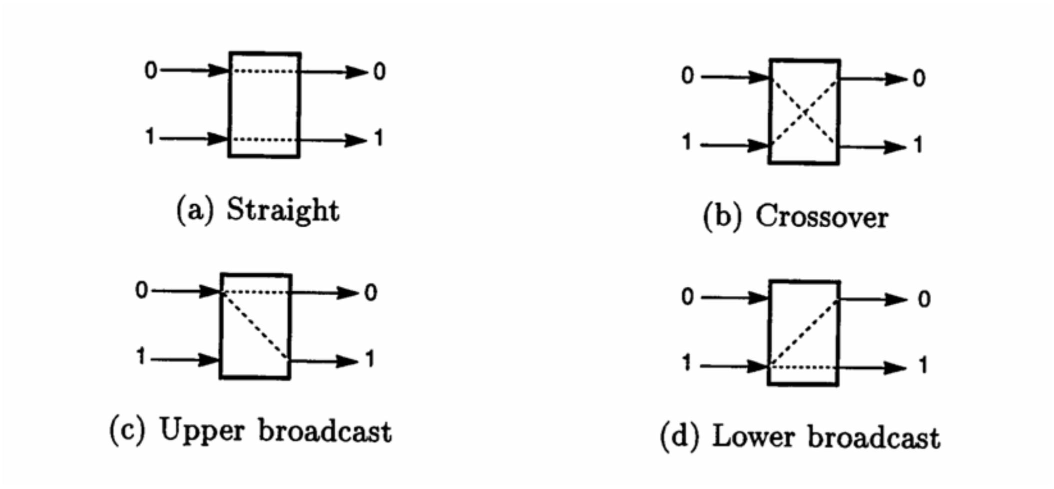


Figure 1 The different settings of a 2x2 Switching Element (SE). [59]

2 Multistage Interconnection Networks

Due to the limited capability of single stage networks, multistage interconnection networks (MINs) were developed as a method to improve on the limitations while at the same time keeping costs low. A MIN consists of several cascaded single stage networks (a set of 2 x 2 switching elements) connected by an Inter-stage Connection Pattern. The switches used perform their own routing based on the destination address (self-routing). For N x N MINs, there are $\log_2 N$ stages and that is also the number of bits in the destination address. [57] For routing, each bit of the destination is used for one stage, and the address is read from left to right. The standard for a MIN is that a 0 as the control signal means the communication should be routed through the upper output of the switch whereas with a 1, the signal is routed through the lower output. With most INs, the values for the control signals come from the destination address. [58]

Multistage Interconnection Networks can be put into 3 categories. They are blocking networks, rearrangeable networks, and nonblocking networks. In a blocking network, an interconnection that is in progress when another interconnection is attempted on unused inputs or outputs can prevent the new IC from being achieved. The block happens when a switching element would be required to be in two different states simultaneously. Blocking networks include omega networks and baseline networks. Rearrangeable nonblocking networks have the property which allows established connections to be rearranged in order that new connections be established. An example of a rearrangeable nonblocking network is the Benes network. The characteristic of nonblocking networks is that it is always possible to connect unused inputs and outputs regardless of any other connections. The Clos is an example of a nonblocking network. [58]

In the Multistage Interconnection Networks discussed in the next sections, there exists only one path between any given input and output. This allows for the MINs to be simple. However, this also means that they are 0-fault tolerant. If any given switch in the implementation fails, that path is no longer available. This is called single-point failure. To correct this, it has been suggested to increase the number of stages to $\log_2 N+1$, allowing for two paths to each destination. [58]

2.1 Omega Networks

The first type of multistage interconnection network we will examine is the Omega Network. A key feature of this network is that there is only one unique path from each input to each output. [58] In other words, every input can find a path to any output. In order to determine the connection paths between switches, the perfect shuffle algorithm is used. This algorithm is repeated at every stage of the network. It can be observed that the first and last inputs are directly connected to the destination elements. An Omega Network is said to be of size N as it is comprised of $\log_2 N$ stages of 2x2 switches. Each stage is made from N/2 of these switches. The number of switches required in the network can also be determined by using the following equation where S represents the total number of switches: $S = (N/2) * \log_2 N$. It should also be noted that this type of interconnection network is blocking. [57] This characteristic implies that when a connection between a pair of switches is being utilized, new requests for connections may not be possible as some resources are busy. To better understand Omega Networks, we will consider an example. It will demonstrate the process of connecting the different switch elements and find the routing from the input to the destination.

Example:

To demonstrate the interconnections of the Omega network, consider a network with 8 inputs and 8 outputs built using 2x2 switches. We wish to connect input 011 to output 110. Using the equations outlined above, the number of stages is calculated to be $\log_2 8 = 3$ and each stage has 8/2 = 4 switches. The total number of switches is the product of these two values, 3*4 = 12. The next step is to connect the inputs to the first set of switches. To do this, we use the perfect shuffle algorithm which performs a shift-left on the input bits. Therefore, 011 (3) would be connected to the switch representing 110 (6). This process is repeated for each input and replicated at each stage. Lastly, we use the bits of the destination address to determine the path of the connection. A 0 represents the upper output of a switch and a 1 represents the lower output. Starting from the MSB, the output 110 would have the following routing: lower, lower, upper. The final Omega Network layout and routing (in red) is shown in the figure below.

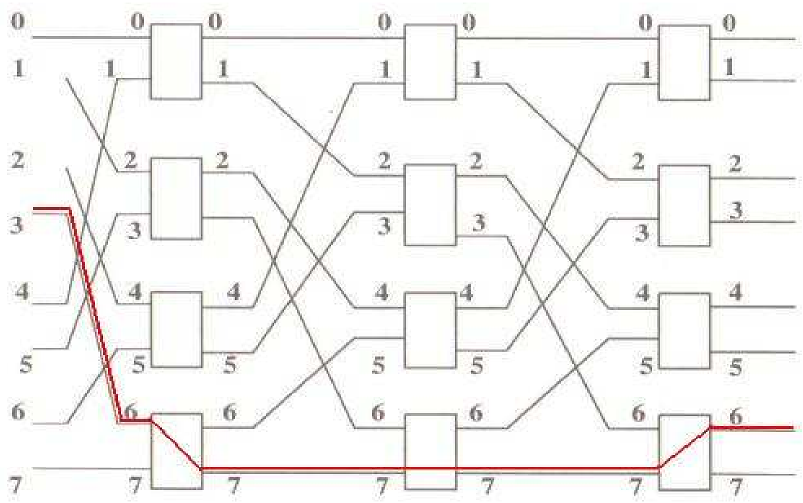


Figure 2 Omega Network layout and input to output connection routing. [58]

2.2 Baseline Networks

Baseline networks are established in a similar fashion to omega networks. Once again there is a unique path from every input to every output. Similarly to omega networks, baseline networks are also blocking networks. [58] The number of stages in a baseline network can be found by doing $\log_2 N$. The main difference between baseline networks and omega networks is that the former uses an inversed shuffle. Furthermore, a baseline network does not reuse the same interconnection. A large network is shared into 2 networks, then 4 networks, then 8 networks, then 16 networks, and so on. Baseline networks can be generated recursively and the first stage is N x N, second stage is N/2 x N/2, third stage is N/4 x N4 and so on. Baseline networks can also have topologically equivalent networks if one network can easily be reproduced by rearranging nodes of another network at each stage. [57] To determine the interconnections of a baseline network, right shift algorithm is used. An example of with 16 inputs is shown below.

Input	**Shifted input**
0000 (0)	0000 (0)
0001 (1)	1000 (8)
0010 (2)	0001 (1)
0011 (3)	1001 (9)
0100 (4)	0010 (2)
0101 (5)	1010 (10)
0110 (6)	0011 (3)
0111 (7)	1011 (11)
1000 (8)	0100 (4)
1001 (9)	1100 (12)
1010 (10)	0101 (5)
1011 (11)	1101 (13)
1100 (12)	0110 (6)
1101 (13)	1110 (14)
1110 (14)	0111 (7)
1111 (15)	1111 (15)

Table 1: First stage Interconnections in a 4 bit baseline network

The process is then repeated in the same way for the next stages of the interconnection network. The difference is that the network gets segmented at each iteration and the input is reduced by one bit at each stage.

The end product of the baseline network can be seen in Figure 3. As an example, we'll look at the connection from 1001 to 0100. Following the convention mentioned above that routing is based on the destination address and that 0 is the upper output and 1 is the lower output, we can see that the trajectory that should be followed is upper, lower, upper, upper. Once again, the routing is outlined on the network in red.

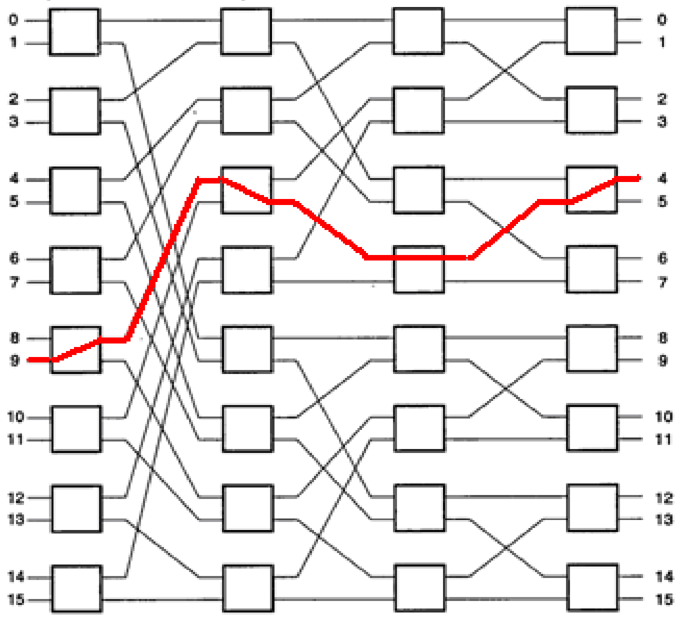


Figure 3 Baseline Network layout and input to output connection routing. [59]

2.3 Crossbar Network

The crossbar network is also built using switching elements. These switching elements are found at every intersection of lines in the network. These intersections can be used to connect the rows of processors with each column of memory. Only one switch from each column may be active at any given time. The combination of having only one switch from each column and only one switch from each row activated means any given processor could access a specific memory element. The access control would typically be managed by an arbiter. [58] The main advantage of this network is its potential for speed since it only requires one clock cycle to establish a communications link between destination and source. Another advantage of this network is that it is typically a non-blocking network but blocking will occur when the destination is already in use. The diameter of such networks is 1 since this is the shortest path between any two nodes. The main drawback of this network design is the high cost of implementation and its complexity. Because of these issues, they are generally used to implement small switches which are then used in large MINs. [57]

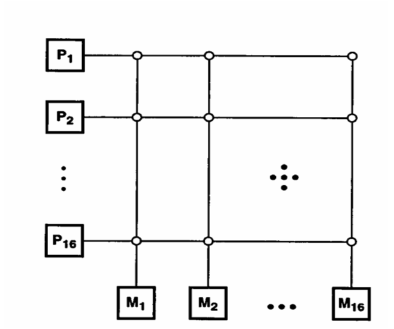


Figure 4 Example of a Crossbar Network topology. [59]

3 Performance Comparison

It is possible to find a way to describe the complexity of various issues for INs. The tool used here is the Big-Oh notation. This tool describes the asymptotic behaviour of mathematical functions. Because of this, it is also extremely useful in the analysis of algorithms. [60] This notation will be utilized here to compare some aspects the following three interconnection networks: Bus, MIN (such as Omega and Baseline) and Crossbar. The facets that will be examined are: latency, switching and wiring complexity and blocking. From the table, we can observe different trade-offs between designs. Some more characteristics which must be considered when choosing a design are bandwidth, node degree, diameter and level of symmetry.

Network	Latency	Switching complexity	Wiring complexity	Blocking
Bus	Constant O(N)	O(1)	O(w)	yes
MIN	$O(\log_2 N)$	$O(N\log_2 N)$	$O(Nw \log_2 N)$	yes
Crossbar	O(1)	$O(N^2)$	$O(N^2 w)$	no

Table 1 Performance comparison of different interconnection networks. [57]

Summary

4 Commercial Example

An example of a commercial product which utilizes an implementation of a Crossbar network is the Nexus core from Fulcrum Microsystems. This core is used in the PMC-Sierra dual MIPS processor RM9000 and acts as the processor switch between two 1 GHz processors. [57] The switch provides 64 Gbit/s of inter-CPU bandwidth for the data transfer between the two processors. The utilization of the switch designed by Nexus in combination of a five-state cache coherency protocol results in an extremely sophisticated multiprocessor architecture. This processor targets commercial applications such as wireless base stations and edge routers. [61]

Paper 1 Nonblocking Properties of Interconnection Switching Networks

Paper 2 Crossbar based design schemes for switch boxes and programmable interconnection networks

Lecture#9: Static Interconnection Networks

Course: Computer Architecture III
Lecturer: Miodrag Bolic
Scribes: Sertan Gun #2663718, Jonathan Plourde #2846371

Date: October 6, 2006

Introduction

Static networks are opposite of dynamic networks in terms of network status, meaning that static networks are fixed and can be unidirectional or bi-directional between processors. There exist two types of static networks. Please refer to the dictionary at end for any terms.

- Completely connected Networks(CCN)
- Limited Connection Networks (LCN)
    - Linear Arrays
    - Rings (Loops)
    - Two-Dimensional Arrays and Tori
    - Tree Networks
    - N-Cube Networks

Terms that have been used in this paper:

Dictionary of the words used in this lecture:

(Node Degree) *d= the number of edges incident on a node.*

(Diameter) D*= the maximum shortest path between any two nodes.*

Bisection width *is the minimum number of wires that must be cut to divide the network into two equal halves*

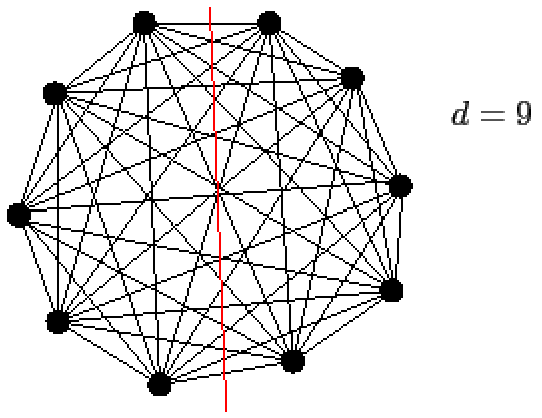
Network latency: *worst-case time for a unit message to be transferred*

Functionality: *how the network supports data routing, IRQ handling, synch, Req/Msg combining & coherence*
Hardware complexity: *implementation costs for wire, logic, switches, connectors, etc.*
Nonblocking: *A network is called strictly nonblocking if it can connect any idle input to any idle output regardless of what other connections are currently in process*
Re-arrange-able nonblocking: *In this case a network should be able to establish all possible connections between inputs and outputs by rearranging its existing connections.*
Blocking interconnection: *A network is said to be blocking if it can perform many, but not all, possible connections between terminals. Example: the Omega network*

Explanation

Completely Connected Networks (CCN)

The first networking strategy we are going to cover is completely connected networks (CCN). A CCN consists of any number of nodes, where all nodes are connected to each other. The network diameter is therefore $D = 1$ and the node degree is $d = N - 1$ (a node is connected to all other nodes, except itself). Bisection width is the minimum number of wires that must be cut to divide the network into two equal halves. Low bisection width means low data transfer and high bisection means high level of data transfer may happen.

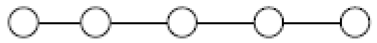


- Shown in the figure above, the Bisection Width of a CCN is $b = N^2/4$. This becomes cumbersome when the number of nodes is large as the number connections increases by $O(N^2)$. [66]

Limited Connection Networks (LCN)

Linear Arrays

As shown in the figure below, a linear array's nodes are connected to each other, forming a straight line. This is an asymmetric network: all nodes have a degree of 2, with the exception of the end nodes, which have a degree of 1. It is also shown in this figure, that this network has a bisection width of 1.



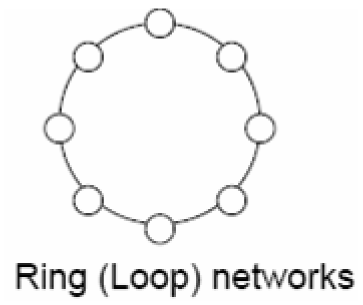
Linear arrays

[66]

This topology is similar to a bus, but there are several differences to note. First, there is no direct connection between master and slave: messages need to hop from node to node. Second, several nodes can transmit at the same time, provided that each channel is used by only one node. Finally, one serious disadvantage is that this network's diameter increases proportionally with the number of nodes. As a result, this topology is not scalable.

Rings (Loops)[66]

The ring topology attempts to solve the "large diameter" problem inherent in linear arrays. As shown below, the ring is simply a linear array, with its end nodes connected together. This has the effect of making the network symmetric: all nodes have a degree of 2.



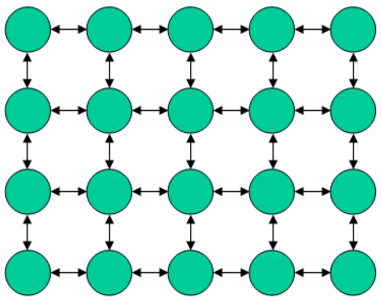
Since the topology of the network is now circular, the mode of communication must be defined. Communication can now be either unidirectional, or bi-directional. Unidirectional communication does very little to solve the "large diameter" problem; however, bi-directional communication has the effect of dividing the network diameter by 2.

Finally, because of the addition of the new connection, the ring topology has a bisection width of 2.

Multi-Dimensional Arrays and Tori

Arrays

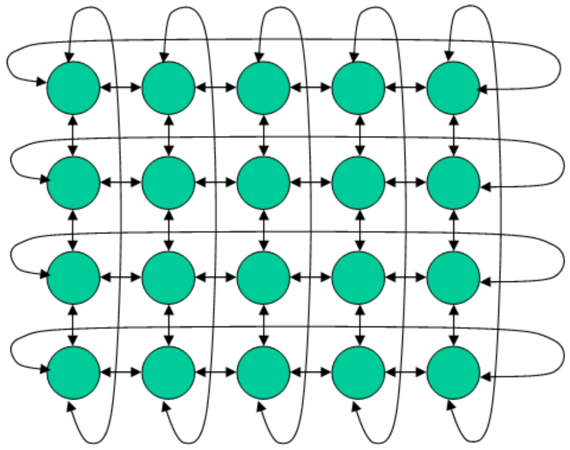
The two-dimensional array is a popular topology, since it directly reflects many data processing applications, such as image processing. It is also relatively easy to manufacture, using planar integrated circuits and (X, Y) connections [64]. It is an asymmetric network, where the corner nodes have d = 2, the sides d = 3, and the centre nodes d = 4. A multi-dimensional array has N = n^k nodes. The bisection width of a k-dimensional mesh is $b = \sqrt[k]{N}$.



Intel Paragon (2D mesh) [65]

Tori [65]

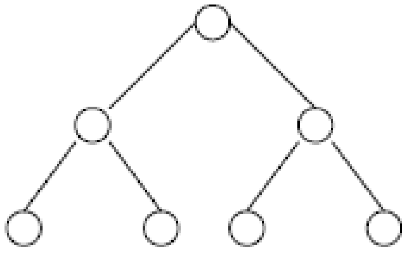
Like the ring topology, the torus topology attempts to decrease the network diameter, for a given number of nodes. The usual diameter of a 2-dimensional mesh is $D = 2\left(\sqrt{N} - 1\right)$. The torus, on the other hand, has a diameter of $D = \sqrt{N} - 1$, effectively reducing the diameter by a factor of k. However, this has the effect of increasing the bisection width by the same amount from $b = \sqrt{N}$ to $b = 2\sqrt{N}$. Furthermore, the torus network is symmetric since all nodes now have a degree d = 4. This greatly simplifies the hardware design process.



Tree Networks

In a tree network, the processors are located at the leaves all other nodes are switches. A k-ary tree has height $h = \log_k N$ and a diameter of $d = 2h$.

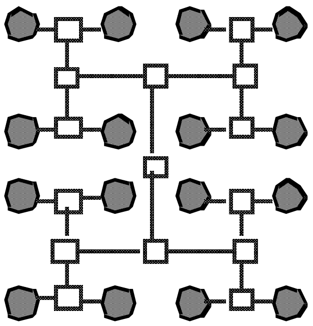
[66]



Tree networks

Unfortunately, as shown in the diagram above, the bisection width of the tree is $b = 1$ resulting in poor bandwidth at the root level. One solution for this problem is the fat tree (discussed below).

Tree networks can also be easily placed, thus simplifying VLSI design. The diagram below demonstrates the efficient 2D layout of a tree network.



[66]

Fat Trees

The fat tree solves the bandwidth problem by doubling the number of connections at each level in the tree; each processor, however, still has a degree of 1, as shown in the figure below.

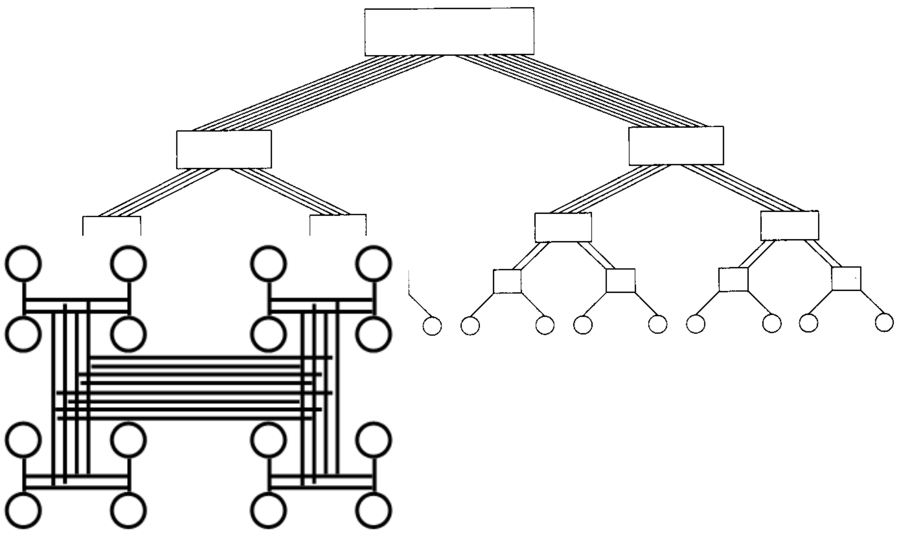


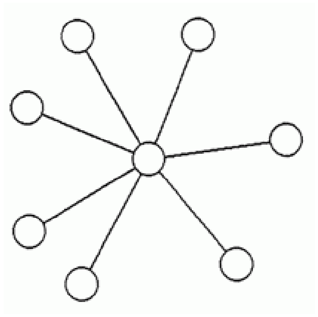
Diagram on the left hand side demonstrates the layout of fat tree network. [66]

As a result, the bisection width of the tree network is now $b = N/2$. P3

Star Topology

The well-known star topology is a special case of a tree network. In this case the degree is $d = N - 1$ (see central node), and the network diameter is always 2. However, the central node is a serious point of failure: if it fails, the entire network fails as well.

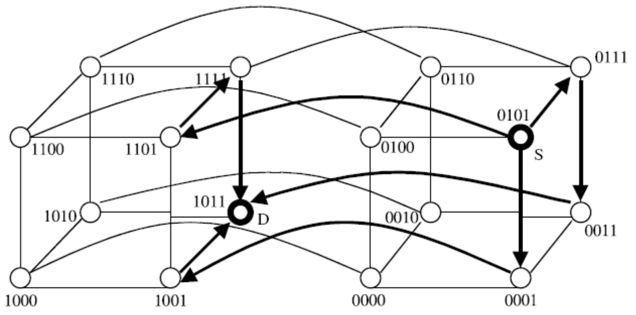
[65]



Cube Networks

In an n-dimensional cube (n-cube) network, there are 2^n processors, and each processor is connected to *n* other processors. In other words, the degree of an n-cube network is $d = n$. The nodes are labelled from 0 to $2^n - 1$, and two nodes are connected if and only if the binary representation of their label is different by only 1 bit (i.e. Hamming distance is 1).

[66]



Taking the exclusive OR of the source and destination labels, then counting the number of'1s' in the result computes the Hamming distance. As a result, the diameter of the network is $D = \log_2 N$. Seen in the diagram above, the bisection width of an n-cube network is $b = 2^{n-1}$. Finally, this network is symmetric, since all nodes are connected to *n* other nodes.

Message Routing in Hypercubes

The following exemplifies how messages are routed in a 4D hypercube. Refer to the diagram above. For this example we will be sending a message from 1001 to 0111. The route will be followed by reading the Hamming code from right to left.

- The first step is to compute the Hamming code for the route: $1001 \oplus 0111 = 1110$

- The first bit is 0; therefore it can be skipped.
- Since the second bit is 1, the first hop will be from 1001 to 1011. The message is now at node 1011.
- The third bit is 1, so the next hop will be from 1011 to 1111. The message is now at 1111.
- The final is also 1, therefore the next hop will be from 1111 to 0111. The message has reached the desired destination.

Summary

Comparing Topologies [65]

The following table compares the various topologies discussed in this paper.

Network	**Degree (d)**	**Diameter (D)**	**Bisection (b)**	**# of Links**
Complete	$N-1$	1	$N^2/4$	$N(N-1)/2$
Star	$N-1$	2	1	$N-1$
Binary Tree	3	$2\log\left(\frac{N-1}{2}\right)$	1	$N-1$
Linear Array	2	$N-1$	1	$N-1$
Ring	2	$\left\lfloor \frac{N}{2} \right\rfloor$	2	N
2D Mesh	4	$2(\sqrt{N}-1)$	$\sqrt{N}$	$2\left(N-\sqrt{N}\right)$
2D Torus	4	$2\left\lfloor \frac{\sqrt{N}}{2} \right\rfloor$	$2\sqrt{N}$	2N
Hypercube (n-D)	$\log_n N$	$\log_n N$	$N/2$	$\frac{N\log_n N}{2}$

Paper1 [67] Performance of Multiprocessor Interconnection Networks

Paper2 [68] Static interconnection network extensibility based on marginal performance/cost analysis

Lecture #10: Shared memory systems

Course: Computer Architecture III
Lecturer: Miodrag Bolic
Scribe: Rudy Hallé (3037323)
Marc-André Dodson (2721736)
Date: October 15, 2007

Introduction

In today's competitive technology where speed and cost are so important, the efficiency of tasks is crucial to achieve success. In the most common human handle tasks, team work is the way to go. Technology follows the same principle as multi-processors allows speedup achievement through parallel programming. Like for team work, communication is very important to achieve maximum efficiency.

One popular way to allow multiple processors to communicate is to use shared memory architecture. This lecture describes the different characteristics of shared memory systems, the paradigm surrounding the software programming, the hardware requirement for implementation and a description of Altera's solution. An emphasis is made on cache coherence and synchronization.

Explanation

1. Characteristics of shared memory systems

A shared memory system is a multiprocessor system communicating with a dedicated space in memory. Each processor can also have its own local memory as shown in figure 1. A shared memory system should allow each processor to reference directly any location of the shared memory; although the programmer should not have to worry about location of data memory. This can create NUMA system (Non-Uniform memory access) as some data can be located closer to some processor than others (on-chip, off-chip, even on different boards. Some additional memory can also be physically distributed among processors. Otherwise, it can be a UMA (Uniform memory access) system if the shared memory has an equal access time for all processors. [69]

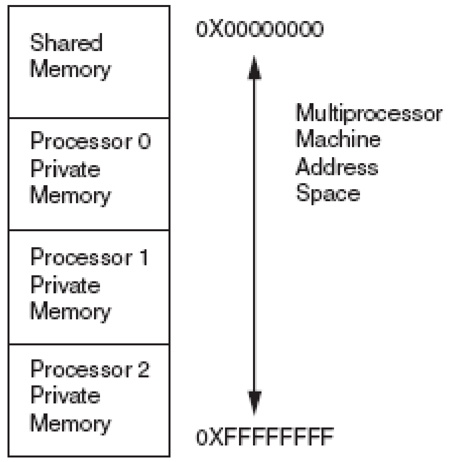

Shared Memory
Processor 0 Private Memory
Processor 1 Private Memory
Processor 2 Private Memory
0X00000000
Multiprocessor Machine Address Space
0XFFFFFFFF

Figure 19 All Processors Access Shared Memory [69]

It is also important to note that the shared memory system is not only used with traditional CPUs but also with other processor type that follow the multiprocessor trend, such as GPUs (graphic processor units). [73]

1.1 Advantages/disadvantages of shared memory systems

The use of a shared memory system for parallel processing has several advantages. For programmers, the coding is very similar to a multi-threaded application running a uniprocessor system because the resources are shared the same way among the tasks, using the same synchronization techniques, which makes the adaptation easy for them. The location of global variables can be loaded in local memory caches which increase the performance of application using extensively the shared memory. Another advantage is that the most popular platforms now offer hardware extensions to take care of memory and cache coherence. This makes the incremental cost very low to add processors to a multi-processor design.

The drawback is the higher complexity of the hardware memory and cache controllers. This complexity can become the bottleneck of the design if not well designed thus making its cost even higher to ensure its good quality.

1.2 Hardware Requirements

The hardware used to implement a shared memory system needs to address two important issues.

First, it needs to ensure memory coherency; as processors might have local cache and memory, the hardware controller must make sure that the data located in those private memory represents the latest data found in the shared memory or in the other local memory. A part of the solution is to add a valid bit to the memory schema. The locality of the latest data is a big challenge as it will be demonstrated later in this document.

Also, the hardware must support atomic operations. Atomic operations are actions that cannot be interrupted, just like single clock cycle instructions. This strategy ensures that the operations using it are not part of a race condition with other operations which is essential in synchronization algorithms.

1.3 Multiprocessor Software Functions

In an SIMD (single instruction, multiple data) or more precisely an SPMD (single program multiple data) system, each processor loads the same set of instructions using special functions to allow proper parallelism. The common functions used in multiprocessor software are the following: INITIALIZE, LOCK, UNLOCK and BARRIER. [69]

Figure 1 illustrates the use of such functions in an example of a simple program adding the elements of an array.

The INITIALIZE function is design to allow the running program to identify which processor it is run on and the number of other processors working in parallel. Such method allows the different parts of the program to dynamically assign task to specific subset of processors according to the parallel model used and the total number of processors allow the program to split logically those task among the assigned processor.

LOCK(data) and UNLOCK(data) are the functions used to allow synchronization. LOCK reserves a piece of shared data with exclusion to other processors.   If a processor tries to LOCK a piece of data that is already locked, its process will be blocked until the resources is available. The UNLOCK function simply returns the availability of the resource to the other processors. Those mechanisms should be implemented in the hardware as atomic actions to prevent the processors to access them simultaneously, leading to unpredictable results. Atomic actions are functions that are executed in one clock cycle hence preventing race conditions between multiple functions executed at the same time.

BARRIER(n_proc)is a function to wait until "n_proc" number of processors get to this instruction before executing the rest of the code.

```
INITIALIZE; //assign proc_nums and num_procs

read_array(array_to_sum, size); //read the array and array size
//from file

if (proc_num == 0) //initialize the sum
{
LOCK(global_sum);
global_sum = 0;
UNLOCK(global_sum);
}

local_sum = 0;

size_to_sum = size/num_procs;
lower_ind = size_to_sum * proc_num;
upper_ind = size_to_sum * (proc_num + 1);

for (i = lower_ind; i < upper_ind; i++)
local_sum += array_to_sum[i];

//if size =100, num_proc=4, processor 0 sums 0 to 24, proc 1 sums
//25 to 49, etc
LOCK(global_sum); //locks the sum variable so only this process can
//change it

global_sum += local_sum;
UNLOCK(global_sum); //gives the sum back so other procs can add to it

BARRIER(num_procs); //waits for num_procs to get to this point in
//the program

if (proc_num == 0)
printf("sum is %d", global_sum);
END;
```

Figure 20 Example of a SIMD piece of software [69]

This example can be interpreted as follows:

Processor 0	Processor 1	Processor 2
INITIALIZES	INITIALIZES	INITIALIZES
READS DATA	READ DATA	READS DATA
INITIALIZES GLOBAL_SUM TO 0	SETS TABLE LOWER AND UPPER INDEX	SETS TABLE LOWER AND UPPER INDEX
SETS TABLE LOWER AND UPPER INDEX	ADDS DATA TO LOCAL_SUM	ADDS DATA TO LOCAL_SUM
ADDS DATA TO LOCAL_SUM	UPDATES GLOBAL_SUM	UPDATES GLOBAL_SUM
UPDATES GLOBAL_SUM	WAITS FOR THE OTHER PROCESSORS	WAITS FOR THE OTHER PROCESSORS
WAITS FOR THE OTHER PROCESSORS	ENDS	ENDS
PRINTS GLOBAL_SUM		
ENDS		

Table 1 interpretation of the above example

The global_sum variable is locked before it is used and unlocked afterward. This ensures the validity of the content of the variable as only a single processor can write to it at a time. Although, this piece is not perfect considering that not all processors will start or execute at the same time; processor 0 could initialize the global_sum variable after some other processor would have already updated it with its local_sum. To resolve this issue, a BARRIER function should be placed after the initialization of the global_sum variable.

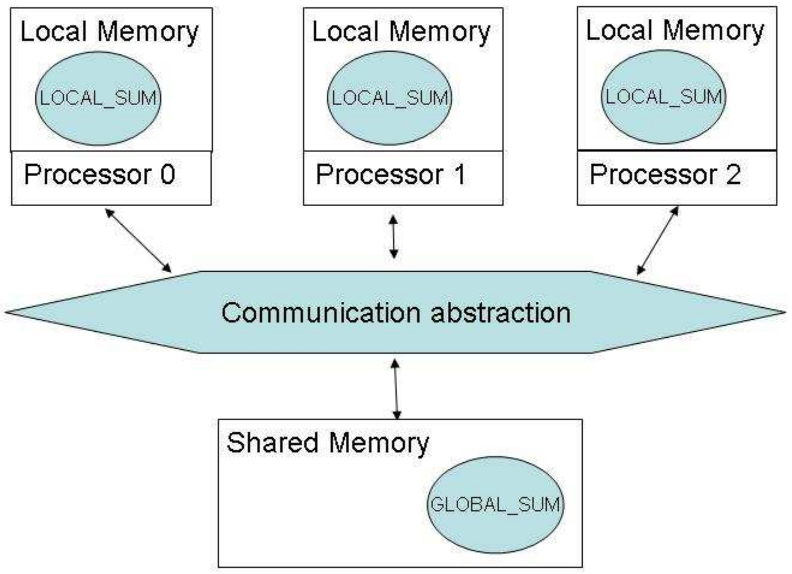


Figure 21 Illustration of the example using separate memory units for local memories

Different Architectures

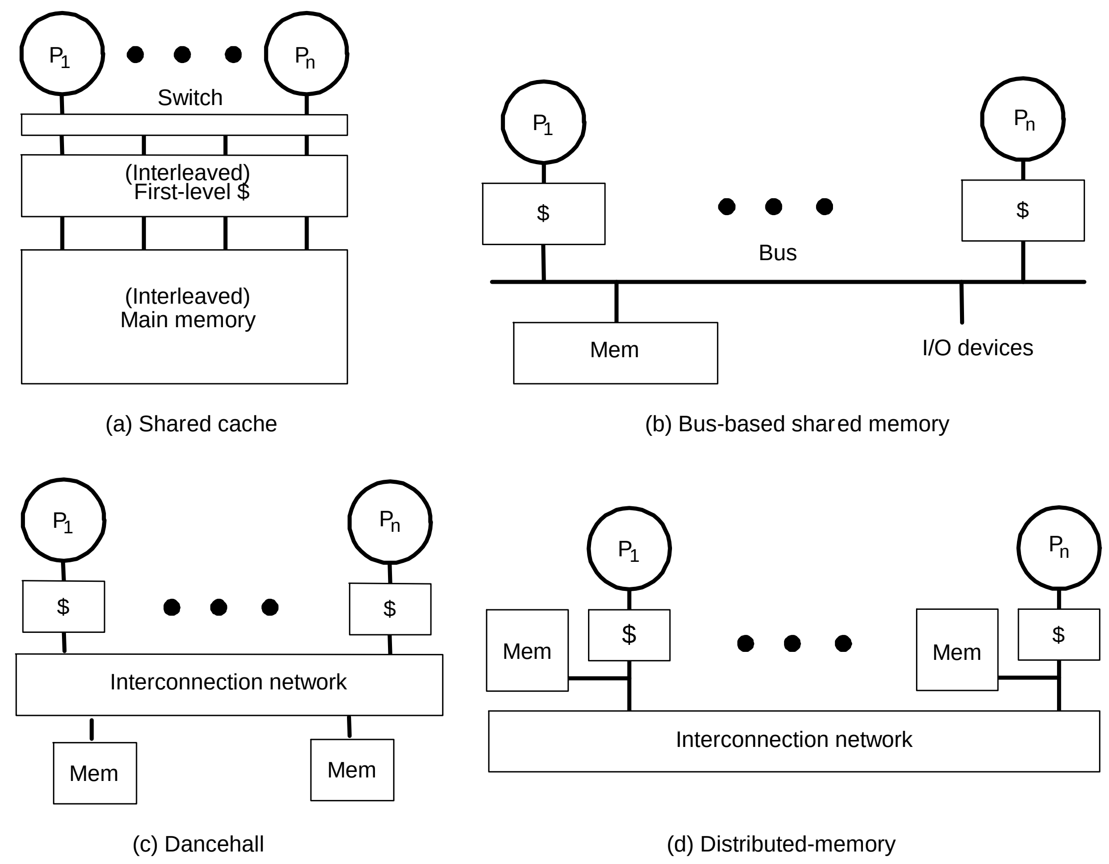


Figure 22 Natural Extensions of Memory System [70]
Shared memory systems can be implemented with many different architectures using either static or dynamic interconnection networks. The shared cache architecture solves completely the cache coherence issue but needs a bigger cache making it more expensive. The most common is the bus-based shared memory, in which each processor has its own cache and main memory is accessed through a bus. Many buses can be used to allow several processors to access main memory concurrently. The dancehall architecture uses an interconnection network to allow many processors to access many shared memories simultaneously. The distributed system also uses an interconnection network but the shared memory is distributed among processors. More detailed performance analyses are available in the lectures on dynamic interconnection networks. [71]

2 Memory Organization for NIOS II processors

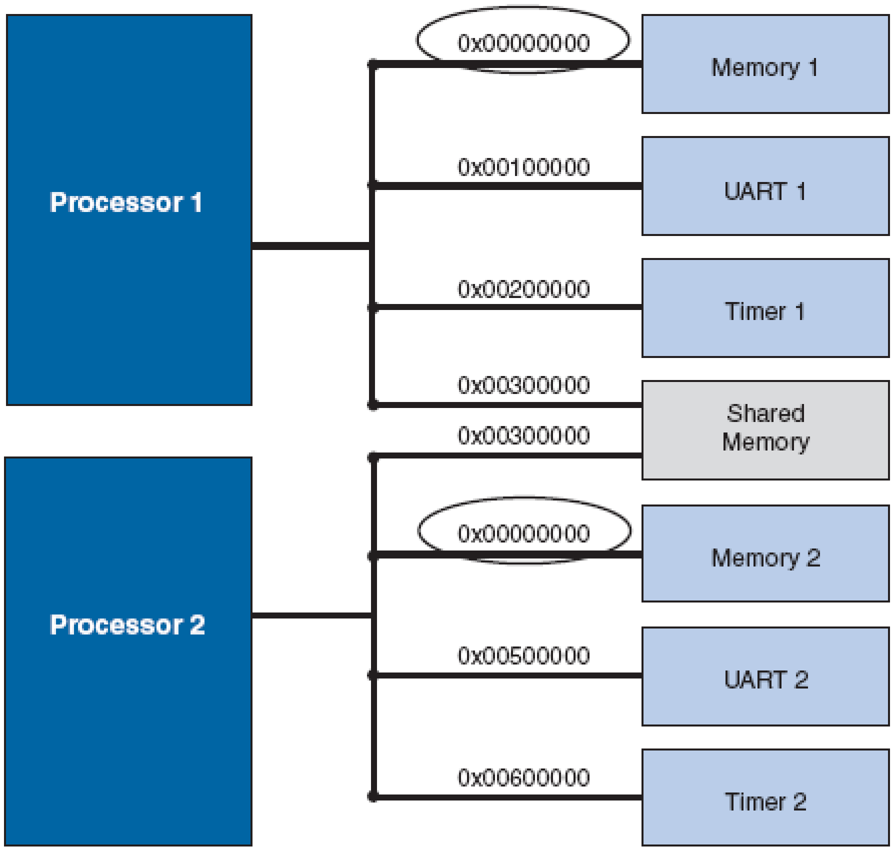


Figure 23 FPGA Design example [72]

In Altera's SOPC designs, memory is accessed independently per processor. Hence, every processor can use the same memory addresses to point at their own local memory location as seen in figure 5. This is possible in NIOS II SOPC design because memory access is done using base addresses set in SOPC's configuration to which the processors add offset values to reference the required memory address. This allows the different processors to address independently their memory although the memory address of the shared memory must be the same for every processor.

3 Cache Coherence

Like mentioned earlier, shared cache is not the optimal solution as it is slower and more expensive. Local caches are still used to gain speed as they reduce average data access time and bandwidth demands placed on shared interconnect. Although, local caches introduce a problem as different copies of variables can be located in multiple location. Those variables can be updated at different times and those updated value needs to be visible by all processors. This is the cache coherence problem.

3.1 Inconsistency in Data Sharing

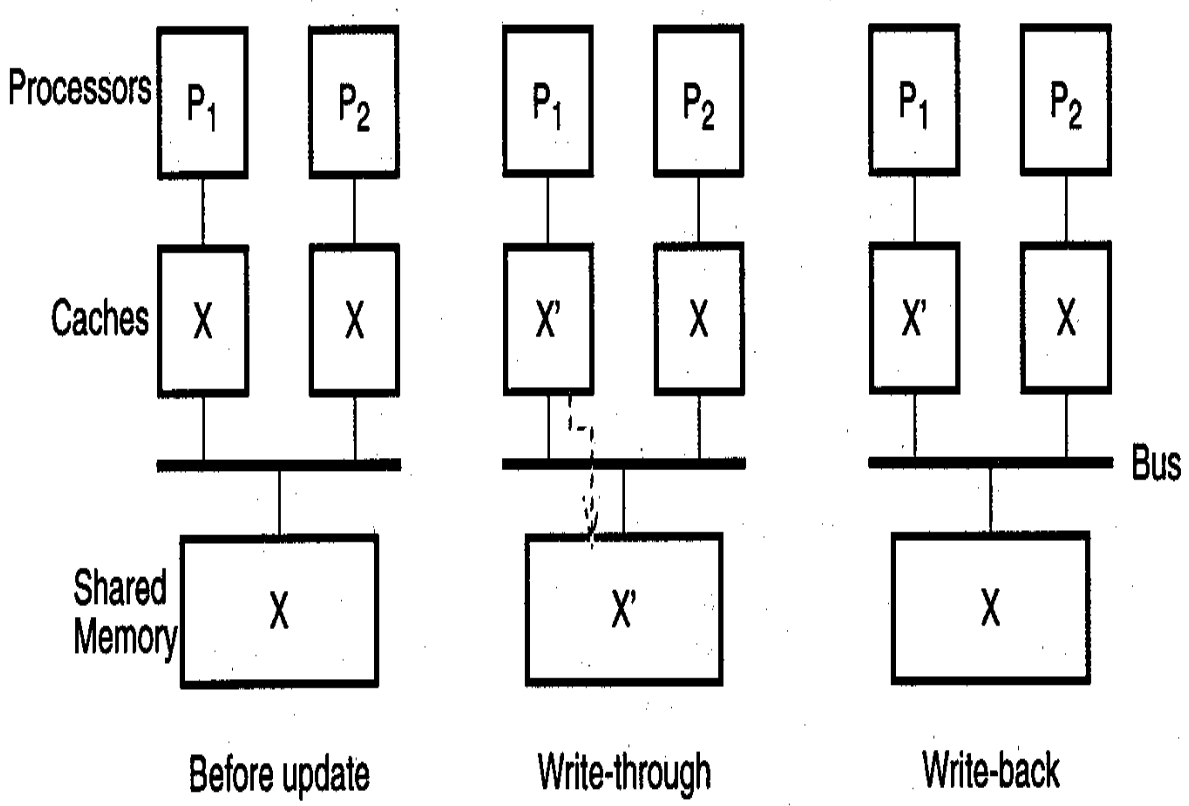


Figure 24 Cache Coherence [71]

In fig. 6, we have a common bus-based shared memory design. If both processors read the same piece of data X, each processor's cache will contain a copy of X. The problem of cache coherency comes up when one or both of the processors write to X. Whether write-through or write-back cache is used, a problem arises when P_1 or P_2 writes to X. With write-through, if P_1 writes to X, X' will be stored in memory and will be consistent with P_1's cache. P_2 however will still have the previous value of X in its cache, so it will have the wrong value the next time it reads X. With Write Back the same cache inconsistency problem occurs, except that the shared memory is not updated immediately. There are several possible solutions to this problem. On a write to X, X can be invalidated in all the other caches, X could be updated in all the other caches, X could be written straight to the shared memory (which is supported by Altera), or it could be arranged in software.

3.2 Mutual Exclusion

Regardless of which shared memory structure is used, multiprocessor systems require mutual exclusion for shared memory. Even in a system where there are no cache inconsistencies, if two processors write to the same shared data at the same time, one processor can overwrite the other's work. Mutual exclusion can be provided by LOCK-UNLOCK commands around critical sections. Critical sections are sections of code that must be executed sequentially (i.e. access shared data). The LOCK-UNLOCK commands are a set of operations that execute atomically (as defined in section 1.2) and guarantee mutual exclusion. A software lock might look something like this:

```
lock:       ld      register, location   /* copy location to register*/
                    cmp     location, #0        /* compare with 0 */
                    bnz     lock                /* if not 0, try again */
                    st      location, #1        /* store 1 to mark it locked */
                    ret                         /* return control to caller */

unlock:     st      location, #0               /* write 0 to location */
                    ret                         /* return control to caller*/
```

Figure 25 Simple Software Lock [70]

The problem with this software implementation is that the testing and setting of the variable is not an atomic operation. This means that two processes could possibly test at the same time and both enter the critical section. This must be solved in hardware by providing an instruction which can atomically test a value, change it, and write it back to memory.

3.3 Atomic Exchange Instruction

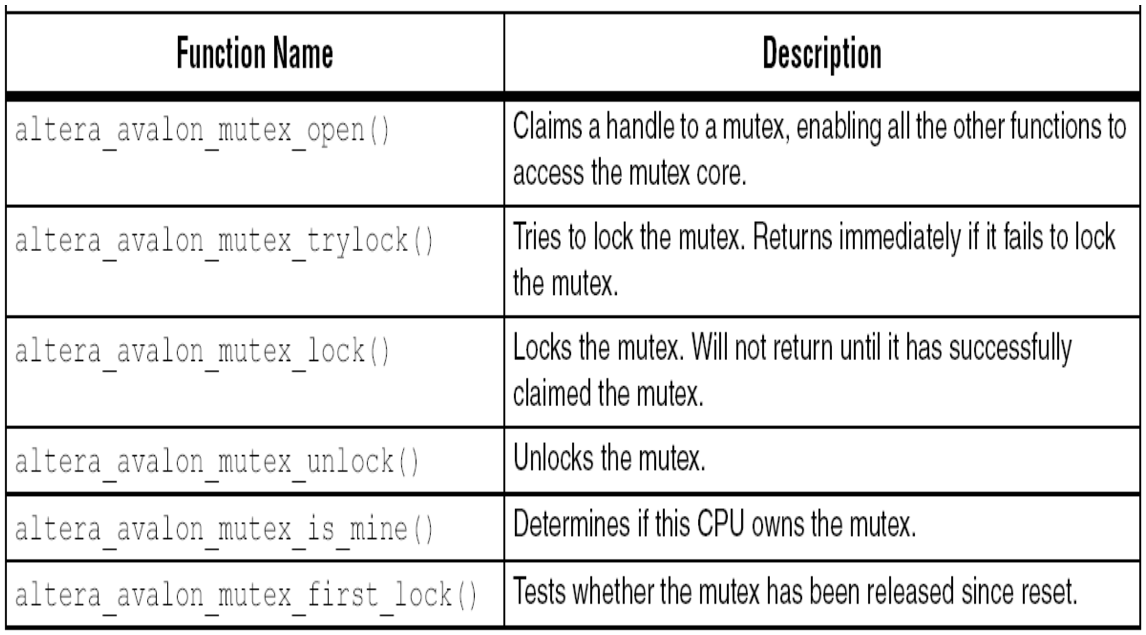
This instruction must read a specified value into a register, set if to another value or not depending on its initial value. Swap or Test and Set type instructions can be used to implement this. For example, an instruction: 'test&set location' would load the value in location into a register and store 1 at location. If the original value (the value in the register) is 0 the test is successful and the calling processor could enter its critical section. This can be used to implement a valid version of the lock and unlock functions shown in fig. 8:

```
lock:           t&s     register, location
                        bnz     lock            /* if not 0, try again */
                        ret                     /* return control to caller */
unlock:   st    location, #0    /* write 0 to location */
                        ret                     /* return control to caller */
```

Figure 26 Test&Set Lock [70]

3.4 Mutual Exclusion in Altera Devices

A mutex (stands for mutual exclusion) is a construct that allows multiple processors, or threads, to coordinate mutual exclusion. Altera includes a core for a hardware mutex in its libraries (see Table 2 for function calls). It is not required to access shared memory, however it's an easy and efficient way to provide mutual exclusion, more so than other methods such as using semaphores. Semaphores are simply variables which can take on two or more states to indicate whether a resource is locked or unlocked. They are modified only using atomic operations such as wait() and notify(). This is a good solution as well, but it is faster to do it in hardware.



Function Name	Description
`altera_avalon_mutex_open()`	Claims a handle to a mutex, enabling all the other functions to access the mutex core.
`altera_avalon_mutex_trylock()`	Tries to lock the mutex. Returns immediately if it fails to lock the mutex.
`altera_avalon_mutex_lock()`	Locks the mutex. Will not return until it has successfully claimed the mutex.
`altera_avalon_mutex_unlock()`	Unlocks the mutex.
`altera_avalon_mutex_is_mine()`	Determines if this CPU owns the mutex.
`altera_avalon_mutex_first_lock()`	Tests whether the mutex has been released since reset.

Table 2 Altera Hardware Mutex Function Calls [72]

Altera_avalon_mutex_open() gets a handle on the device from the Hardware Abstraction Layer (HAL). Once this is done, the other functions can be used to operate the mutex. Each shared resource can have its own mutex, though it’s not strictly necessary. This can be used to implement mutual exclusion, for example:

```
#include <altera_avalon_mutex.h>

/* get the mutex device handle */
alt_mutex_dev* mutex = altera_avalon_mutex_open( “/dev/mutex” );
/* acquire the mutex, setting the value to one */
altera_avalon_mutex_lock( mutex, 1 );
/*
* Access a shared resource here.
*/
/* release the lock */
altera_avalon_mutex_unlock( mutex );
```

Figure 9 Using the Hardware Mutex

First the pointer to the mutex is declared using altera_avalon_mutex_open(), then the mutex is locked and unlocked by passing the pointer to the lock and unlock functions.

Summary

This lecture has presented different shared memory system architectures and issues surrounding their implementation. The main concepts are shared memory architectures, cache coherence, and mutual exclusion.

Shared memory systems consist of a shared memory section which can be directly referenced by any processor. Cache and main memory may be shared by all processors (shared cache), many memories may be shared between many processors (dancehall), each processor may have its own cache but share the main memory (bus-based shared memory), or each processor may have a memory and a cache (distributed memory).

In shared memory systems, steps must be taken to ensure that there are no cache inconsistencies (situations where a value in cache is invalid because it has been changed by another processor). There are several schemes to remedy this.

When dealing with shared memory, it is important to maintain synchronicity between sections of code accessing the same memory locations to ensure there is mutual exclusion. This is accomplished through locking mechanisms called mutexes. These can be implemented with atomic operations or as a hardware component.

Paper 1 Transactional Memory Coherence and Consistency

Paper 2 Performance Measurement and Modeling to Evaluate Various Effects on a Shared Memory Multiprocessor

Lecture #11a: OpenMP

Course: Computer Architecture III
Lecturer: Miodrag Bolic
Scribe: Roxane Beaudoin - 2957385
Steve Bandele - 2396231
Date: December 1, 2006 and October 18, 2006

Introduction

Parallel computing includes several popular parallel programming models that are in use today, each of which has different possible implementations. OpenMP is one such implementation that addresses multiple parallel programming models, making it a hybrid programming model[74]. OpenMP is an Application Program Interface (API) that is based on a set of compiler directives and library routines. It is a manual form of specifying parallelism and does not provide any automated parallelism.

This paper will discuss OpenMP and how it provides support for different parallel programming models. It will provide an overview of the language by showing and explaining several examples which are included in the appendices.

Explanation

1.1 What is OpenMP?

OpenMP is an API that is used for writing multi-threaded applications. It combines three different parallel programming models, which will be specified below. It starts with the serial code and relies on incremental parallelism, adding parallel constructs. The programmer is responsible for determining the parallelism by specify to the compiler what portions of the code to make parallel, there is no automated parallelism.

The API consists of three main components. The first are compiler directives, which provide a recommendation to the compiler and not a command. These directives will indicate to the compiler that a specific portion of the code is suitable for parallel execution, but it is ultimately up to the compiler to partition the code into parallel programs. The other components are runtime library routines, which are specified in the openmp.h file for C programming, and several environment variables.

An important feature of OpenMP is its portability. It is designed for multi-platforms and OpenMP code can be written for many Windows NT and UNIX platforms using C/C++ or Fortran. Fortran might not seem to be very popular, however it is frequently used in the parallel programming community.

OpenMP stands for Open specifications for Multi Processing. It is standardized so it is defined by many different hardware and software companies from the industry, as well as contributors from government and academia.

1.2 History

OpenMP was released in 1997. The most current version for C/C++ and Fortran is 2.5 which was released in May 2005. There are many companies involved in the specification and use of OpenMP, including those that are listed below.

Architecture Review Board
Compaq / Digital
HP Company
Intel Corporation
IBM
Sun Microsystems, Inc.
Software Vendors (that write parallel code)
Absoft Corporation
Edinburgh Portable Compilers
GENIAS Software GmBH
Myrias Computer Technologies, Inc.
Application Vendors (that use OpenMP)
ADINA R&D, Inc.
ANSYS, Inc.
Dash Associates
Oxford Molecular Group PLC

1.3 Goals

There are four major goals of OpenMP.
To provide a standard among the shared memory architectures. Before OpenMP, every manufacturer with a shared memory architecture had their own programming language.
To establish a limited set of directives for parallel programming. With OpenMP, parallel programs can be written with only a few, simple directives.
To provide an easy to use method of implementing parallelism.
To provide a portable method of implementing parallelism.

2.0 Programming Models

OpenMP combines the shared memory model, the thread based model, and the data parallel model. It addresses the thread based model because we start with a master thread, executing it sequentially, until we reach a portion of the code that can be executed in parallel. At this point we can have a fork and execute the parallel portion using several parallel threads, including the master thread. After completing the parallel portion we have a join and return to sequential execution with just the master thread. This is shown in Figure 1.

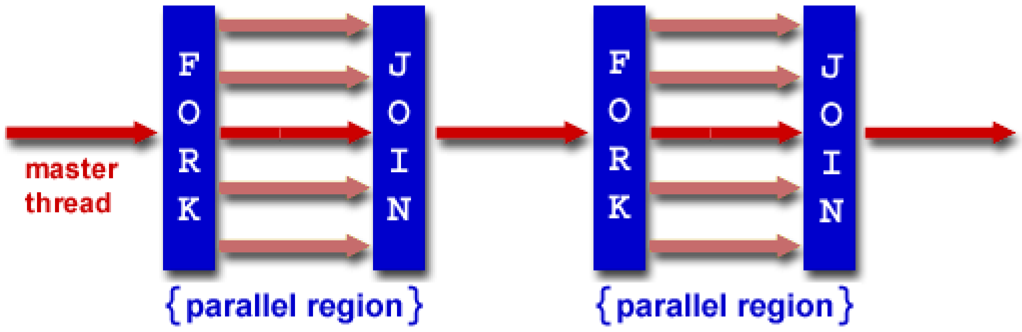


Figure 1: OpenMP using the thread-based (fork-join) model in [75]

There threads use shared memory for communication, therefore OpenMP is a shared memory programming model. If this fork occurs in front of a loop, then the loop is divided into several parallel loops; therefore OpenMP is also a data parallel programming model. The important thing to note is that we have an explicit parallelism, therefore we have to tell the compiler what is parallel and what is not, resulting in manual parallelism.

2.1 Parallel Region Example

The parallel region example included in Appendix 1 is described below.
Header file omp.h is included.
Main is specified in C.
#pragma omp parallel is a compiler directive which indicates to execute the part of code within the {} in parallel.
Omp_get_thread_num() is a command from the OpenMP library which allows every thread to get its own thread number and this thread number is written into the integer *tid* (thread identifier). This integer is private, which means that each thread will have its own separate variable *tid*. Each thread is using shared memory to communicate but each thread can also have its own private data.
Each thread prints "Hello World from thread" followed by the number of the thread.
The master thread always has an identifier of 0. It can get the total number of threads using the command *omp_get_num_threads()*. The master thread gets the total number of threads and then prints this number. This is done sequentially and only by the master thread. Even though this part is performed sequentially, it still had to be completed within the parallel section of the code. This is because the end of the parallel section is called a barrier, and at this point all of the created threads are destroyed. Therefore, if this sequential part was moved outside of the parallel region, it would provide a different result.
Note: The default number of threads is equal to the number of processors. This example did not specify how many threads, therefore it will be relying on compiler of the system to provide this number.

2.2 Work-sharing Constructs

A work-sharing construct distributes the execution of the associated region among a team of threads that encounters it. A work-sharing region is usually bound to an active parallel region in order for the work-sharing region to execute in parallel. If execution encounters a work-sharing region in the sequential part, it is executed by the initial (master) thread. A work-sharing construct does not launch new threads, and a work-sharing region has No barrier on entry. However, an implied barrier exists at the end of the work-sharing Region, unless a nowait clause is specified. In the presence of nowait, threads that finish early may proceed straight to the next instructions following the work-sharing region without waiting for the other members of the team to finish the work-sharing region.

2.2.1 Types of Work-Sharing Constructs

The following work-sharing constructs are defined by OpenMP.

• Do/For construct
• Sections construct
• Single construct

Work-sharing constructs are constrained by the fact that each working region must be consistent and encountered by all the threads at the same time or none at all.

2.2.1.1 Do/For construct

This construct specifies that the iterations of the associated do/for loop will be executed in parallel. The iterations are distributed amongst the threads that already exist in the team executing the parallel region .The syntax for the construct is as follows

`#pragma omp for` *[clause[[,] clause] ...] new-line*

The clause could be one of the following. private(*list*), firstprivate (*list*), Nowait lastprivate(*list*), reduction(*operator*: *list*),ordered, schedule (*kind[, chunk]*). An example C/C++ directive example is shown in Appendix 2.

SCHEDULE: Describes how iterations of the loop are divided among the threads in the team. The schedule could be static, dynamic, guided or runtime, but the default is implementation dependent.
Static – Iterations are divided into chunks of size *chunk*, and the chunks are statically assigned to threads in the team in a round-robin fashion in the order of the thread number. The last chunk to be assigned to one of the thread may have a smaller number of iterations. When no *chunk* is specified, the iteration space is divided into chunks which are approximately equal in size, and each thread is assigned at most one.
Dynamic - Loop iterations are divided into pieces of size *chunk*, and dynamically scheduled among the threads as they request them. The thread executes the chunk of iterations, then requests another chunk, until no chunks remain to be assigned. When no *chunk* is specified, it defaults to 1.
Guided - For a chunk size of 1, the size of each chunk is proportional to the number of unassigned iterations divided by the number of threads, decreasing to 1. For a chunk size

with value k (greater than 1), the size of each chunk is determined in the same way with the restriction that the chunks do not contain fewer than k iterations (except for the last chunk to be assigned, which may have fewer than k iterations). The default chunk size is 1.
Runtime - The scheduling decision is deferred until runtime by the environment variable OMP_SCHEDULE. It is illegal to specify a chunk size for this clause. the schedule and chunk size are taken from the *run-sched-var* control variable.

NO WAIT / nowait: If specified, then threads do not synchronize at the end of the parallel loop. Threads that finish executing can continue executing the rest of the code without having to wait for other threads.

2.2.1.2 SECTION CONSTRUCT

The sections construct is a noniterative work-sharing construct that contains a set of Structured blocks that are to be divided and executed by the threads in a team.
Each structured block is executed once by one of the threads in the team. Different sections are executed by different threads. If there are more threads than sections, then some threads would not execute any section. If there are more section than threads, the implementation defines how extra sections are executed. It is illegal to branch into or out of a section being executed by the thread. The syntax is given by

`#pragma omp sections` *[clause[[,] clause] ...] new-line*

Where clause could be private(*list*), firstprivate(*list*), lastprivate(*list*), reduction(*operator*: *list*) nowait. A sample C/C++ code is given in Appendix 3.

2.2.1.3 SINGLE CONSTRUCT

The single construct specifies that the associated block is executed by only one thread in the team which may not be necessarily the master thread. The other threads in the team do not execute the block, and wait at the predefined barrier at the end of single construct, unless a nowait clause is specified. May be useful when dealing with sections of code that are not thread safe (such as I/O).

3.0 Combined Parallel Work-sharing Constructs

Combined parallel work-sharing constructs are shortcuts for specifying a work-sharing Construct nested immediately inside a parallel construct. The semantics of these Directives are identical to that of explicitly specifying a parallel construct containing one work-sharing construct and no other statements. The combined parallel work-sharing constructs allow certain clauses which are permitted on both parallel constructs and on work-sharing constructs. The following sections describe the combined parallel work-sharing constructs:
• The parallel loop(do/for) construct.
• The parallel sections construct.

Summary

3.1 PARALLEL DO / parallel for Directive

The syntax is given below. The *clause* can be any of the clauses accepted by the parallel or for directives, except the nowait clause, with identical meanings and restrictions. Iterations of the DO/for loop will be distributed in equal sized blocks to each thread in the team an can be implemented on single processors, multiple data.(SPMD). A sample C/C++ code is given in Appendix 4.

`pragma omp parallel for` *[clause[[,] clause] ...] new-line for-loop*

3.2 PARALLEL SECTIONS Directive

The PARALLEL SECTIONS directive specifies a parallel region containing a single SECTIONS directive. The single SECTIONS directive must follow immediately as the very next statement. Usually implemented on Multiple processors, multiple data(MPMD) .The format is given below.

```
#pragma omp parallel sections [clause ...]  newline
                    default (shared | none)
                    shared (list)
                    private (list)
                    firstprivate (list)
                    lastprivate (list)
                    reduction (operator: list)
                    copyin (list)
                    ordered

  structured_block
```

4.0 Synchronization Constructs

Can be viewed as a mutual exclusion mechanism where only one processor is given the access to one particular data at a time keeping memory consistent. They include

Master Directive – specifies the region that should be executed only by the master processor and skipped by others. Branching into and out of this region is not allowed. The syntax is

```
                #pragma omp master newline
                structured_block
```

Critical Directive - restricts the execution of a specific block to just one thread. Other threads trying to access this section would block until the thread is done. All critical sections must have a name .Those without a name are given the same unspecified name .The C/C++ syntax is given below and sample code given in Appendix 5.

```
#pragma omp critical [ name ]  newline
structured_block
```

Barrier directive – Is used to synchronize threads since all of the threads of the team executing the binding parallel region must execute the barrier region before any are allowed to continue execution beyond the barrier. The syntax is given below

```
#pragma omp barrier  newline
```

Atomic Directive – Provides a minimal critical section by ensuring that a specific storage location is updated atomically, rather than exposing it to the possibility of multiple, simultaneous writing threads. The syntax is given below. Atomic regions enforce exclusive access with respect to other atomic regions that update the same storage location among all the threads in the program without regard to the team(s) to which the threads belong.

```
#pragma omp atomic  newline

 statement_expression
```

Expression could be ne f the following statements, ++X ,X++, X--

5.0 Run-Time Library Routines

OMP_SET_NUM_THREADS - Sets the number of threads that will be used in the next parallel region. Must be a positive integer.

OMP_GET_NUM_THREADS - Returns the number of threads that are currently in the team executing the parallel region from which it is called.

OMP_GET_MAX_THREADS - Returns the maximum value that can be returned by a call to the OMP_GET_NUM_THREADS function.

OMP_GET_THREAD_NUM - Returns the thread number of the thread, within the team, making this call. This number will be between 0 anOMP_GET_NUM_THREADS-1. The master thread of the team is thread 0.

OMP_GET_NUM_PROCS - Returns the number of processors that are available to the program.

OMP_IN_PARALLEL - May be called to determine if the section of code which is executing is parallel or not.

OMP_SET_DYNAMIC - Enables or disables dynamic adjustment (by the run time system) of the number of threads available for execution of parallel regions.

OMP_GET_DYNAMIC - Used to determine if dynamic thread adjustment is enabled or not.

OMP_SET_NESTED - Used to enable or disable nested parallelism.

OMP_GET_NESTED - Used to determine if nested parallelism is enabled or not.

OMP_INIT_LOCK - This subroutine initializes a lock associated with the lock variable.

OMP_DESTROY_LOCK- This subroutine disassociates the given lock variable from any locks.

OMP_SET_LOCK - This subroutine forces the executing thread to wait until the specified lock is available. A thread is granted ownership of a lock when it becomes available.

OMP_UNSET_LOCK - This subroutine releases the lock from the executing subroutine.

OMP_TEST_LOCK- This subroutine attempts to set a lock, but does not block if the lock is unavailable

6.0 Appendices

6.1 Appendix I

Parallel region example sample C code

```
#include <omp.h>

main ()  {

int nthreads, tid;

/* Fork a team of threads giving them their own copies of variables */
#pragma omp parallel private(tid)
  {

  /* Obtain and print thread id */
  tid = omp_get_thread_num();
  printf("Hello World from thread = %d\n", tid);

  /* Only master thread does this */
  if (tid == 0)
    {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }

  }  /* All threads join master thread and terminate */

}
```

6.2 Appendix II

Sample C code for DO/For Construct

```
#include <omp.h>
#define CHUNKSIZE 100
#define N     1000
main ()
{
int i, chunk;
float a[N], b[N], c[N];
/* Some initializations */
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;
chunk = CHUNKSIZE;
#pragma omp parallel shared(a,b,c,chunk) private(i)
  {
```

```
#pragma omp for schedule(dynamic,chunk) nowait
  for (i=0; i < N; i++)
    c[i] = a[i] + b[i];
}  /* end of parallel section */
}
```

6.3 Appendix III

C / C++ - sections Directive Example

```
#include <omp.h>
#define N     1000
main ()
{
int i;
float a[N], b[N], c[N];
/* Some initializations */
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;
#pragma omp parallel shared(a,b,c) private(i)
  {
#pragma omp sections nowait
    {
#pragma omp section
    for (i=0; i < N/2; i++)
      c[i] = a[i] + b[i];
#pragma omp section
    for (i=N/2; i < N; i++)
      c[i] = a[i] + b[i];
}  /* end of sections */
  }  /* end of parallel section */
}
```

6.4 Appendix IV

C / C++ - parallel for Directive Example

```
#include <omp.h>
#define N       1000
#define CHUNKSIZE   100
main ()  {
int i, chunk;
float a[N], b[N], c[N];
/* Some initializations */
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;
chunk = CHUNKSIZE;
#pragma omp parallel for \
   shared(a,b,c,chunk) private(i) \
   schedule(static,chunk)
  for (i=0; i < n; i++)
    c[i] = a[i] + b[i];
}
```

6.5 Appendix v

C / C++ - critical Directive Example

```
#include <omp.h>
main()
{
int x;
x = 0;
#pragma omp parallel shared(x)
  {
  #pragma omp critical
  x = x + 1;
  }  /* end of parallel section */
}
```

Paper 1 A Practical Open MP Compiler for System on Chips

Paper 2 OpenMP Implementation and Performance on Embedded Renesas M32R Chip Multiprocessor

Lecture #11b: Shared memory systems and OpenMP

Course: Computer Architecture III
Lecturer: Miodrag Bolic
Scribe(s): Reinaldo Gift , Parvin Kamarujaman
Date: October 17, 2006

Introduction:

The lectures in question addressed the question of "how is shared memory used in parallel processors?" The first lecture outlines the problems with respect to the shared memory, the cache and the processors which interact together. Research papers on the topic in question have been cited so as to better explain the issues addressed in the lectures. One namely: Some Solutions for Critical Problems In The Theory and Practice of Distributed Shared Memory: New Ideas to Analyse by Veljko Milutinovic(http://tab.computer.org/tcca/news/sept96/dsmideas.pdf).
Another Research paper used was on the topic: Synchronization Algorithms for Shared-Memory Mulitprocessors byGary Graunke and ShreeKant Thakkar.
http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=55501

Explanation

SHARED MEMORY:

Shared Memory is memory for which any processor in the system can access that memory space. Accessing memory is costly in terms of time and as a result, Memory may be physically distributed among processors. (a small record of parts of the previously accessed memory). When a processor needs a certain data, it first checks its cache for the data. If it does not find the data, the processor then checks either another processor's cache or the memory for the required data.The problem answered in the lecture was how a system handles data coherency of the individual caches and the shared memory. There are a number of protocols that have been categorized as either snooping or directory based to deal with this problem.

SNOOPING PROTOCOL:

Snooping Protocols are based on monitoring bus activity and carrying out commands depending on that bus activity. The are four states that might occur when accessing memory: Read-Hit, Read-Miss, Write-Hit and Write-Miss. A Read-Hit occurs when a value to be found in memory is found in the processors cache. A Read-Miss occurs when a value to be found in memory is not found in the processors cache. A Write hit occurs when the location in memory to be referenced is listed in the cache as a latest copy. A

Write-miss occurs when the location in memory to be referenced is not listed in the cache or is not listed in the cache as a latest copy(but rather an older copy).

Write Through: This uses two flags for the caches data- INV and VALID. On a read hit, a local copy from cache is used. On a read miss, a copy is fetched from global memory and the cache's copy's flag is set to VALID. On a write hit, the write is performed locally to the cache as well as the global memory, and all other cache copies are set to an INVALID state. On a write miss, a copy from global memory is gotten. The local cache as well as the global memory is then written to, while all other cache copies are updated to an INVALID state.
Write Back: This uses three flags for the caches data INV, RO and RW. On a read hit a local cache copy is used. On a read miss, if none of the cache copies are in the RW state, then a copy is fetched from global memory. If a cache copy does exist in RW state, then a copy of that is used. At this point the global memory is also updated with the RW copy and both caches are set to RO state. On a write hit, if the local cache copy is RW, then the copy is simply written over. If the local cache copy is RO then this copy is set to RW and all other cache copies are set to INV. On a write miss the write is also performed locally and its state is set to RW. At this point all other cache copies are set to INV.

Synchronization:
Components of a Synchronization event
There are three major components of a synchronization event:
Acquire method is a method by which a process tries to acquire the right to the synchronization (ie. To enter the critical section or proceed past the event synchronization)
Waiting algorithm is a method by which a process waits for a synchronization to become available. For example if a process tries to acquire a lock but the lock is not free, or to proceed past an event but the event has not occurred yet.
Release method is a method for a process arriving at a barrier to release the waiting process, or a method for the last process arriving at a barrier to release the waiting processes, or a method for notifying a process waiting at a point-to-point event that the event has occured.
There are two main types of waiting : *Busy-waiting* and *Blocking*. *Busy-waiting* is when the process spins in a loop that repeatedly tests for a variable to change its value. A release of the synchronization event by another processor changes the value of the variable, allowing the waiting process to proceed. *Blocking* is when the process does not spin but simply suspends itself and releases the processor if it finds that it needs to wait. When the release it was waiting for occurs, it will be awakened and run again. Busy-waiting avoids the cost of suspension but consumes the processor and cache bandwidth while waiting. Blocking is powerful than busy-waiting because if the processor or thread that is being waited upon is not allowed to run, the busy-wait will never end. Busy waiting is better when the waiting period is short. But blocking is better if the waiting period is long.

Mutual exclusion algorithms are used in concurrent programming to avoid the simultaneous use of un-shareable resources by pieces of computer code called critical sections. A **mutex** is also a common name for a program object that negotiates mutual exclusion among threads, also called a lock.

Simple Software Lock

```
lock:   ld  register, location /* copy location to register*/
        cmp register, #0       /* compare with 0*/
        bnz lock               /* if not 0, try again */
        st   location, #1      /* store 1 into location to mark it locke d */
        ret                    /* return control to caller of lock */
```

and

```
unlock:st location, #0       /*write 0 to location*/
        ret                  /* return control to caller */
```

For the acquire method, a process trying to obtain a lock much check that that lock is free and, if it is, then claim ownership of the lock. The state of the lock can be stored in a binary variable, with 0 representing free and 1 representing busy. A process trying to obtain the lock should check if the variable is 0 and if so set it to 1, thus marking the lock busy; if the variable is 1(lock is busy), then it should wait for the variable to turn to 0 using the waiting algorithm. And unlock algorithm should simply set the variable to 0.

Problem: The problem with this lock, which is supposed to provide atomicity for the critical section that follows it, is that it needs atomicity in its own implementation. Suppose that the lock variable was initially set to 0 and two processes P0 and P1 execute the above assembly code implementation of the lock operation. Process P0 reads the value of the lock variable as 0 and thinks it is free, so it proceeds past the branch instruction. Its next step is to set the variable to 1, marking the lock as busy, but therefore it can do this, process P1 reads the variable as 0, thinks the lock is free, and passes the branch instruction too. We now have two processes simultaneously proceeding past the lock and entering the same critical section, which is exactly what the lock was meant to avoid.

Solution: An efficient, solution to the lock problem is to support an atomic *read-modify-write* instruction in the processor's instruction set. A typical solution is to have an atomic exchange instruction.

Atomic exchange Instruction: A value at memory location specified by the instruction is read into a register, and another value is stored into the location, all in an atomic operation with no other accesses to that location allowed to intervene. Many variants of this operation exist with varying degrees of flexibility in the nature of the value that can be stored. An example that works for mutual exclusion is an atomic test&set instruction.

Simple test&set Lock

```
lock: t&s   register, location
      bnz   lock              /*if not 0, try again */
      ret                     /*return control to caller */
unlock:st location, #0        /*write 0 to location*/
        ret                   /*return control to caller
```

Summary

In this case, the value in the memory location is read into a specified register, and the constant 1is stored into the location atomically. The success of the test&set is determined by examining the value in the register. If it is 0, the test&set was successful. It it is 1, it was unsuccessful; the value written to memory by the test&set instruction is the same as was already there, so no harm is done.
The lock implementation keeps trying to acquire the lock using test&set instructions until the test&set leaves zero in the register, indicating that the lock was free when tested(in which case the test&set has set the lock variable to 1, thus acquiring it). The unlock construct simply sets the locatin associated with the lock to 0, indicating that the lock is now free and enabling a subsequent lock operation by any process to succeed. A simple mutual exclusion construct has been implemented in software, relying on the fact that the architecture supports an atomic test&set instruction.

In the second research paper "Synchronization Algorithms for Shared-Memory Multiprocessors", a performance evaluation of the Symmetry multiprocessor system was done. For highly contested locks, like the ones in some parallel application, the synchronization mechanism does not perform well. In the paper many software synchronization mechanisms were developed and evaluated, using a hardware monitor. The mechanisms were to reduce contention for the locks on the symmetry multiprocessor system. Even when changes were made to the hardware synchronization mechanisms to Improve support for highly contested locks, the mechanisms remained useful. In the research paper, symmetry architected is described in detail and a number of lock algorithms and their use of hardware resources are examined in detail. In the end the performance of each lock mechanism is explained in terms of the program and total system performance.

This research paper explains some Lock algorithms in detail like Simple lock, Snooping lock, Collision avoidance locks, Tournament Locks, Queing Lock etc. This research paper analyzes the performance of all the lock algorithm including test and set lock. The simple lock algorithm and the Test and set lock algorithm was covered in the Monday's lecture. From this research paper we understand the performance of Simple Lock algorithm a Test&set algorithm with comparison to other efficient algorithms.

For the simple lock that was covered in class, we can learn from the research paper that the memory requirement is very minimal. We can also learn from the research paper that Simple lock floods the communication network and are unreasonable when the number of processors become large. The collision avoidance algorithm can adjust to the growing number of processes. Queue lock remains unaffected by the number of processes. There are also other lock algorithms for which the memory requirements are minimal and they are snooping and collision lock algorithms. We learn that Tournament Lock algorithm requires multiple cache blocks to perform well. Queue locks require memory in proportion to the number of processes. When the critical section is short snooping lock has a worst-case behavior.

When a miss occurs, more time is taken to retrieve the data. In Mr. Milutinovic's paper, he suggests that a calculation of probability of a cache miss from past data be done. If the miss probability is above a certain number or too high, then replace/refresh the cache value. This way, when the cache block is referenced it is less likely to miss, thereby likely speeding up the system. Mr. Milutinovic calls this Direct Cache Injection. Mr. Milutinovic in his paper divides data into temporal and spatial and suggests that dealing with each differently will improve processing speed. More directly relating to the lecture, he suggests dealing with memory as temporal and to thus use an entry consistency model as opposed to lazy release. Spatial Data will be dealt with using lazy release. His paper continues beyond the scope of the lecture.

Lecture # 12 a : Parallel Programming Models

Lecturer :	Miodrag Bolic
Scribes :	Arulvany Ambalavanan, Robin Tropper
Lecture date :	October 18, 2006

Introduction

What is Parallel Programming? [81]
In contrast to traditional sequential programming, which can be read as a series of instructions to be executed one after the other as if reading a script, Parallel computing is the simultaneous use of multiple computer resources to solve a computational problem. Parallel Programming is the process of organizing tasks so they can be executed concurrently.

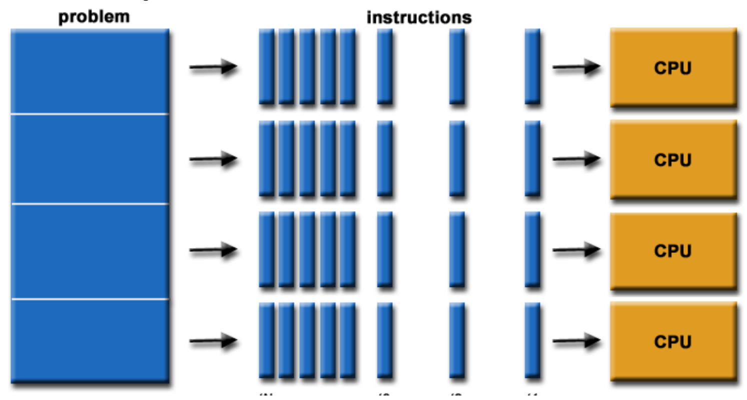


Figure 27: single problem broken down to tasks for parallel processors [81].
Several problems can be addressed by this process where several entities show independent behaviour, yet may interact. For example in the natural world, planetary and galactic orbits, weather and oceanic patterns and so on can be grouped into a cohesive whole, yet the behaviour of their parts (e.g. the orbit of a single planet) can be computed alone. In fact, natural science and the simulation of their mathematical models is the traditional domain of parallel computing.

Explanation

Why use it? [81]
Commercial applications are increasingly a driving factor in the development of faster computers. The increasing complexity and sophistication of the programs required is a driving factor in developing parallel computing as a way to improve performance. Parallel programming techniques are therefore necessary to make best use of the underlying technology.
The advantages to parallel computing are manifold. Parallel computing provides concurrency of tasks, be they similar or different. When a parallel solution is possible, several tasks executing at the same time will naturally reduce the time to completion. The added benefit of reduced time is the possibility of solving larger problems. Parallel computing can make it possible to use non-local resources: a system can be built with several less expensive components rather than one more expensive system. [81]

There are also physical and practical limits to building ever faster sequential computers. Transmission speeds on communication lines are an upper bound to communications between processors. For example, there is a transmission limit on copper wires and, notwithstanding the transport media, the speed of light is a limit. Increasing proximity of processing elements is therefore important which indicates that sequential computing can be improved by miniaturisation. But there are ultimately limits there also. Finally, increasing the performance of any system is always costly. A business analysis of a required system will therefore impose constraints. [81]

Terminology:

Architectural concepts (summarized from [81]):

von Neumann Architecture :

a 'stored program' and the required data

are 'fetched' for sequential execution of the instruction set by the CPU

the results of which are stored again in memory

Flynn's Classical Taxonomy refers to how data and instruction sets can (or cannot) be distributed. It refers to what can be done in a single clock cycle :

SISD: single-instruction, single-data (a serial, non-parallel computer)[2];

SIMD: several processors work on a single piece of data – best suited to problems with high degree of regularity (e.g. image processing);

MISD: multiple instruction, single data (rarely used).

MIMD: multiple instruction, multiple data – may be synchronous or asynchronous.

Terminology [81]:

Task: A logically discrete section of computational work. A task is typically a program or program-like set of instructions that is executed by a processor.

Parallel Task: A task that can be executed by multiple processors safely (yields correct results)

Serial Execution: Execution of a program sequentially, one statement at a time. In the simplest sense, this is what happens on a one processor machine. However, virtually all parallel tasks will have sections of a parallel program that must be executed serially.

Parallel Execution: Execution of a program by more than one task, with each task being able to execute the same or different statement at the same moment in time.

Shared Memory: From a strictly hardware point of view, describes a computer architecture where all processors have direct (usually bus based) access to common physical memory. In a programming sense, it describes a model where parallel tasks all have the same "picture" of memory and can directly address and access the same logical memory locations regardless of where the physical memory actually exists.

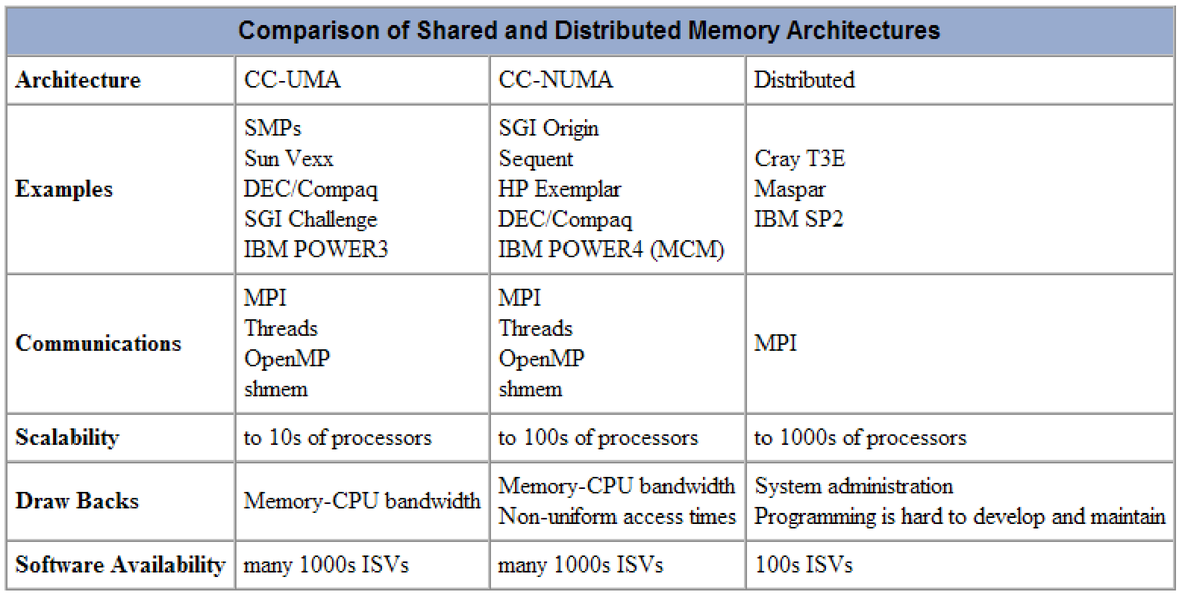
Distributed Memory: In hardware, refers to network based memory access for physical memory that is not common. As a programming model, tasks can only logically "see" local machine memory and must use communications to access memory on other machines where other tasks are executing.

Communications: Parallel tasks typically need to exchange data. There are several ways this can be accomplished, such as through a shared memory bus or over a network, however the actual event of data exchange is commonly referred to as communications regardless of the method employed.

Synchronization: The coordination of parallel tasks in real time, very often associated with communications. Often implemented by establishing a synchronization point within an application where a task may not proceed further until another task(s) reaches the same or logically equivalent point. Synchronization usually involves waiting by at least one task, and can therefore cause a parallel application's wall clock execution time to increase.
Granularity: In parallel computing, granularity is a qualitative measure of the ratio of computation to communication.
Coarse: relatively large amounts of computational work are done between communication events
Fine: relatively small amounts of computational work are done between communication events
Observed Speedup: Observed speedup of a code which has been parallelized, defined as: One of the simplest and most widely used indicators for a parallel program's performance.
Parallel Overhead: The amount of time required to coordinate parallel tasks, as opposed to doing useful work. Parallel overhead can include factors such as: task start-up time, synchronizations, data communications, task termination time, software overhead imposed by parallel compilers, libraries, tools, operating system, etc.
Massively Parallel: Refers to the hardware that comprises a given parallel system - having many processors. The meaning of many keeps increasing, but currently BG/L pushes this number to 6 digits.
Scalability: Refers to a parallel system's (hardware and/or software) ability to demonstrate a proportionate increase in parallel speedup with the addition of more processors. Factors that contribute to scalability include: Hardware - particularly memory-cpu bandwidths and network communications, application algorithm, parallel overhead related, characteristics of your specific application and coding .

Parallel Computer Memory Architectures

The crux of parallelism is contention for resources (memory, I/O devices, busses, etc.). The most important of these are memory and communication lines because both instructions and data are found on memory and must be transported to the various processors. We find 3 typical solutions: shared memory, distributed memory and hybrid systems. [81]



Comparison of Shared and Distributed Memory Architectures			
Architecture	CC-UMA	CC-NUMA	Distributed
Examples	SMPs Sun Vexx DEC/Compaq SGI Challenge IBM POWER3	SGI Origin Sequent HP Exemplar DEC/Compaq IBM POWER4 (MCM)	Cray T3E Maspar IBM SP2
Communications	MPI Threads OpenMP shmem	MPI Threads OpenMP shmem	MPI
Scalability	to 10s of processors	to 100s of processors	to 1000s of processors
Draw Backs	Memory-CPU bandwidth	Memory-CPU bandwidth Non-uniform access times	System administration Programming is hard to develop and maintain
Software Availability	many 1000s ISVs	many 1000s ISVs	100s ISVs

Figure 28: comparison table from [81].

Shared Memory

Processors are independent but share same resources. Changes in one memory location are visible to all others. This solution is divided into

Uniform Memory Access (UMA): where access time is the same for all processors – commonly represented by symmetric multiprocessor machines (SMP); (summarized from [81])

Non-Uniform Memory Access (NUMA): is often made by physically linking several SMP's together in such a way that any processor can access the memory of any other (memory access over the link is slower); (summarized from [81])

COMA: (cache-only memory architecture) the shared memory is distributed among the caches of each processor [82].

Advantages are mostly related to the ease of implementation with fast and relatively uniform task and data sharing among processors. The disadvantages turn around difficult scalability, the expense and the programmer's responsibility for synchronizing. [81]

Distributed Memory

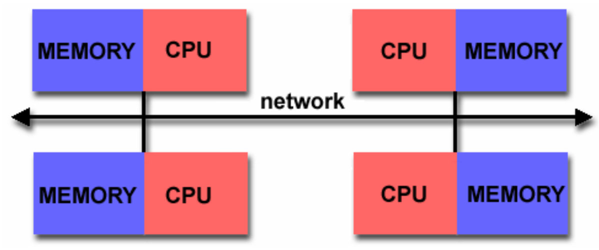


Figure 29: distributed memory block diagram [81].

Processors have their own memory and may be accessible to others, but there is no global addressing scheme. Each processor can operate independently and cache-coherency does not apply (changes made to local memory need not be reflected to the memory of other processors). The programmer is responsible for providing access to memories of other processors and the required synchronization. [81]

Scalability becomes an important advantage. Rapid access to a processor's own memory and not needing to maintain cache-coherency reduce processing time and interference from other processors. the whole result is commodity and cost-effectiveness. The disadvantage is that the application programmer must pay more attention to inter-processor communication, data structures and non-uniform access times. [81]

Hybrid (Distributed-Shared) Memory

The largest and fastest computers in the world today employ both shared and distributed memory architectures. The shared memory component is usually a cache coherent SMP machine. Processors on a given SMP can address that machine's memory as global. The distributed memory component is the networking of multiple SMPs. SMPs know only about their own memory - not the memory on another SMP. Therefore, network communications are required to move data from one SMP to another. Current trends seem to indicate that this type of memory architecture will continue to prevail and increase at the high end of computing for the foreseeable future. [81]

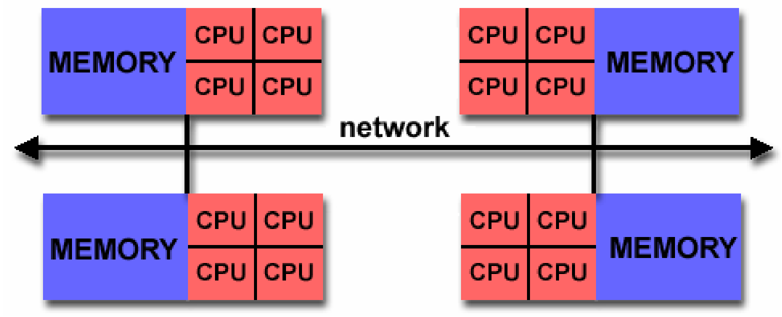


Figure 30: hybrid of shared and distributed memory. [81]

Parallel Programming Models [81]

Although Parallel Programming Models may resemble the computing models described above, the programming models **are not related to architecture**. Rather, they exist to answer some of the problems posed by the computing models, either instead of or in addition to hardware implementation.

Shared Memory Model [81]

The shared memory model is characterized by tasks sharing common address spaces that are read and written asynchronously. Being shared, these address spaces must be controlled for integrity by such mechanisms as locks and semaphores.

The advantage to the programmer is an apparent “lack of ownership”: communication between tasks need not be specified explicitly which simplifies the task of development. However, data locality may become difficult to manage.

On shared memory platforms, a direct mapping of program variables into actual address spaces is made globally.

Threads Model [81]

Threads can be described as a subroutine in a process that can be replicated many times and some or all of these replications run concurrently.

The important difference between a Process and a Thread is that different Processes do not share memory space and therefore context-switching requires more overhead. Threads may carry 'private' data, but much of the overhead belongs to the parent Process. Thus, they use shared memory to communicate between them, when need be. Memory is therefore global among Threads until the application has terminated. They are commonly associated with shared memory architectures and operating systems.

Threads are created through libraries, subroutines and compiler directives in either serial or parallel source-code. Different implementations of Threads exist. Most notably, POSIX Threads, Open MP and Java. Microsoft has its own implementation which is independent from the other's standards.

Message Passing Model [81]

Tasks have their own local memory for computation. When different tasks need to communicate, they exchange data by sending and receiving data. This requires at least a minimum amount of synchronization, as for each message sent, a receiver must be paired and ready.

Message passing is accomplished through libraries and subroutines. The programmer is responsible for all parallelism. A forum (Message Passing Interface, MPI) was formed with the goal of standardizing an interface for message passing. It is now the *de facto* industry standard.

Data Parallel Model [81]

Where data is organized as a set, it is often also organized as a common structure such as an array or a matrix. Where different tasks must operate on different sector or 'chunks' of this data, the data can be accessed 'in parallel'. Shared memory systems can access these chunks directly and distributed systems can have them spread over each processor's local memories.

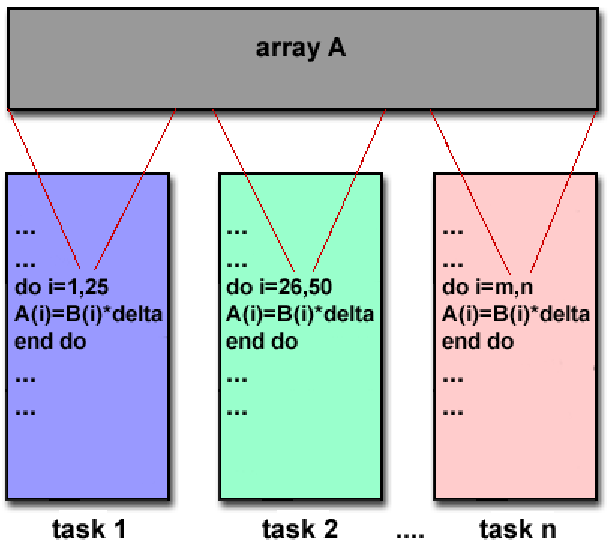


Figure 31: block diagram of the data parallel model.

The programming can be achieved through subroutines and/or compiler directives, although the programmer is often left with the task of defining it.

Distributed memory implementations of this model usually have a compiler to distribute the memory using MPI. The message passing is invisible to the programmer.

Other Models

Hybrid [81]

The Hybrid model combines any two or more shared memory systems by way of message passing. For example, Thread model systems can communicate among themselves or with shared memory models via MPI.

Single Program Multiple Data (SPMD) [81]

SPMD is a high level programming model that can be build atop any combination of the aforementioned parallel programming models. The same program is run concurrently on different processors. The data may be the same or different.

Multiple Program Multiple Data (MPMD) [81]

MPMD is a high level programming model that can be built atop any combination of the aforementioned parallel programming models. Different programs are run concurrently on different processors. The data may be the same or different.

Designing Parallel Programming

Automatic vs. Manual Parallelization

'Manual' parallelization of a source-code written sequentially is often difficult, therefore time-consuming and error-prone. It is an iterative process of improving upon previous attempts. Tools have been available for a number of years to perform the task automatically.

Compilers may have tools to allow for either fully automated parallelization of sequential code or programmer directed parallelization. In the case of fully automatic parallelization the compiler identifies all opportunities for turning the code into sequential tasks as well as the inhibitors thereto. Where programmers prefer to control parallelization themselves, they can direct it by using compiler directives, flags etc. to instruct the compiler on how to parallelize the code.

As with using C or assembly language to program processors used to know, there is some contention as to the benefits of automated parallelism. Programmers used to designing serial code may prefer to use the automatic parallelization tools. They must beware, however, that these tools are by no means fail-safe. Compilers usually rely on for-loops exclusively and may not be well optimized today. They remain less flexible than the manual method, may perform incorrectly (give wrong results) and even degrade performance. Therefore, a programmer, at present, may still better understand the parts of the code that can be assigned to different processors and they can best test for correct results. Much research is still needed in this domain.

Understand the Problem and the program

Some problems are best solved with parallelism while others are not. Where several similar routines can be executed independently from each other can be parallelized. For example, calculating the energy potential for every possible configuration of a system requires only reading the shared data. However, anything where the calculation of one operation depends on the results of another is inadmissible. For example, calculating the Fibonacci sequence is not suited to parallelism.

'Hotspots' and 'bottlenecks' in the code must be identified. Hotspots are the code segments where the real computation is actually happening. Performance analysis takes place here and this is where parallelism should be focussed. Bottlenecks are operations that may slow down the whole. For example, dependency on I/O operations may render the benefits of parallelizing negligible. Sometimes, alternate algorithms can be found to overcome these bottlenecks.

The big inhibitor to parallelism is data dependency, as illustrated with the Fibonacci example.

Partitioning

This is the process of breaking down the program into 'chunks' for parallelism. As described in the sections on parallel architectures and programming models, this can focus on data, routines or both. [81]

Domain Decomposition

Domain Decomposition is concerned with distributing different chunks of data among different routines (Processes or Threads). A good example is image 'snow removal' where a 'result' image is made from chunks of an 'input' image distributed among different instances of a same routine. [81]

Functional Decomposition

Functional Decomposition is concerned with getting different tasks to run concurrently towards a combined result. Many processes in the natural world are the combination of several, independent processes. For example, weather prediction relies on several models including atmosphere, ocean currents and temperatures, the effects of land relief etc. Each of these can be mathematically modelled separately and run in independent processes.

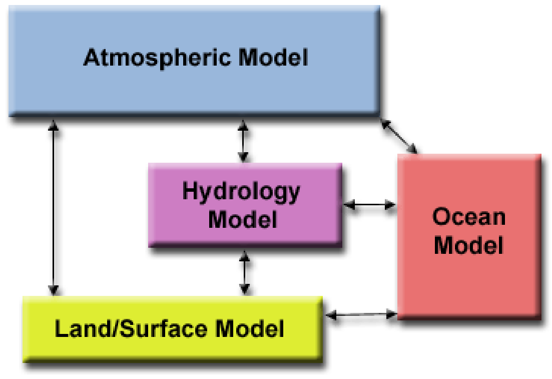


Figure 32: functional decomposition of weather modelling. [81]

Signal processing is another example of Functional Decomposition. A signal is typically passed through several filters. Each filter works independently in a pipeline. By the time the beginning of the signal data reaches the last filter, all filters are busy.

Communication

Which processes need to communicate, when and how? This depends on the problem definition.

Inter-process communication is not needed when the results of one routine need not be known by any other. The 'image snow removal' example cited requires no communication. This is often called “embarrassingly parallel” because the parallelization is so obvious and simple[81]. However, the obviousness of parallelism might require so much communication between processors that the end result is actually slower than the sequential solution.

Communication is needed when tasks require data sharing. For example, thermal regulators may need to consider the data from several thermostats before deciding what to do next. [81]

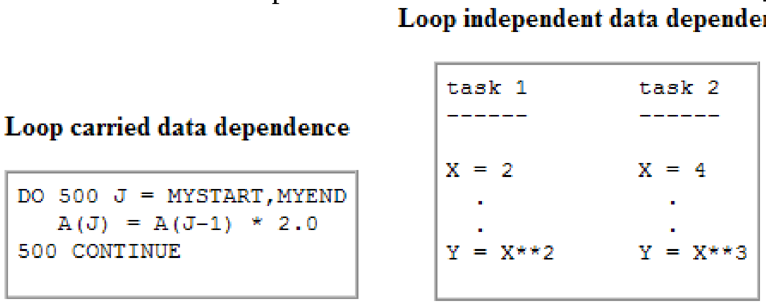
When deciding if communication is needed or not, the factors to consider include overhead and complexity of inter-task communication (do they outweigh the benefits of parallelism), latency and bandwidth (are the sufficient to improve overall performance), visibility of communications (depending on the communications model used, how much extra work is required by the programmer and are the employed programmers able?), is the hardware communication asynchronous (is the latency caused by handshaking problematic?). Many more factors need to be considered as dictated by the problem to be solved and the system designs considered. [81]

Synchronization

We can distinguish 3 types of synchronization. A Barrier ensures that processes continue only after all have reached specified meeting points. Locks (or Semaphores) may be either blocking or non-blocking and are used to ensure data integrity. Synchronous communication operations are used for coordination between sender and receiver.

Data Dependencies

Data dependencies are the principal obstacles to parallel computing. Loop carried data dependence is code where the calculation in one loop iteration depends on the results of the previous iteration. These are the most important to identify since they are the primary target for code parallelization. Similarly, Loop independent data dependence takes place when more than one processor writes to the same variable. [81]



Loop carried data dependence

```
DO 500 J = MYSTART,MYEND
   A(J) = A(J-1) * 2.0
500 CONTINUE
```

Loop independent data dependence

```
task 1              task 2
------              ------

X = 2               X = 4
  .                   .
  .                   .
Y = X**2            Y = X**3
```

Figure 33: two types of data dependence [81].

Data dependencies can be handled by applying synchronization at specific places in the execution of the task. For distributed memory architectures, communicate required data at synchronization points. For shared memory architectures, synchronize read/write operations between tasks. [81]

Load Balancing

When multiple processors run in parallel, they will not all be busy all the time. Performance optimization, in this context, is to minimize the waiting time of the processors. This is called Load Balancing: to ensure that all processors are busy as much as possible. However, the execution of any given parallel program depends on its slowest task. Therefore, processors performing shorter tasks may have to wait for the others to complete. It is therefore preferable, where possible, to distribute tasks unevenly in order to maximize the use of every processor.

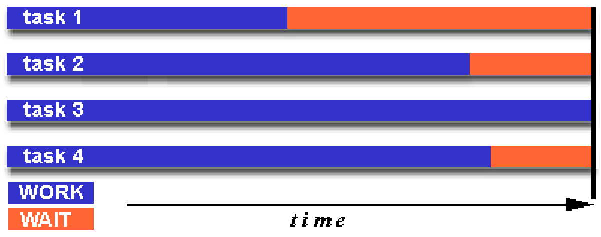


Figure 34: time processors are busy and idle.

Load balancing can be achieved in two steps. First, work can be evenly partitioned among tasks (e.g. array or matrix operations can be divided into similar chunks, or the work in any iteration of a loop is often quite similar) and then distributed among the processors. Performance analysis tools can be used to determine task length where it is

more difficult for intuition or analysis. Second, dynamic work assignment is used to immediately give a new task whenever any processor terminates its previous task. These tasks are assigned by as scheduler and may require a detection algorithm to find out when processors go idle.
Examples of situations with uneven work distribution include sparse arrays (arrays where some cells have values that produce no computation), adaptive grid methods (where some tasks must refine their mesh while others don't) and N-body simulations (where some particles of data may need to migrate to/from their original domains). [81]
Reduce communication:
Often times, a mathematician may say that a given algorithm is perfectly suited to parallel programming. However, this algorithm may require such inter-process communication that the end result is nearly as slow as the sequential solution.
In such cases, it may be possible to modify the algorithm or choose another one in order to make the extra effort and cost of parallel computing worthwhile.
Granularity
In the context of parallel computing, granularity is the ratio of communication time over computation time. Fine grain parallelism is characterized by seemingly more communications as the relative computation time is shorter. Coarse grain parallelism, then, is characterized by seemingly fewer communications with much longer computation time.

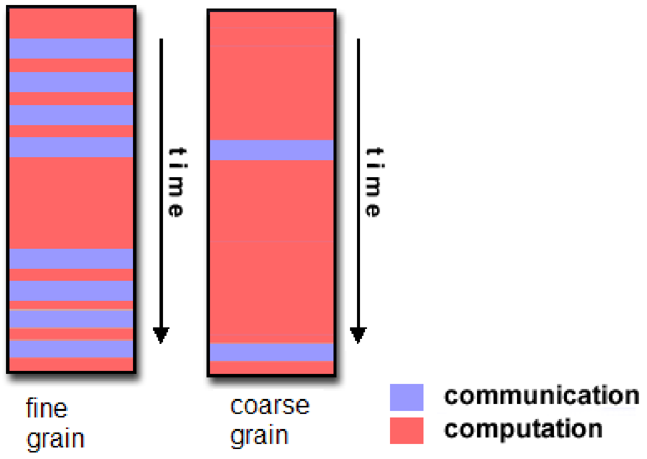


Figure 35: granularity of parallel computing.
Load balance is easier to achieve with fine grain parallelism because small tasks depend less on the operating system, interrupts and so on. Coarse grain parallelism, on the converse, makes it harder to predict when any given task will terminate, therefore making it harder to assign tasks for optimal usage of the multiple processors.
Fine grain parallelism requires more synchronization overhead due to the need to communicate data and synchronize tasks among processors. Therefore, the fewer communications in coarse grain parallelism reduces overhead. [81]

Summary

The utilisation of fine or coarse grain parallelism, then, depends on the goal. When the goal is to keep the processors as busy as possible, then fine grain is better. But when the goal is to reduce traffic, coarse grain parallelism is often a better choice.

Paper 1: Parallel Programming Models for a multiprocessor SoC Platform Applied to Networking and Multimedia. [83]

Paper 2: THE STANFORD HYDRA CMP [84]

Lecture #12b: Parallel programming models

Course: Computer Architecture III
Lecturer: Miodrag Bolic
Scribe(s): Muheto Yvan and Tony Lteif
Date: October 21st, 2006

Introduction

Computer programmers and designers realized that the process of solving a problem can usually be divided into smaller tasks, which can be executed simultaneously with some coordination, where in this case a given program will be executed much faster then being executed on a sequential computing system. In order to take advantage of this principle they developed parallel computing systems, which are systems with more than one processor for parallel processing. By parallel processing, we mean the simultaneous execution of the same task on multiple processors in order to obtain results faster.
To program and take advantage of the particular architecture of such systems, parallel programming models have been developed.

A parallel programming model is a set of software technologies to express parallel algorithms and match applications with the underlying parallel systems. It encloses the areas of applications, programming languages, compilers, libraries, communications systems, and parallel I/O. [86]
These models are independent of any hardware and memory architecture.
To develop parallel application on a particular platform we have to choose one parallel programming model or a combination of several of them. The ultimate goal is to improve the productivity of programming.

Explanation

In this scribing lecture we will be talking about:

- Shared Memory
- Threads
- Message Passing
- Data Parallel
- Hybrid (SPMD and MPMD)

1. Shared memory model

In a shared memory model, multiple processors can communicate together by reading and writing from a shared memory (large block of random access memory) which can be equally accessible by all processors in the multiple-processor computer system (Figure1). In this model, multiple processors will be executing in parallel simultaneously often with a different functionality.
In this model, two Processors can communicate together sharing variables or data structures without even talking or knowing each others identity. In addition to the shared memory space, each processor has its own private memory space (registers, buffers, caches ….) which is only accessible by him and not other processors.

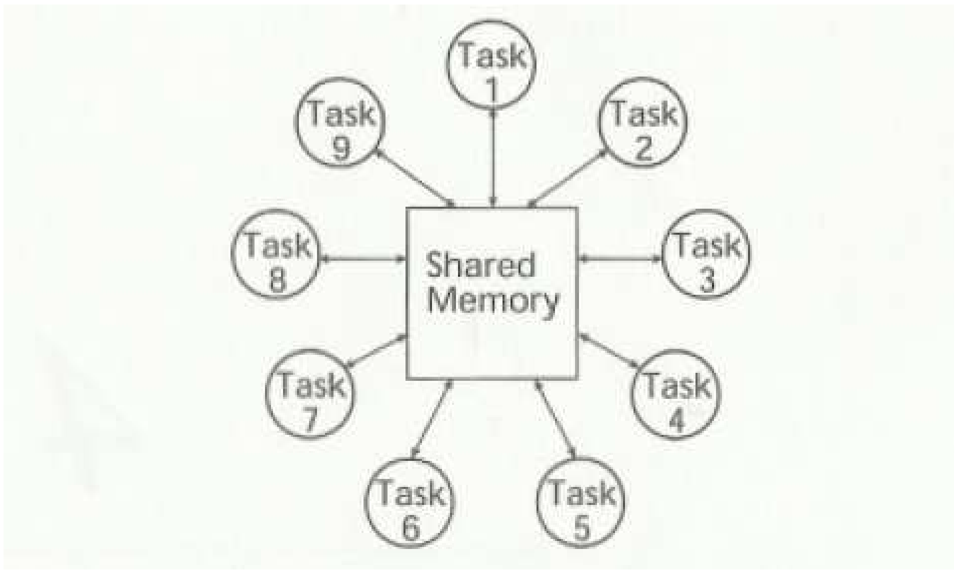


Figure1. Shared memory model [91]

Synchronization In a shared memory system, synchronization must be provided. Processors should explicitly coordinate together their actions to ensure that only one of them is performing an operation on some data at a time so the processors always use the last updated right data. So in case a processor needs the result of an operation being performed by another processor then it has to wait till the computation is done. Synchronization ensures that the information flows properly and ensures system functionality.
A disadvantage of synchronization is that some processors have to wait which affect the performance of the system by slowing down the program (it is only a slow down in case processors are performing a useful task). So for efficiency, synchronizations should be avoided wherever possible [91].

Access control should be provided too. Many ways can be used to control access to shared memory. One of them is by using a lock. A processor can access the shared memory by encountering the lock. The other processors have to wait till this processor unlocks the key so one of them can access the shared memory.
For example if we have three parallel processes using locks to ensure mutual exclusion. If P1 is executing in the critical section then P2 has to wait until P1 unlock the critical section; same for P3 has to wait until P2 issues the unlock statement.

2. <u>Threads model</u>

A thread can be defined as “a sequence of instructions which may execute in parallel with other threads” [86]. Threads were invented to be a faster alternative to processes. Each process can have several threads, and each thread belongs to a process (figure 2).

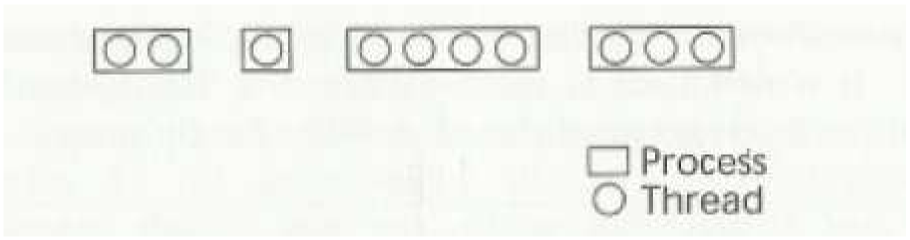


Figure 2. relationship between processes and threads. [91]

Therefore, a program that has multiple concurrent execution paths can create and assign a separate thread for each one of them, while it is itself assigned to one process (figure 3).

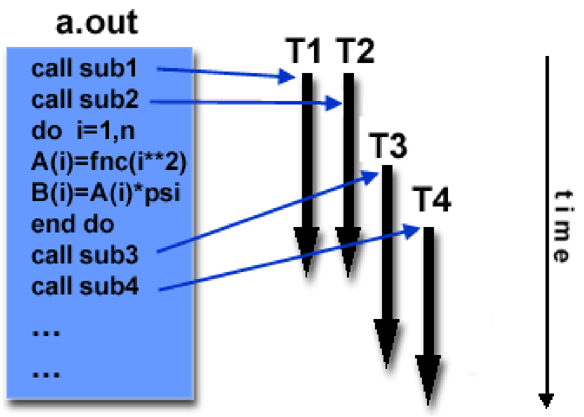


Figure 3. One program in one process, and several threads for the different subroutines. [87]

The following reasons make multiple threads faster than multiple processes:
- The threads will share their program code, address space, file pointers and other resources.
- and when the operating system transfers control from one thread to another, the information is stored in user space, whose access is faster than the kernel space.
The thread model considers a program as a collection of threads that cooperate in order to accomplish a common goal or to share resources in a programmer-controlled way. The threads communicate with each through shared memory, but they do not share their local variables.
Thread models have many application areas such as client/server networks, applications with very large number of tasks.

The implementations of thread models is mainly done through either a library of subroutines that are called from within parallel source code or a set of compiler directives imbedded in either serial or parallel source code. There are many different implementation of thread model but two of them are more widely used:

The POSIX threads

It is library based and requires parallel coding. The Pthreads (stands for POSIX threads) library defines a set of about 50 functions. They can be called from a C program easily or from another programming language but it is much more complicated. It provides several functionalities, these are the main ones:
- Thread management: creation and termination of threads.
- Thread scheduling: how threads are assigned to the processor(s) (scheduling algorithms).
- Synchronization: protection of the critical section of codes and avoidance of busy waiting. Mutexes and semaphores are examples of implementations available to protect the critical section.

Java Threads

This model reflects the object oriented nature of java. Among the set of predefined classes of the java language there is a class Thread that contains methods to deal with threads. To create and use threads the programmer has to create a class that inherits methods from the Thread class (example of methods: start (), run (), stop ()).
It also provides several functionalities, these are the main ones:
- Thread management: creation, suspension, resuming, interruption and termination of threads.
- Thread scheduling: how threads are assigned to the processor(s) (scheduling algorithms).
- Synchronization: protection of the critical section of codes and avoidance of busy waiting. The main synchronization mechanism is called “monitor”.

There is also **Open MP** (a structured shared memory model) which is a better fit for fine-grained parallel applications than the threads models. It is compiler directive based and can use serial code not only parallel code. It is very simple to use and allow incremental parallelism (progressive transformation of serial code in parallel code).

<u>Message passing Model</u>

In Message Passing systems, unlike shared memory systems, processors communicate together by explicitly exchanging messages. Processors can communicate with other processors on the same machine or on different one. Each processor has access to its own memory space. Communications are performed using **send and receive** operations.
If two processors wants to communicate, one of them has to invoke a **send** and the other must invoke a matching **receive**. After that, the two processors start exchanging messages(figure 4) (Note that the communication is always established between only two processors [91]).

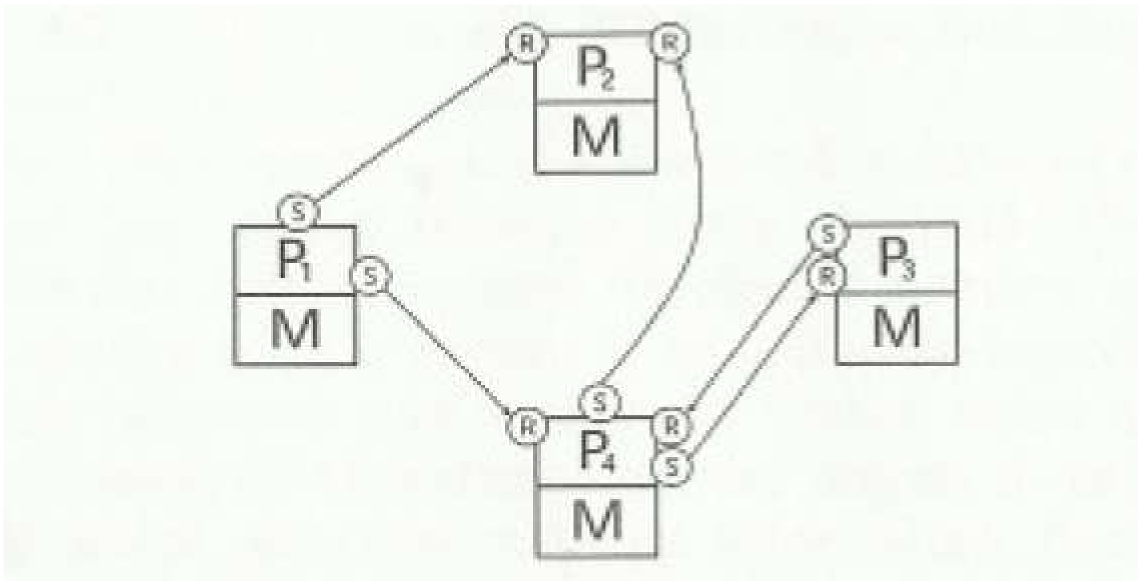


Figure 4. The message passing model. S stands for send and R stands for Receive [91]

Message passing can be used for different purposes.

1. Two processors (sender and receiver) can exchange data in order to accomplish a certain operation and whose interaction was planned in detail by the programmer.
2. In case that the interaction wasn't planned in advance, then two processors can establish a connection between them. The receiver posts an anonymous receive and waits for requests. The sender sends an initial message that contains details about the desired connection to this receiver. After that the connection is established.
3. Synchronization can be done here by a process that sends a message which tells the other process that it has reached a certain point of program execution.

Programmability:
The program is responsible for a heavy workload that has to be done. He is responsible for data distribution, task scheduling and management of the communication between tasks and other particular responsibilities as load balancing and more. So we can conclude that programmability is time-consuming, and lots of errors can occur which include debugging and maintaining the programs.

Efficiency:
Message passing are best known by their efficiency since the programmer controls everything. The programmer can do any necessary modification that can make the performance better. In this case optimum performance can be achieved. Efficiency is the major reason for the predominance of message passing in compute-intensive application domains[92].

<u>Data Parallel Model</u>

This model is based on the concept of data parallelism which is characterized by the parallel execution of the same operation on different parts of a large data set. [91]

The data is organized in any kind of data structure like an array, a cube, a list …, but most usually an array (figure 5).

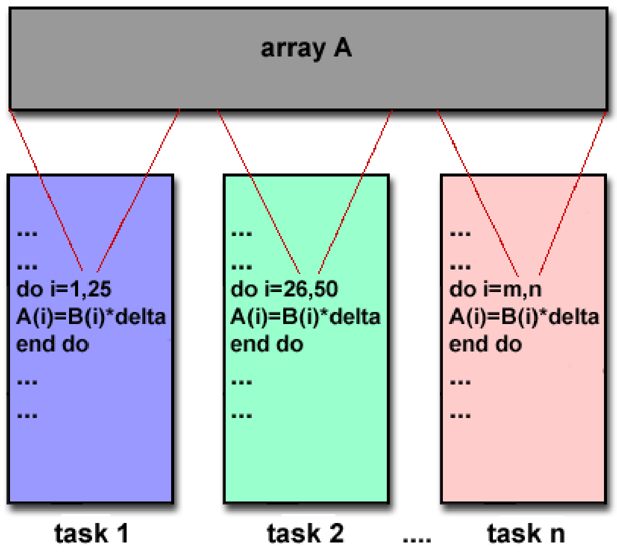


Figure 5. Example of data-parallel execution with data organized in an array. [87]

The main advantage of this model is program simplicity. In fact, we have a single thread of control. The parallelism is applied during the data-parallel steps only, which are mixed with the sequential steps. In the parallel steps, the operation is applied to all the data domains independently and in parallel (figure 6).

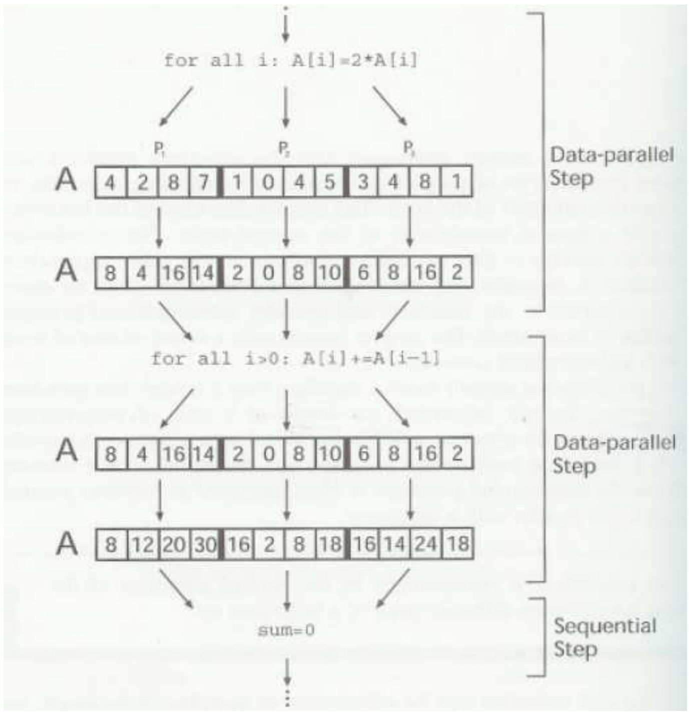

for all i: A[i]=2*A[i]
P_1
P_2
P_3
A 4 2 8 7 1 0 4 5 3 4 8 1
A 8 4 16 14 2 0 8 10 6 8 16 2
Data-parallel Step
for all i>0: A[i]+=A[i-1]
A 8 4 16 14 2 0 8 10 6 8 16 2
A 8 12 20 30 16 2 8 18 16 14 24 18
Data-parallel Step
sum=0
Sequential Step

Figure 6. The principle of data parallelism. [91]

We distinguish two basic styles of data parallelism according to the complexity of the parallel operation: SIMD (Single Instruction, Multiple Data) and SPMD (Single Program, Multiple Data).

SIMD parallelism

Data parallelism originates from programming models for SIMD architecture. In this architecture, the operation performed in the data-parallel step is a very elementary one (for example: addition). This data parallelism model has a major drawback which is its restriction to problem with a very regular structure.

SPMD parallelism

In this style of data parallelism the operation performed in the data-parallel steps are complex. The advantage of this style over SIMD is a greater performance; however it gives up to a certain extent the simplicity of the single flow of control. Programs become harder to design and implement.

Nested Data Parallelism

This is not really a style different from the two above. This style comes in when the data domains the above styles are themselves internally structured and if the operations to be executed on those data domain are internally data-parallel again.

Implementation

Programming with the data parallel model is usually accomplished by writing a program with data parallel constructs. The constructs can be calls to a data parallel subroutine library or, compiler directives recognized by a data parallel compiler. [87]

- Compiler Directives: Allow the programmer to specify the distribution and alignment of data. Fortran implementations are available for most common parallel platforms. [87]
- Libraries: there are several programming languages that support data parallelism programming: C*, Fortran, HPF (),(High Performance Fortran), HPF2 (a newer version that extends HPF with both regular language features and a large set of approved extensions) ….

<u>Other Models:</u>

<u>Hybrid:</u>

In the hybrid model, two or more parallel programming models can be combined.
Two examples of those models are Single Program Multiple Data stream (SPMD) and Multiple Program Multiple Data (MPMD).

Single Program Multiple Data stream (SPMD)

(High level programming model)
To build this model, we need to combine any two parallel programming models that we mentioned above. The SPMD parallelism which is characterized by the parallel execution of the same operation on different parts of a large data set. Each process may use different data then the others.
For the implementation of the SPMD model, common available functionality in the serial environment should be provided in the parallel environment so the serial source code can be used on the distributed memory machine.
One of the most important functionalities that is provided in the parallel programming model to support basic functionality of the serial code is a parallel I/O system. This can then be used in place of the serial I/O system, to support the required functionality of the parallel SPMD programming environment. [91]
In SPMD model, processors have the capability to execute only the parts of the program they are designed to execute so they don't have to execute the entire program.
In this case two different processors can be executing the same program but at any moment in time they can be executing different instructions within it(figure 7).

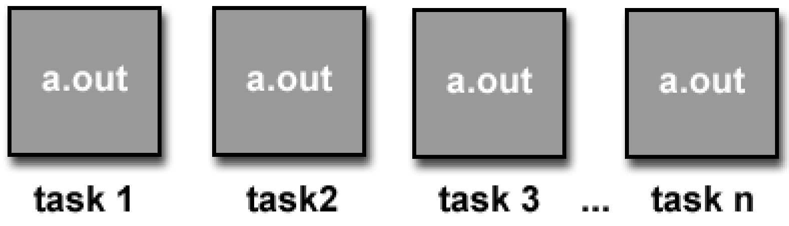


Figure 7. SPMD (all tasks executing the same program a.out) [91]

Multiple Program Multiple Data (MPMD):

(High level programming model)

Same as SPMD model, MPMD model is built by combining two parallel programming models of the previously mentioned one. Like SPMD, common available functionality in the serial environment should be provided in the parallel environment so the serial source code can be used on the distributed memory machine.
The difference between this model and SPMD model is that in MPMD model, processors are executing same or different programs (not only one) then the ones being executed by the other processors using same or different data (figure 8).

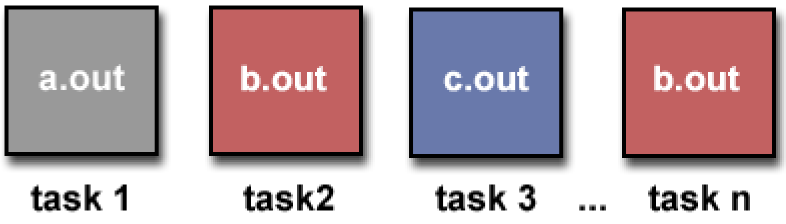


Figure 8. MPMD [91] (tasks executing different programs)

Summary

As we can see that several parallel programming models can be used to achieve more and more speed. Shared memory model is where multiple processors can communicate together by reading and writing from a shared memory which can be equally accessible by all processors in the multiple-processor computer system. Threads model is where a program that has multiple concurrent execution paths can create and assign a separate thread for each one of them, while it is itself assigned to one process. Message passing model, unlike shared memory systems, is where processors communicate together by explicitly exchanging messages. Data Parallel model is where we have the parallel execution of the same operation on different parts of a large data set. Hybrid model where two or more parallel programming models can be combined; for example SPMD where we have parallel execution of the same operation on different parts of a large data set and MPMD where processors are executing same or different programs (not only one) then the ones being executed by the other processors using same or different data.

Paper 1 Global Arrays: A Non-Uniform-Memory-Access Programming Model For High-Performance Computers

Paper 2 A Comparisons of Three Programming Models for Adaptive Applications on the Origin2000

Lecture #13: Cache Coherence – Snooping Protocols

Course: CEG4131 – Computer Architecture III
Lecturer: Miodrag Bolic
Scribe: Hala Issa and Mazin Alkarkhi
Date: October 25, 2006

Introduction

The topic covered in this lecture was cache coherence and its protocols. Cache coherence is a mechanism itself which manages the cache of a multiprocessor system. The purpose of cache coherence is so that no data is lost or overwritten before that data that is used is transferred from a cache to the target memory. When two processors work together this is called multiprocessing, where each processor may have its own memory cache that is separate from the larger RAM. A memory cache, also called a RAM cache is a portion of memory made of high-speed static RAM (SRAM) instead of the slower and cheaper dynamic RAM (DRAM) which would be used for main memory. Memory caching is very effective and useful since it stores the most accessed data or memory that programs tend to use over and over. [96]

Explanation

Figure 1 is a depiction of basic structure of a memory hierarchy. Here since the memory is implemented as a hierarchy it gives the user an illusion of a memory that is as large as the largest level of the hierarchy, but the access is built as if it was built from the fastest memory. [95]

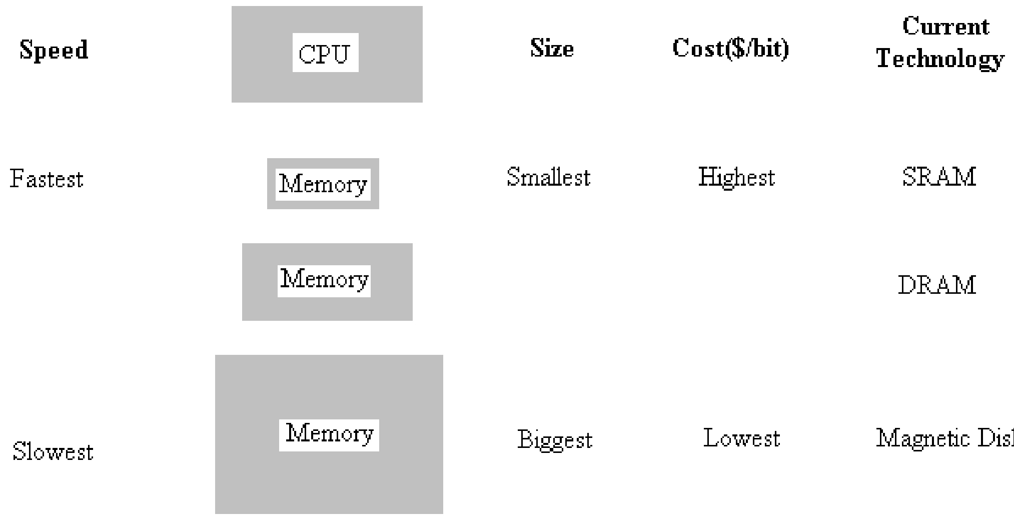


Figure 1 The Basic Structure of a memory hierarchy [96]

When data is requested by a processor and the data is located this is called a hit, the data is found in the cache. Whereas if the data is not found in the upper level of the cache this is called a miss, the data is not found in the cache. We also have a hit rate which is denoted by the letter h, which is a fraction of the memory access that is found in the upper level, this is usually used to measure the performance of the memory hierarchy. The miss rate is denoted by the letter m, and is calculated with this equation: $m = (1 – h)$. This is the fraction of memory access not found in the upper level. These values are highly important since this is an indication of performance of the system. [96]

When we have multiple processors that have separate caches and share a common memory it is important to keep the cache in a state of coherence, which means that any shared operand that is changed in any of the cache is changed throughout the whole system. This is done through two different ways, one is directory based and the other is a snooping system. This lecture covers the snooping protocols. Snooping protocols is best explained by stating that when all the caches on the bus either monitor (or snoop) the bus to determine if they have a copy of the block of data that is being requested and is available on the bus. Each cache has a copy of the sharing status of every block of physical memory it has. Directory based on the other hand, is thought of more as a filter where the processor must ask permission to load an entry from the primary memory to its cache. [95]

Cache Coherence Policies

These policies are implemented to avoid any inconsistencies throughout the parallel system. This is important since the values once changed may not give a correct final result. This is needed since the values need to be the same, if they are not it will lead to errors for the final result.

Cache-Memory Coherence: Writing to cache with one processor

When a system has a single cache the coherence between memory and cache is kept using one of two policies; write-through or write-back.

Write-through always updates the cache and memory whenever there is are new updates, this makes sure that the data is always consistent between the cache and memory. Every write, writes data to main memory. Table 1 shows what happens in the cache and memory location during a two instruction program that handles the write-through policies. [96]

Write-back on the other hand, handles a write with updating the values only to the block in the cache, then the block which was modified is written to the lower level of the hierarchy whenever the block is replaced. This can improve the performance, and is more complex than write-through. [96] Table 1, shows what happens with cache and memory location during a two instruction program that handle write-back. [97]

		Write-Through		Write-Back	
Serial	Event	Memory	Cache	Memory	Cache
1		X		X	
2	P reads X	X	X	X	X
3	P updates X	X'	X'	X	X'

Table 1: Write-Through vs. Write-Back [97]

Cache-Cache Coherence: Writing to cache with n processors

When many processors share the same cache there are problems that may occur. For example, if a processor requests the data from a memory location X, and this memory location is copied into the processors cache. Now if a processor Q requests the same memory location but decides to replace the value, then there is a conflict with the memory location for both processors. We deal with this in two ways: write-invalidate and write-update.

The following will be an example which will be used to explain write-invalidate and write-update. Write-invalidate remains consistent by reading the local cache until a write takes place, once this happens a dirty bit is set for data X, which has been written. Now lets assume another processor decides to write to the bit X which has a dirty bit, it will wait until the dirty bit is cleared. This is where write-update occurs, it maintains consistency by continuously updating all the copies in the cache, and all the dirty bits are set during a write operation. Once all the copies are updated, the dirty bits are cleared. Table 2 is a summary of the policies that run write-update and write-invalidate. [97]

		Write-Update		Write-Invalidate	
Serial	Event	P's Cache	Q's Cache	P's Cache	Q's Cache
1	P reads X	X		X	
2	Q reads X	X	X	X	X
3	Q updates X	X'	X'	INV	X'
4	Q updates X'	X"	X"	INV	X"

Table 2: Write-Update vs. Write-Invalidate [97]

Shared Memory System Coherence

The combination of these four together creates coherence amongst all caches and the global memory, combined they can form any of the following four protocols:

- Write-update and write-through
- Write-update and write-back
- Write-invalidate and write-through
- Write-invalidate and write-back

Either one of these combinations will be used according to the needs of the situation at hand to create coherence in the system, snooping is used to assign the proper protocol when needed.

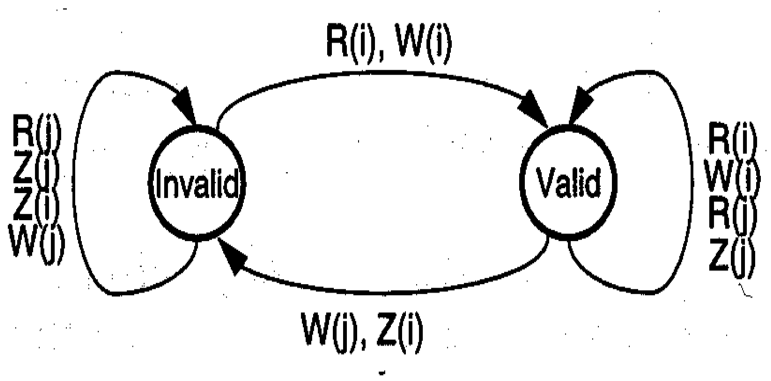
Snooping Protocols

Snooping protocols were placed to keep the coherency in order, what happens is the bus activities are constantly being watched and then according to the needs, the appropriate protocol is used to rectify the situation. Global memory is moved in blocks, the block as a whole becomes evaluated and then this determines which protocol is to be used for the whole block. The state of that particular block may become different due to the operations Read-Miss, Read-Hit, Write-Miss, and Write-Hit. The following are snooping protocols. [97]

Write-Invalidate and Write-Through

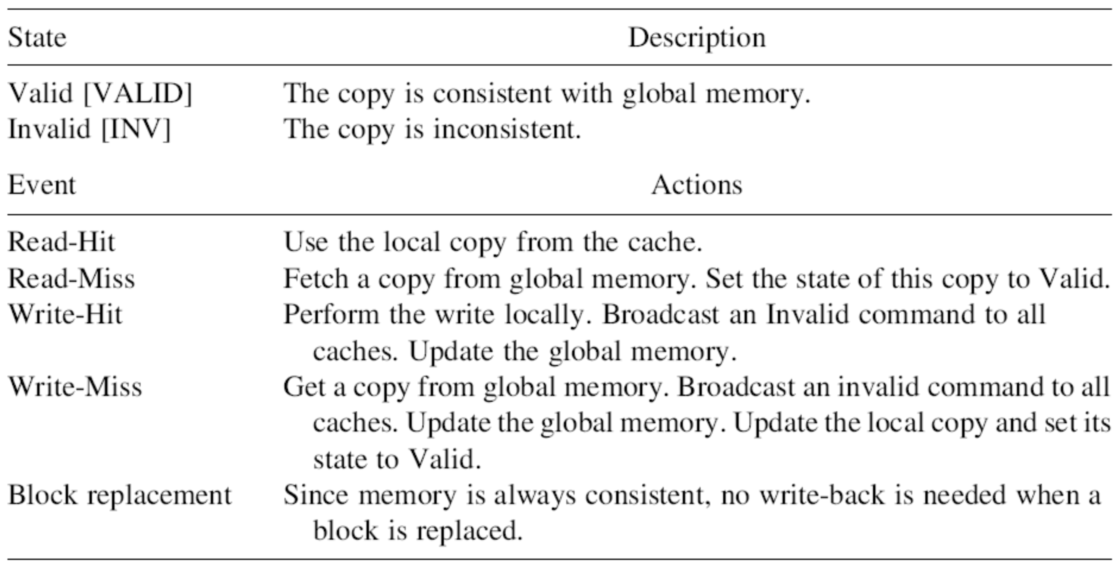
This protocol allows the memory to always contain the most updated cache copy. From the global memory blocks the multiple processors can safely read until the update takes place, at which time, the cache copies are invalidated and the memory is updated to remain consistent. [97]

Figure 2, is a diagram of how the cache can change its state from valid to invalid. For example if the cache is in an invalid state and a read and write takes place from the local processor, this action takes the cache state to valid.



R = Read, W = Write, Z = Replace,
i = local processor, j = other processor
Figure 2: Write-Through Cache State Transitions [98]

Table 3 defines the different states and events that may occur in the write-invalidate and write-through protocols, and table 4 is an example, step by step showing the different values that are located in each caches location and its state.



State	Description
Valid [VALID]	The copy is consistent with global memory.
Invalid [INV]	The copy is inconsistent.
Event	**Actions**
Read-Hit	Use the local copy from the cache.
Read-Miss	Fetch a copy from global memory. Set the state of this copy to Valid.
Write-Hit	Perform the write locally. Broadcast an Invalid command to all caches. Update the global memory.
Write-Miss	Get a copy from global memory. Broadcast an invalid command to all caches. Update the global memory. Update the local copy and set its state to Valid.
Block replacement	Since memory is always consistent, no write-back is needed when a block is replaced.

Table 3: Write-Invalidate Write-Through Protocol[97]

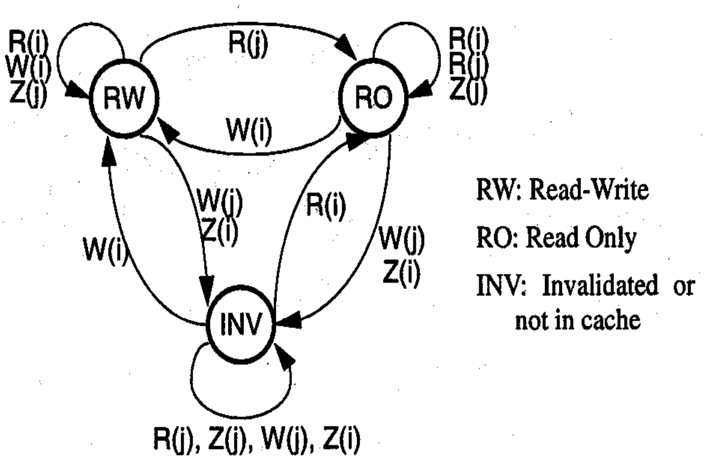
		Memory	P's Cache		Q's Cache	
Serial	Event	Location X	Location X	State	Location X	State
0	Original value	5				
1	P reads X (Read-Miss)	5	5	VALID		
2	Q reads X (Read-Miss)	5	5	VALID	5	VALID
3	Q updates X (Write-Hit)	10	5	INV	10	VALID
4	Q reads X (Read-Hit)	10	5	INV	10	VALID
5	Q updates X (Write-Hit)	15	5	INV	15	VALID
6	P updates X (Write-Miss)	20	20	VALID	15	INV
7	Q reads X (Read-Miss)	20	20	VALID	20	VALID

Table 4: Example (Write-Invalidate Write-Through) [97]

Write-Invalidate and Write-Back (Ownership Protocol)

This protocol allows the valid block to be owned by memory and shared in multiple caches; these caches can only contain the shared copies of the block. The multiple processors present can read these blocks from their cache, until the processor updates its copy. At this point the only owner of this block is the writer, all other copies are invalidated. [97]

The figure below, figure 3 is a visual aid in showing the transitions that can take place in a write-back cache. For example, to get out of the Read Only state, a write has to take place to processor i, or a write to block copy in the cache j by processor j ≠ i, and a replace takes place in block copy in cache j ≠ i. Tables 5 and 6 are a run through the different block states and protocols which are summarized and the contents of the memory and the two caches as they are executing different write and reads.

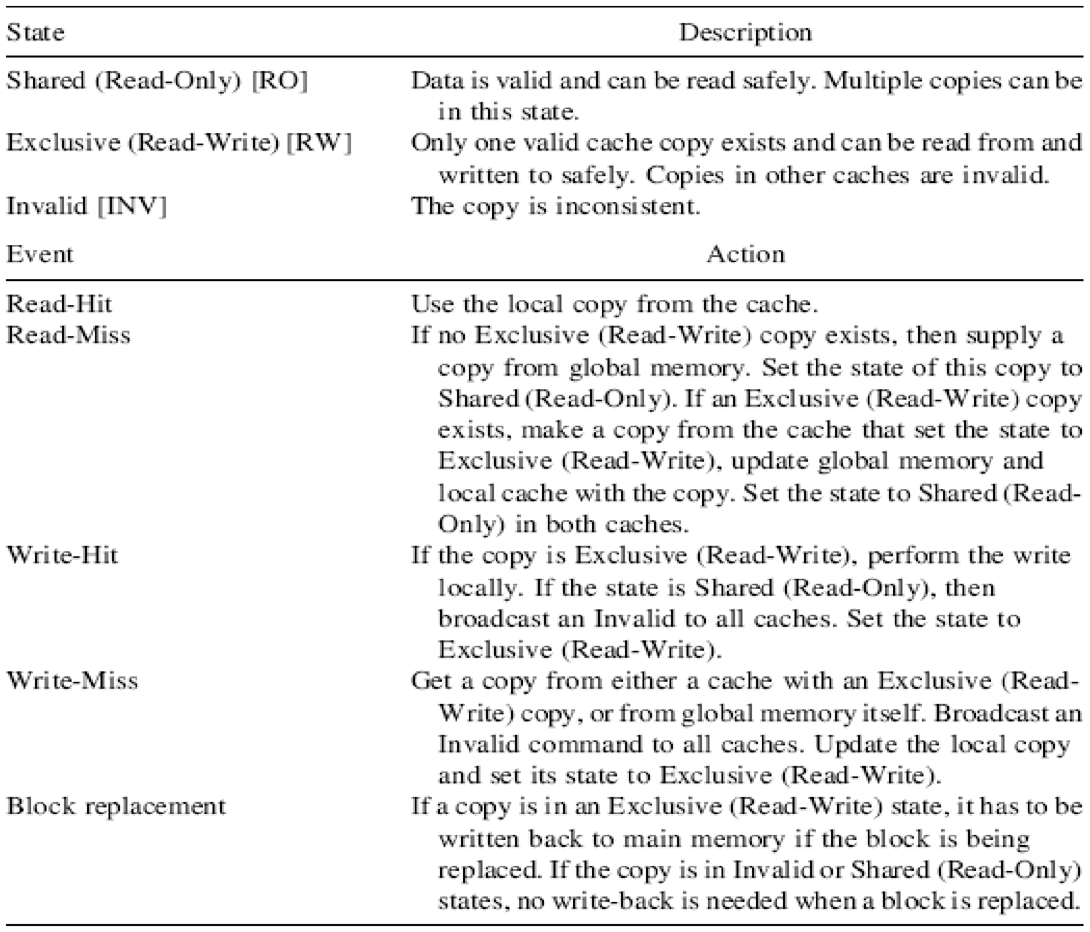


W(i) = Write to block by processor *i*. W(j) = Write to block copy in cache *j* by processor $j \neq i$.

R(i) = Read block by processor *i*. R(j) = Read block copy in cache *j* by processor $j \neq i$.

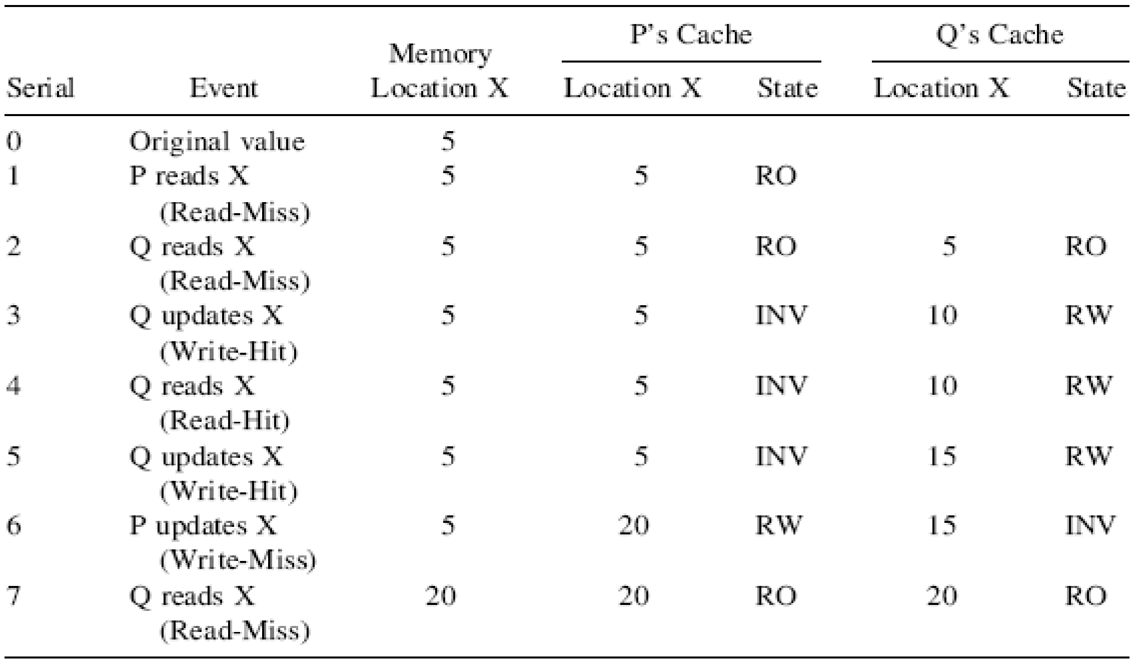
Z(i) = Replace block in cache *i*. Z(j) = Replace block copy in cache $j \neq i$.

Figure 3: Write-Back Cache[98]



State	Description
Shared (Read-Only) [RO]	Data is valid and can be read safely. Multiple copies can be in this state.
Exclusive (Read-Write) [RW]	Only one valid cache copy exists and can be read from and written to safely. Copies in other caches are invalid.
Invalid [INV]	The copy is inconsistent.
Event	Action
Read-Hit	Use the local copy from the cache.
Read-Miss	If no Exclusive (Read-Write) copy exists, then supply a copy from global memory. Set the state of this copy to Shared (Read-Only). If an Exclusive (Read-Write) copy exists, make a copy from the cache that set the state to Exclusive (Read-Write), update global memory and local cache with the copy. Set the state to Shared (Read-Only) in both caches.
Write-Hit	If the copy is Exclusive (Read-Write), perform the write locally. If the state is Shared (Read-Only), then broadcast an Invalid to all caches. Set the state to Exclusive (Read-Write).
Write-Miss	Get a copy from either a cache with an Exclusive (Read-Write) copy, or from global memory itself. Broadcast an Invalid command to all caches. Update the local copy and set its state to Exclusive (Read-Write).
Block replacement	If a copy is in an Exclusive (Read-Write) state, it has to be written back to main memory if the block is being replaced. If the copy is in Invalid or Shared (Read-Only) states, no write-back is needed when a block is replaced.

Table 5: Write-Invalidate Write-Back Protocol [97]



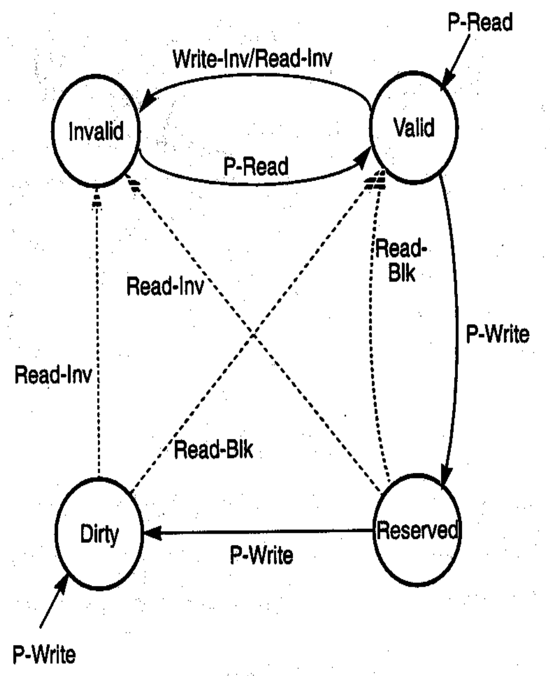
Serial	Event	Memory Location X	P's Cache Location X	P's Cache State	Q's Cache Location X	Q's Cache State
0	Original value	5				
1	P reads X (Read-Miss)	5	5	RO		
2	Q reads X (Read-Miss)	5	5	RO	5	RO
3	Q updates X (Write-Hit)	5	5	INV	10	RW
4	Q reads X (Read-Hit)	5	5	INV	10	RW
5	Q updates X (Write-Hit)	5	5	INV	15	RW
6	P updates X (Write-Miss)	5	20	RW	15	INV
7	Q reads X (Read-Miss)	20	20	RO	20	RO

Table 6: Example (Write-Invalidate Write-Back) [97]

Write-Once

This protocol is similar to the write-invalidate protocol, which was proposed by Goodman in 1983. Where it uses a combination of write-through and write-back. Write-through is used only on the first try where the block is written, then any writes thereafter use write-backs. [97]

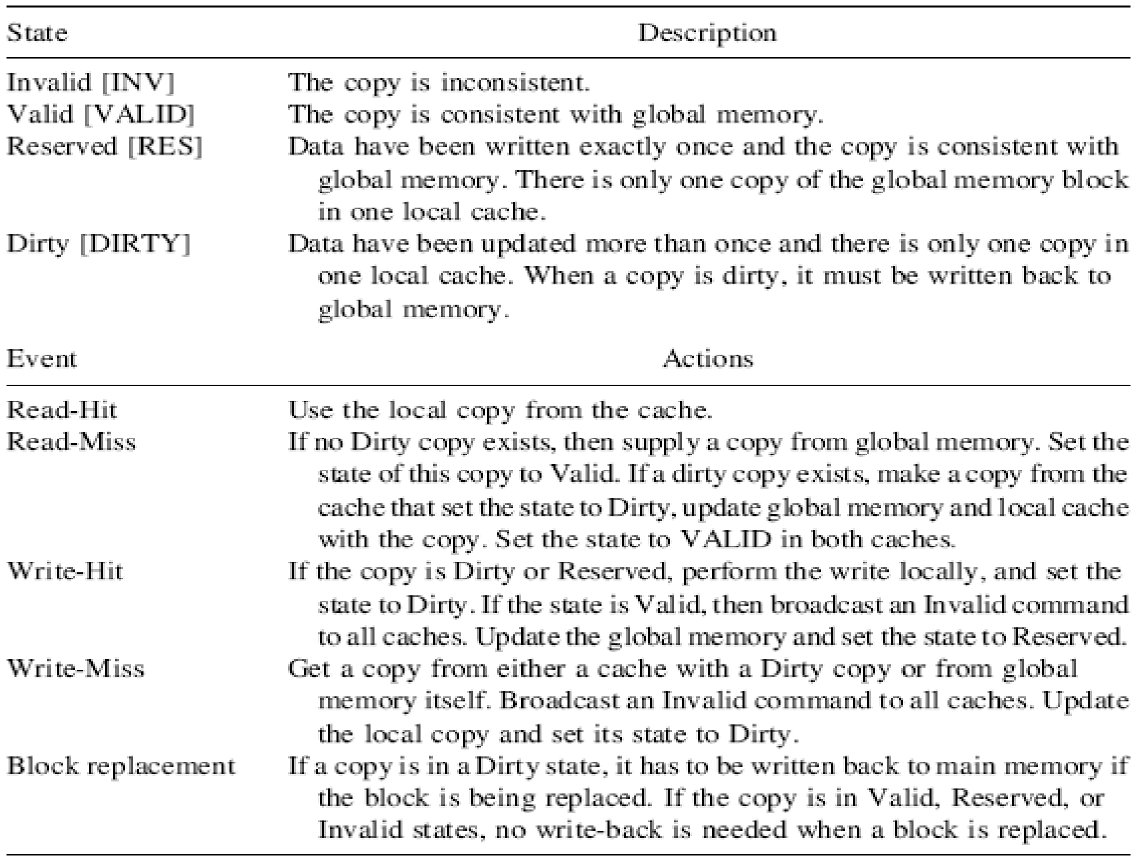
Figure 4 shows the Goodman Write-Once protocol state diagram, showing the different states and how you arrive at each state, it shows the lines that are commanded by the local or remote processors. Table 7 defines the write-once protocols and table 8 is an example that walks through the steps of the write-once protocol with different read, and update events.



Solid lines: Command issued by local processor

Dashed lines: Commands issued by remote processors via the system bus.

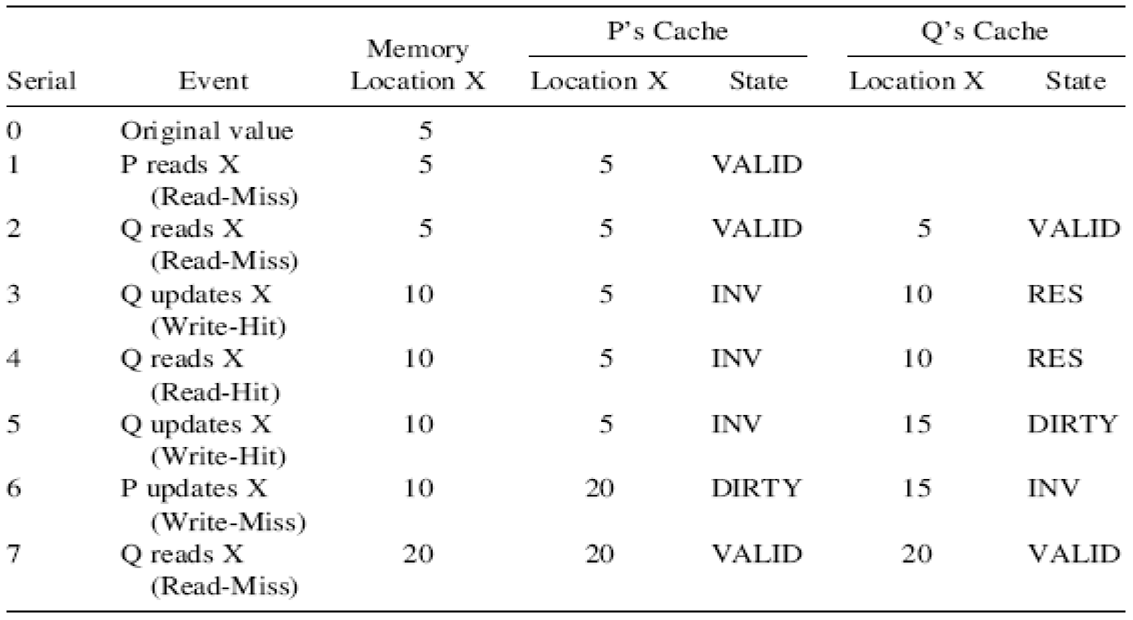
Figure 4: Goodman’s Write-Once Protocol State Diagram [98]



State	Description
Invalid [INV]	The copy is inconsistent.
Valid [VALID]	The copy is consistent with global memory.
Reserved [RES]	Data have been written exactly once and the copy is consistent with global memory. There is only one copy of the global memory block in one local cache.
Dirty [DIRTY]	Data have been updated more than once and there is only one copy in one local cache. When a copy is dirty, it must be written back to global memory.

Event	Actions
Read-Hit	Use the local copy from the cache.
Read-Miss	If no Dirty copy exists, then supply a copy from global memory. Set the state of this copy to Valid. If a dirty copy exists, make a copy from the cache that set the state to Dirty, update global memory and local cache with the copy. Set the state to VALID in both caches.
Write-Hit	If the copy is Dirty or Reserved, perform the write locally, and set the state to Dirty. If the state is Valid, then broadcast an Invalid command to all caches. Update the global memory and set the state to Reserved.
Write-Miss	Get a copy from either a cache with a Dirty copy or from global memory itself. Broadcast an Invalid command to all caches. Update the local copy and set its state to Dirty.
Block replacement	If a copy is in a Dirty state, it has to be written back to main memory if the block is being replaced. If the copy is in Valid, Reserved, or Invalid states, no write-back is needed when a block is replaced.

Table 7: Write-Once Protocol [97]



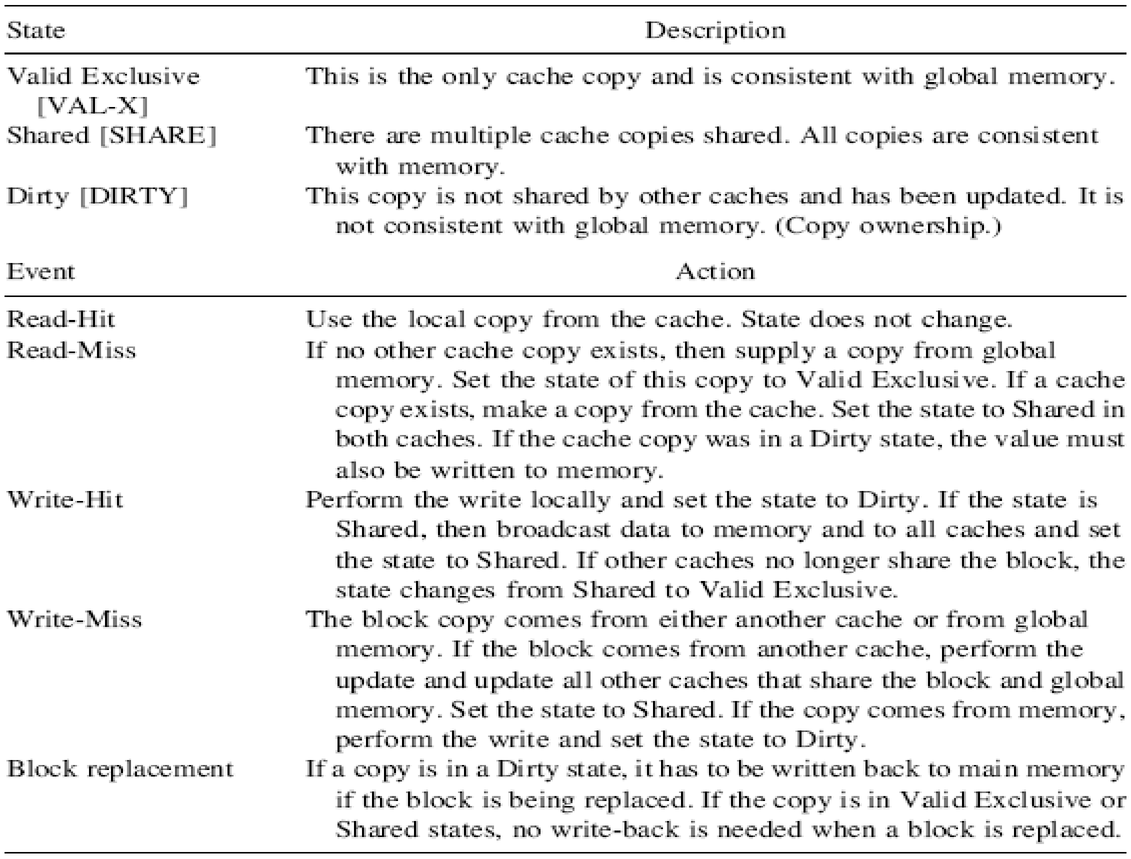
			P's Cache		Q's Cache	
Serial	Event	Memory Location X	Location X	State	Location X	State
0	Original value	5				
1	P reads X (Read-Miss)	5	5	VALID		
2	Q reads X (Read-Miss)	5	5	VALID	5	VALID
3	Q updates X (Write-Hit)	10	5	INV	10	RES
4	Q reads X (Read-Hit)	10	5	INV	10	RES
5	Q updates X (Write-Hit)	10	5	INV	15	DIRTY
6	P updates X (Write-Miss)	10	20	DIRTY	15	INV
7	Q reads X (Read-Miss)	20	20	VALID	20	VALID

Table 8: Example (Write-Once Protocol) [97]

Write-Update and Partial Write-Through

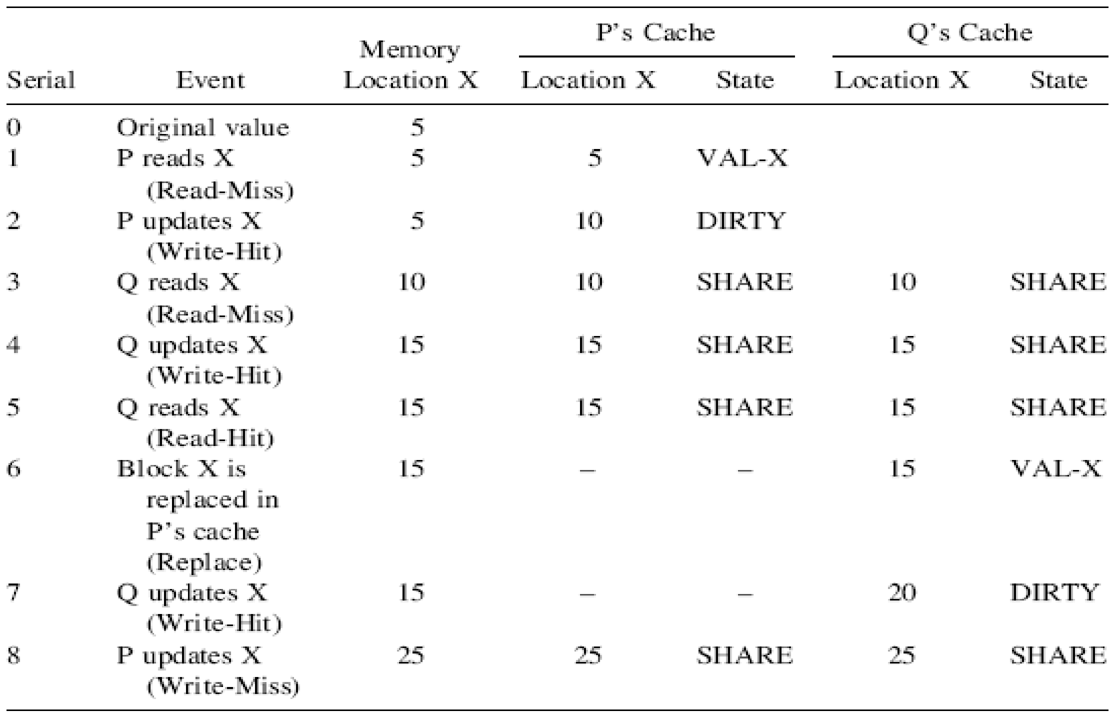
This protocol the update to one cache is written to memory, also it is broadcast to other caches which share the updated block. These caches carry out the snooping protocol on the bus and perform updates to their local copies. Available is also a special bus line, what this does is that it indicates that at least one cache is being shared by the block. [97]

The following tables (tables 9 and 10) are descriptions of the write-update and write-through protocols, with an example in table 10.



State	Description
Valid Exclusive [VAL-X]	This is the only cache copy and is consistent with global memory.
Shared [SHARE]	There are multiple cache copies shared. All copies are consistent with memory.
Dirty [DIRTY]	This copy is not shared by other caches and has been updated. It is not consistent with global memory. (Copy ownership.)
Event	**Action**
Read-Hit	Use the local copy from the cache. State does not change.
Read-Miss	If no other cache copy exists, then supply a copy from global memory. Set the state of this copy to Valid Exclusive. If a cache copy exists, make a copy from the cache. Set the state to Shared in both caches. If the cache copy was in a Dirty state, the value must also be written to memory.
Write-Hit	Perform the write locally and set the state to Dirty. If the state is Shared, then broadcast data to memory and to all caches and set the state to Shared. If other caches no longer share the block, the state changes from Shared to Valid Exclusive.
Write-Miss	The block copy comes from either another cache or from global memory. If the block comes from another cache, perform the update and update all other caches that share the block and global memory. Set the state to Shared. If the copy comes from memory, perform the write and set the state to Dirty.
Block replacement	If a copy is in a Dirty state, it has to be written back to main memory if the block is being replaced. If the copy is in Valid Exclusive or Shared states, no write-back is needed when a block is replaced.

Table 9: Write-Update Partial Write-Through Protocol [97]



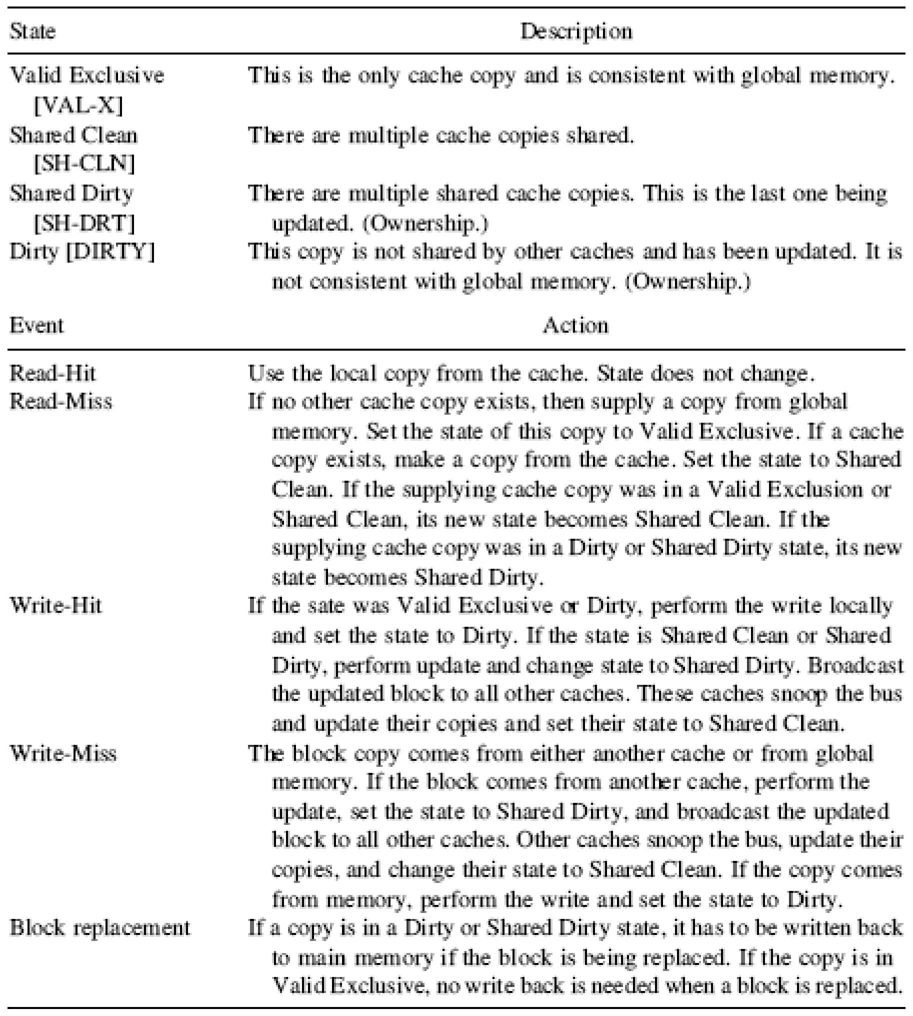
		Memory	P's Cache		Q's Cache	
Serial	Event	Location X	Location X	State	Location X	State
0	Original value	5				
1	P reads X (Read-Miss)	5	5	VAL-X		
2	P updates X (Write-Hit)	5	10	DIRTY		
3	Q reads X (Read-Miss)	10	10	SHARE	10	SHARE
4	Q updates X (Write-Hit)	15	15	SHARE	15	SHARE
5	Q reads X (Read-Hit)	15	15	SHARE	15	SHARE
6	Block X is replaced in P's cache (Replace)	15	–	–	15	VAL-X
7	Q updates X (Write-Hit)	15	–	–	20	DIRTY
8	P updates X (Write-Miss)	25	25	SHARE	25	SHARE

Table 10: Example (Write-Update Partial Write-Through) [97]

Write-Update and Write-Back

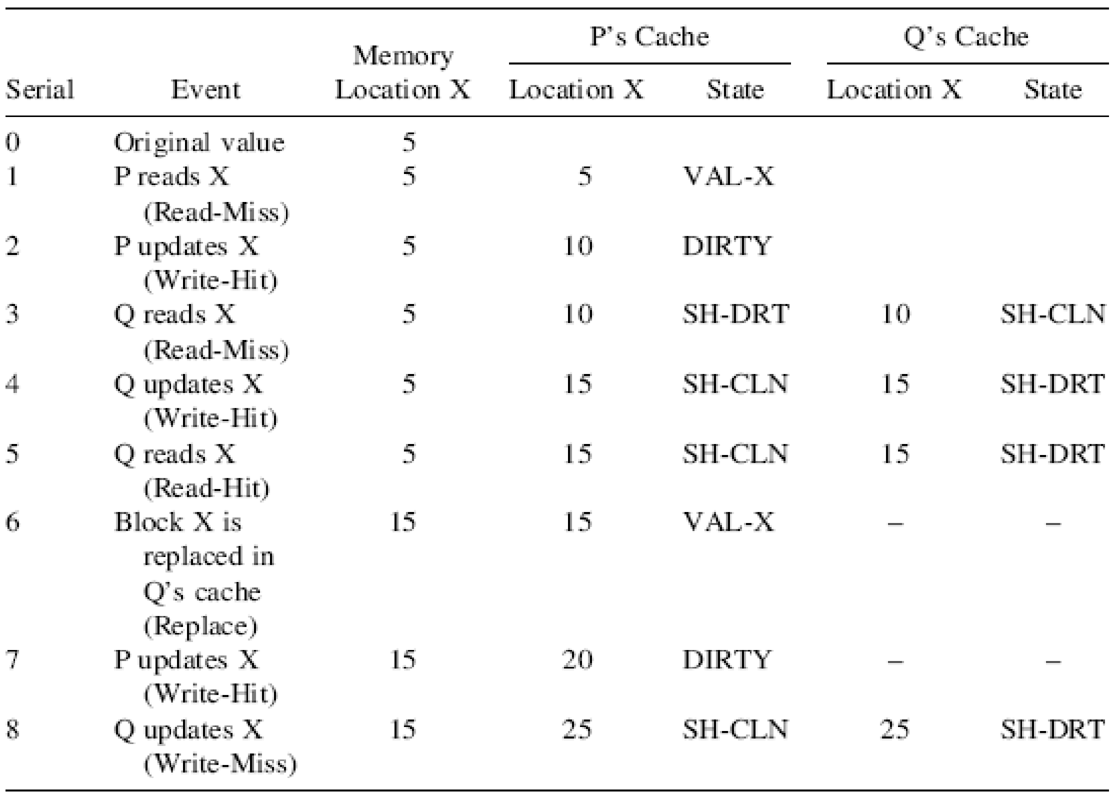
This protocol is very similar to the write-update and write-through protocol, the only difference would be that instead of a write-through it is using a write-back update. The memory blocks are updated whenever they are shared.

Table 11 lists the protocols/states used in this protocol. Table 12 is an example.



State	Description
Valid Exclusive [VAL-X]	This is the only cache copy and is consistent with global memory.
Shared Clean [SH-CLN]	There are multiple cache copies shared.
Shared Dirty [SH-DRT]	There are multiple shared cache copies. This is the last one being updated. (Ownership.)
Dirty [DIRTY]	This copy is not shared by other caches and has been updated. It is not consistent with global memory. (Ownership.)
Event	Action
Read-Hit	Use the local copy from the cache. State does not change.
Read-Miss	If no other cache copy exists, then supply a copy from global memory. Set the state of this copy to Valid Exclusive. If a cache copy exists, make a copy from the cache. Set the state to Shared Clean. If the supplying cache copy was in a Valid Exclusion or Shared Clean, its new state becomes Shared Clean. If the supplying cache copy was in a Dirty or Shared Dirty state, its new state becomes Shared Dirty.
Write-Hit	If the sate was Valid Exclusive or Dirty, perform the write locally and set the state to Dirty. If the state is Shared Clean or Shared Dirty, perform update and change state to Shared Dirty. Broadcast the updated block to all other caches. These caches snoop the bus and update their copies and set their state to Shared Clean.
Write-Miss	The block copy comes from either another cache or from global memory. If the block comes from another cache, perform the update, set the state to Shared Dirty, and broadcast the updated block to all other caches. Other caches snoop the bus, update their copies, and change their state to Shared Clean. If the copy comes from memory, perform the write and set the state to Dirty.
Block replacement	If a copy is in a Dirty or Shared Dirty state, it has to be written back to main memory if the block is being replaced. If the copy is in Valid Exclusive, no write back is needed when a block is replaced.

Table 11: Write-Update Write-Back Protocol [97]



Serial	Event	Memory Location X	P's Cache Location X	P's Cache State	Q's Cache Location X	Q's Cache State
0	Original value	5				
1	P reads X (Read-Miss)	5	5	VAL-X		
2	P updates X (Write-Hit)	5	10	DIRTY		
3	Q reads X (Read-Miss)	5	10	SH-DRT	10	SH-CLN
4	Q updates X (Write-Hit)	5	15	SH-CLN	15	SH-DRT
5	Q reads X (Read-Hit)	5	15	SH-CLN	15	SH-DRT
6	Block X is replaced in Q's cache (Replace)	15	15	VAL-X	–	–
7	P updates X (Write-Hit)	15	20	DIRTY	–	–
8	Q updates X (Write-Miss)	15	25	SH-CLN	25	SH-DRT

Table 12: Example (Write-Update Write-Back) [97]

Summary

We discussed snooping protocols, such as write-invalidate and write through, write-invalidate and write-back, write-once, write-update and partial write-through, and write-update and write-back. These help the memory keep its memory updated so as to keep cache coherence, depending on which protocol is being implemented.

Paper1 Design of an Adaptive Cache Coherence Protocol for Large scale Multiprocessors

Paper2 Effects of Cache Coherency in Multiprocessors

Lecture #14a: Cache Coherence Directory Based Protocols

Course:	Computer Architecture III
Lecturer:	Miodrag Bolic
Scribe:	Michel Bray and Marc Branchaud
Date:	Friday, October 27th 2006

Introduction

The topic of this lecture is directory-based protocols for cache coherence. Cache coherence problems occur in multiprocessor systems. Cache coherence refers to the integrity of data stored in local caches of a shared resource. [100] Whenever a processor equipped with data cache accesses a variable, the value of this variable is copied into the processor's cache. The coherence problem occurs when two or more processors access the same data element and at least one processor modifies its value. If no cache coherence protocols are implemented different processors are not made aware of the modifications made to the data and different processors keep using an invalid value of the variable, this in turn means that the final results of the program will be invalid.

Two general categories of cache coherence protocols exist; they are "Snoopy protocols" and "Directory protocols". Snooping is the process where the individual caches monitor address lines for accesses to memory locations that they have cached. When a write operation is observed to a location of which a cache has a copy, the cache controller invalidates its own copy of the snooped memory location. [100] A directory-based protocol uses hard-ware to keep track of shared variables. When shared variables are accessed and/or modified by different processors, different bits are set or reset in the directory to indicate whether the cached data is valid or not.

Explanation

2 What is a directory based protocol?

A directory based protocol is a hardware based scheme used to eliminate cache coherency problems. A list of every processor that holds the same instances of certain blocks of data is kept in a directory which in turn is kept in shared memory. An entry of a block in a directory will hold pointers to each cache that holds a copy of the data and a dirty bit that permits or denies a cache to update the block. [101]

3 When to use a directory based protocol?

When snoopy protocols are used in networks, the broadcasting techniques used in these protocols can become costly. For example: when a snoopy protocol is used in a 2D mesh, the update information is sent to every processor in the mesh creating high traffic in the interconnection network, hindering efficiency. [102]

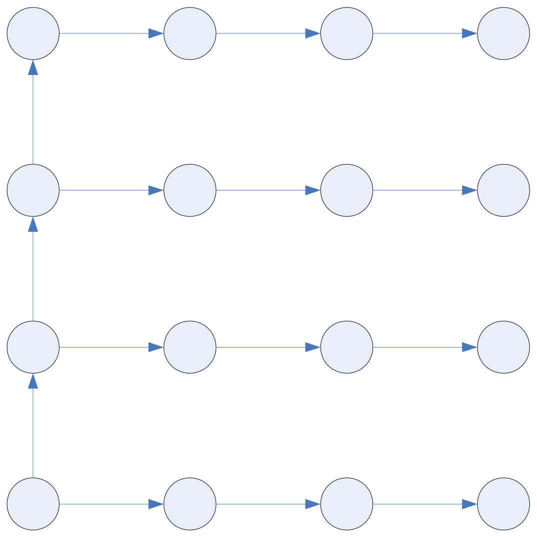


Figure 1: Broadcast signal through a 2D mesh. [103]

On the other hand, directory based protocols send information to specific processors listed in the block entry of the directory. This protocol discourages the use of the entire interconnection network in a multi-processor environment. In our previous example of the 2D mesh, the update information is sent only to the processors that hold a copy of the block, which stops the use of the complete interconnection network when it is not needed. [101]

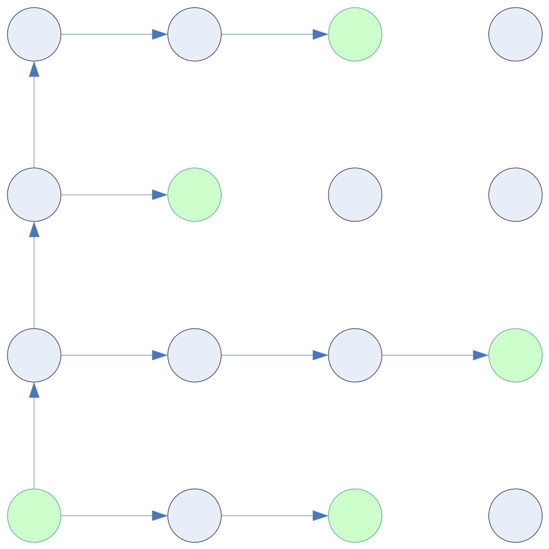


Figure 2: A directory based protocol working through a 2D mesh. The green circles contain a copy of the same variable. [103]

4 Types of directory protocols
Directory-based cache coherence protocols can be classified in one of two general categories; they are either “centralized” or “distributed”; [101] both of these have the common goal of reducing traffic on the interconnection network. In order to achieve this goal both categories can support multiple shared copies of a memory block. [101]

4.1. *Centralized Directory Protocols*
As the name implies these protocols use a single directory to track shared memory content. This category of protocol can be implemented in more than one way.

4.1.1 *Full-map Protocol*
This protocol implements a directory containing information on every cache entry. In such an implementation of the centralized directory protocol every directory entry contains “presence bits” and a “single-inconsistent bit”.

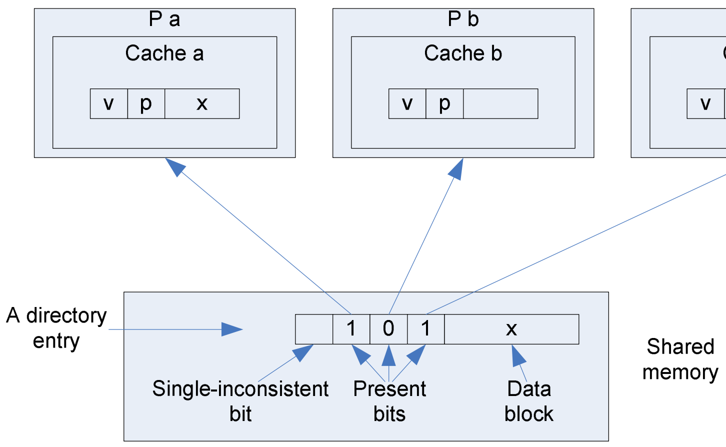


Figure 3**:** Full map protocol directory [101]

The directory contains one presence bit associated with every available cache. When a presence bit is set to ‘1’ it means that the associated cache contains a copy of the data block associated with the directory entry. In the case that one and only presence bit is set to ‘1’ then the single-inconsistent bit can be set to ‘1’ this means that only the associated processor has writing privileges to that memory element.

Read miss
In the event that the single-inconsistent bit is set, the memory sends an update request to the cache with the private bit set. The private cache then sends the information to the requesting cache, clears its own private bit and the single-inconsistent bit is cleared in the block entry. [101]

Whether or not the single-inconsistent bit is set, the requesting cache is sent a copy of the block and the present bit is set for the cache. Once the cache receives the copy, the valid bit is set and the private bit is cleared. [101]

Write miss

When a write miss is occurs, all other caches are invalidated and their present bits are cleared. The invalidated caches will then send acknowledgements to the memory. During the acknowledgements, if one of the newly invalidated caches happens to have its private bit set, then the memory block is updated with that cache's data. Once all acknowledgements are received, the requesting cache's present bit is set as well as the single-inconsistent bit. Finally, the cache receives a copy of the information, modifies it and the valid and private bits are set. [101]

Write hit
In the case where the private bit is clear, the requesting cache sends a privacy request to the memory. Other caches are invalidated similar to the write miss. The single-inconsistent bit is set and an acknowledgement is sent from the memory to the requesting cache. The cache's private bit is then set.

4.1.2 *Limited Directories*
The limited directory protocol solves the potential directory size problem by setting a fixed directory size. This means that there are a fixed number of pointers in the directory regardless of the number of processors. This limits the number of copies found in multiple caches and therefore reduces the size of the directory and the search time. [102]

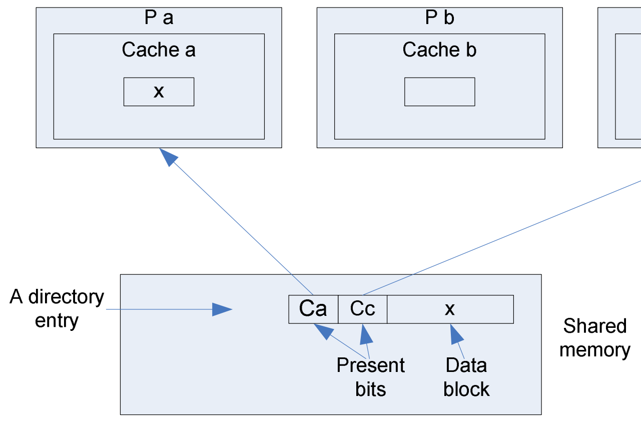


Figure 4: Limited directory (based on figure 3 [101])

Limited directories are similar to full-map directories in the way that they handle a read miss, a write miss and a write hit. The only difference is that when a cache requests a copy of a block but there is no room for a new pointer in the entry, then a previous pointer is replaced by the new pointer. When this happens, the block in the old cache is set to invalid. The pointer to be replaced is chosen by a replacement policy to the designer's discretion. [101]

4.2 *Distributed Directory Protocols*
The distributed directory protocols have the same functions as the centralized directory protocols. However the distributed directory protocols differ from their centralized counterparts in the way the directory is implemented. These protocols partition and spread the directory among the caches and/or memories. This helps reduce the directory sizes and memory bottlenecks in large multiprocessor systems. [101]

4.2.1 *Hierarchical Directory Protocols*

This type of directory protocol is mostly used in systems comprised of a set of connected clusters. Each cluster contains a set of processing units and a directory connected by an interconnection network. A request that cannot be serviced by the caches within a cluster is sent to the other clusters as determined by the directory. [101] This type of architecture is like a group of centralized directories.

4.2.1.1 *Centralized Directory Invalidate*

This protocol is one possible implementation of a distributed directory protocol. More specifically it uses a hierarchical implementation of the directory.

When this protocol is used and a write-request is issued within a cluster, the cluster's centralized directory establishes which processors have a copy of the memory block to which the processor wishes to write. Once the directory has established which processor has a copy of the data block, invalidating signals and a pointer to the requesting processor are forwarded to all processors that have a copy of the block. [102] The invalidating signal ensures that only the writing processor has a valid copy of the memory block.

Write-miss

When a write miss request is received the directory establishes which processors have a copy of the memory block and proceeds to send the appropriate processors an invalidating signal as well as a pointer to the requesting processor. Once the invalidating signal is received by the processors they invalidate themselves and send an acknowledgement signal to the requesting cache. When the invalidating process is complete only the requesting cache will have read-write access to the memory block.

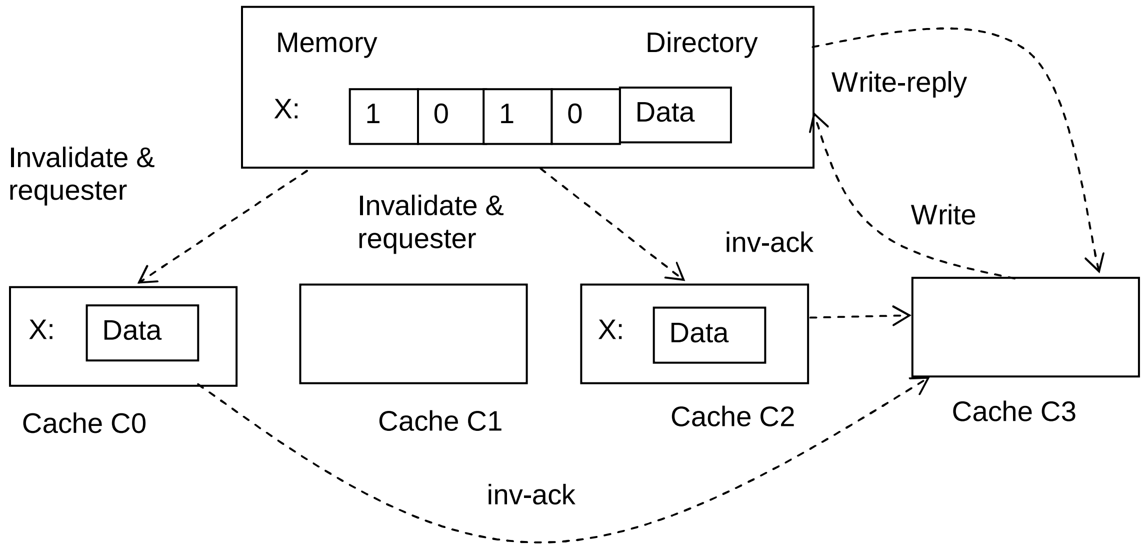


Figure 5: A write operation performed by processor 3 [102]

4.2.2 *Chained directory protocols*

Chained directories are similar to full-map directories since there is an unlimited amount of caches that can hold a copy of the block entry. To solve the directory size problem that is present in full-map directories, a chained directory makes use of a singly or doubly linked list meaning the directory holds a single pointer to a cache. That cache then holds a pointer to the next cache until a cache is found with a terminator pointer (CT). [102] The directory itself will only hold one pointer which is the head of the list, the first cache with the copy of the information.

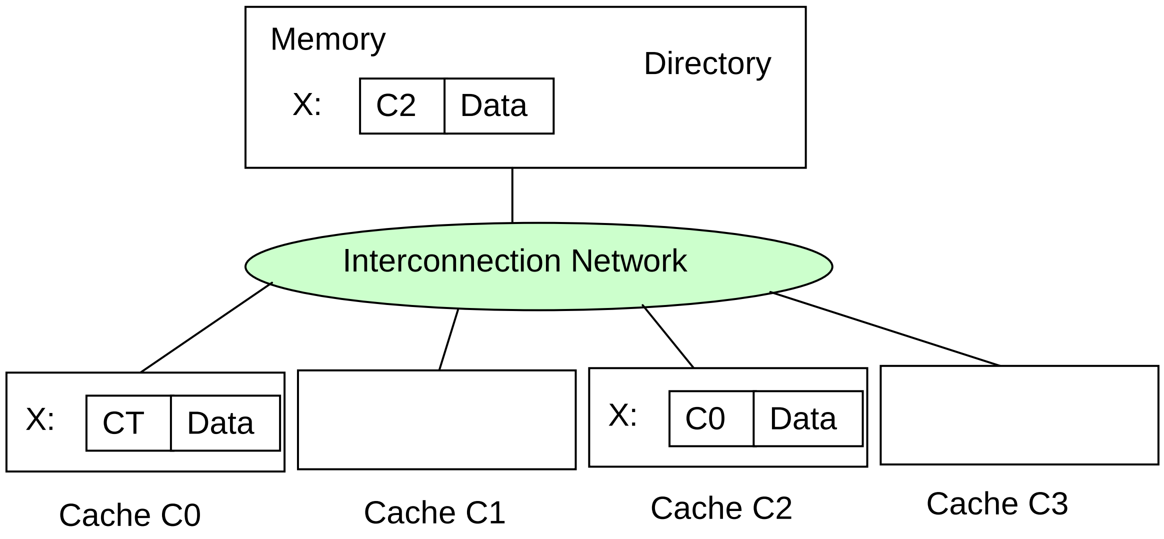


Figure 6: Chained Directory [102]

4.2.2.1 ***Scalable Coherent Interface (SCI)***
Scalable coherent interface protocols are doubly linked lists that fall under the categorization of chained directory. Every block address found in the directory; whether it is in shared memory or in a cache, contains additional tag bits. Tag bits in the memory contain a pointer to the first cache with the block, also called the head. The tag bits in the cache contain the next and previous list entries. [102]

Read miss
Before the block is cached, meaning there is no head pointer in the directory, the block in memory is in the uncached state and cached copies are invalid. The cache requesting the memory is sent a copy of the information. The directory is put into cached state and the head pointer in the directory points to the cache requesting the information. [101]

If the block is already cached and a new cache requests the block, the head pointer is set to the new cache requesting the block. The backwards pointer on the new head cache is set to the block in memory. A request is then sent to the old head cache, which sets its backwards pointer to the new head cache and sends its data to the cache.[102]

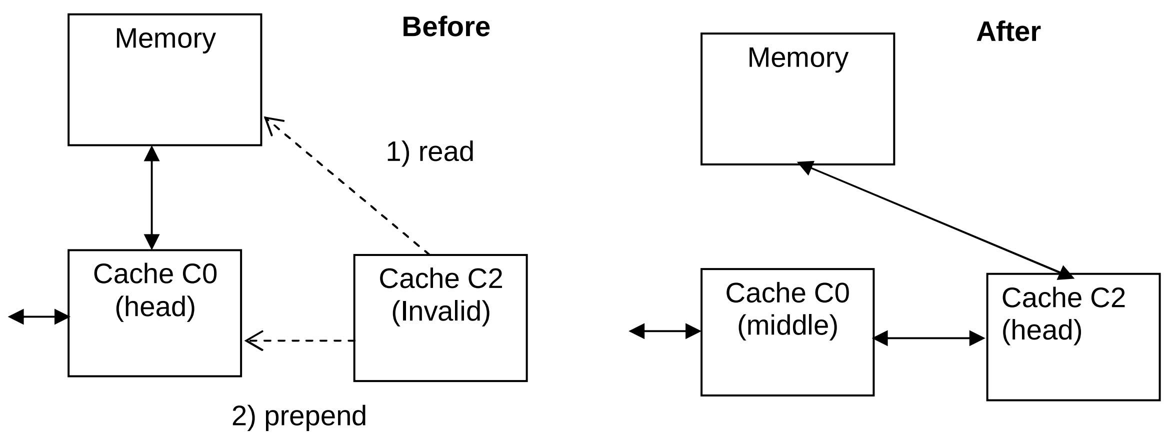


Figure 7: Sharing list addition [102]

Write miss
When a cache requests a write-miss to the memory, the head pointer in the directory is set to the requesting cache. The old head cache sets its forward pointer to the new head cache. The new head cache sets its backwards pointer to the memory block. The new head cache then proceeds to send an invalidate request through its forward pointer, invalidating the next cache. That cache, once invalidated, sends an invalidated request to the next cache and so on until every cache except the head cache is invalidated. Only once every cache is invalidated does the new head cache write the information. [101]

Write hit
If the cache requesting the write is the only cache in the linked list, then the write will proceed immediately. [101]

When the cache is already the head cache, every other cache becomes invalidated. After the invalidation, the head cache is written. This is similar to the write miss algorithm. [101]

When the cache is already the head cache, every other cache becomes invalidated. After the invalidation, the head cache is written. This is similar to the write miss algorithm. [101]

If the cache is not the head cache, then the cache will be detached from the linked list to place itself next to the current head cache. The cache requesting the write becomes the new head cache. Similarly to the write miss and the previous write hit, the new head cache will invalidate all the other caches before writing the new information.

In the case of every successful write, the size of the linked list will always be reset to 1.

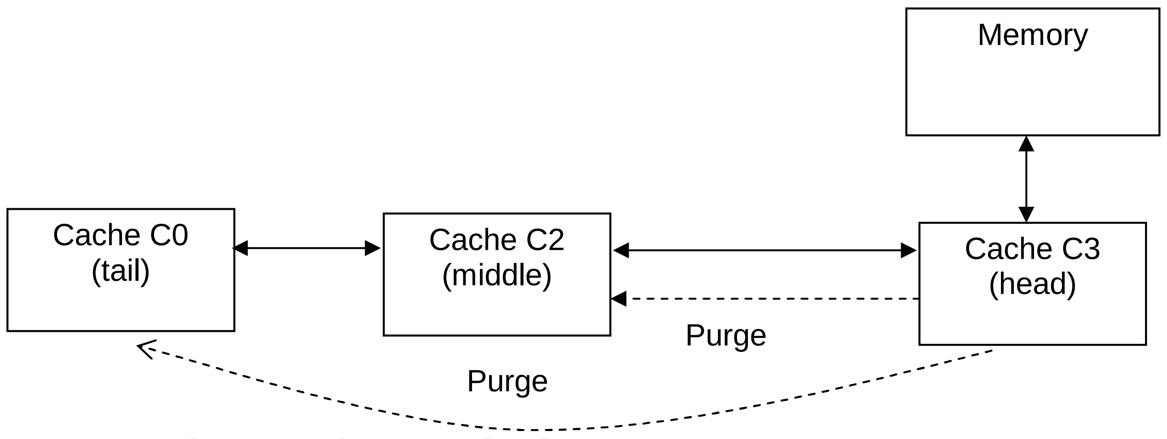


Figure 8: Head purging other entries [102]

4.2.2.2 *Stanford Distributed Directory (SDD)*
This protocol is similar to the SCI protocol however this one is based on a singly linked list implementation of a chained directory. As with the SCI protocol memory points to the head of the sharing list. However since this protocol is based on a singly linked list each processor can only point to the previous processor in the chain and cannot point to the next processor. When the sharing list must be extended or shortened these functions are handled differently than from the SCI protocol.

Read-miss
When a read-miss is encountered by a processor, a read-miss signal is sent to the memory. Upon receiving this signal the memory proceeds to updating its head pointers to contain the address of the requesting processor. After having updated the head pointer, the memory then sends a read-miss-forward signal to the old head. When the old head receives this message it sends the requested information to the new head along with its own address. [102] The new head then proceeds to copy the data and uses the address to aim its pointer towards the old head of the sharing list.

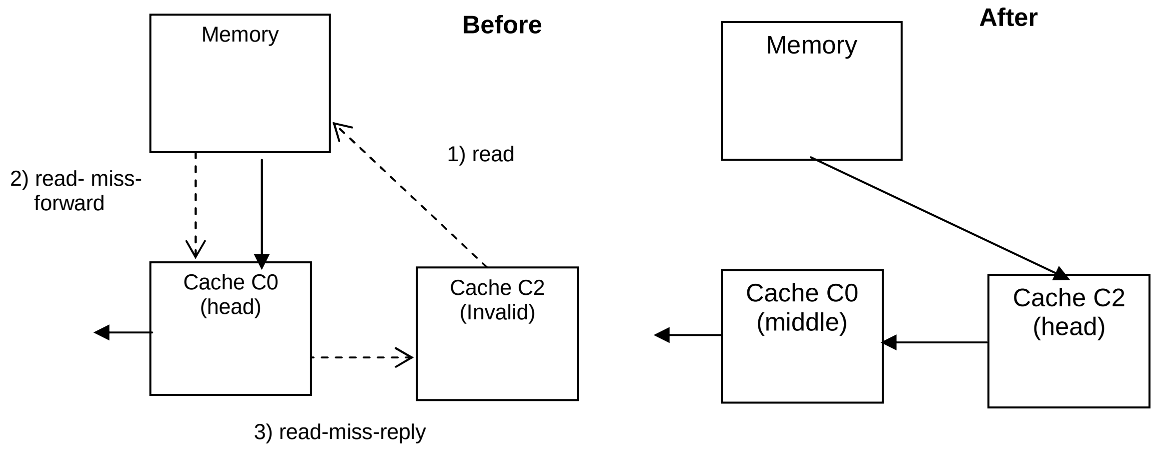


Figure 9: Sharing list addition [102]

Write-miss

In the event of a write-miss request, the requesting processor sends a write-miss message to the memory. Upon receiving this message the memory proceeds aiming its pointer to the requesting processor. A write-miss-forward message is also sent to the old head. When such a message is received the old head sends a write-miss-reply-data message to the requesting processor. This processor also proceeds to invalidating itself and forwards the write-miss-forward message to the next processor on the list who repeats the process of invalidating itself and forwards the same message to the next processor in the list. When the last processor of the list receives the message is proceeds to invalidating itself and sends a write-miss-reply message to the requesting processor. When the requesting processor has received both the write-miss-reply-data and the write-miss-reply the write-miss process is completed. [102]

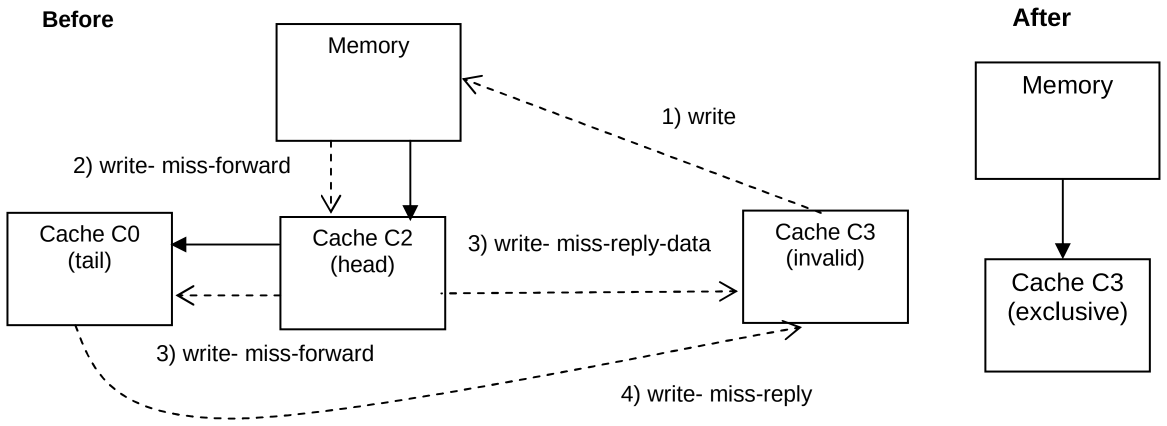


Figure 10: Write miss sharing list removal [102]

Summary

The cache coherence problem is of great importance when dealing with shared memory multiprocessing systems. Fortunately as discussed in this document many solutions were developed in order to solve the coherency problem. These solutions are valid regardless of the architecture of the system.

The full map directory protocol would be the easiest one to implement however it is very costly in terms of memory size. Furthermore, when using a full map directory we run the risk of having a large amount of unused resources since not every processor will be sharing every variable therefore large portions of the directory would simply be set to ‘0’. This implementation also has the disadvantage of not being very scalable since every time a new processor is added to the system the directory size has to be increased.

The limited directory protocol helps reduce the cost of the memory. This protocol achieves this by having a limited amount of data slots for directory entries. This protocol also helps with the problem of scalability when using this implementation. If a new processor is added to the system the directory does not require any modifications. This is possible since processes are already in place to deal with missing data slots in the directory. Furthermore each slot is not associated with a single processor but rather contains the address of the processor in question.

Distributed directory protocols further reduce the cost of memory and increase the scalability of the system. These protocols partition and spread the directory among the available caches and memories. When using these protocols the directory contains the addresses of the processors.

Each directory protocol has its own way of dealing with read and write operations. Unlike their snooping counterparts the directory-based protocols do not rely on broadcasting signals to function thus reducing traffic on the interconnection network and therefore increasing the system’s efficiency. [102]

Paper 1: Cache coherence for large scale shared memory multiprocessors

Paper 2: An Evaluation of Directory Schemes for Cache Coherency

Lecture #14b: Cache Coherence

Course: Computer Architecture III
Lecturer: Miodrag Bolic
Scribe: Benoit Beauvais(3065159)
Date: October 15, 2007

Introduction

Most of the multiprocessors system today includes shared memory to better communicate with each other. However, by being able to read and write to the main memory it might happen that different values of the same data are located in different cache. To solve this, cache coherence protocols are implemented. To be able to understand better cache coherence, this lecture is going to describe first how writing to the cache is done. After, the directory based protocol will be explained in details with the full-map category, limited category and chained category.

Explanation

1. Write policies

When a processor is writing to a shared memory system, multiple behaviors can occur depending on the policy implemented in the system. The first write policy is the write through policy. The processor first writes its data to the cache then it automatically updated the value in the main memory. This policy is easy to implement because it doesn't need to maintain the dirty bit. Also, the cache remains consistent with the main memory. The other policy is the write back policy. This one mainly focuses on performance because it doesn't write to the main memory when the cache is updated. It only updates the main memory when a block in the cache is being replaced. However, this comes at the cost of the data integrity because it is possible that the cache and the memory do not contain the same information at a given time.]removed example and added explanation. Figure 1 shows both policies. X' is the newest data in the memory and X is the older data.

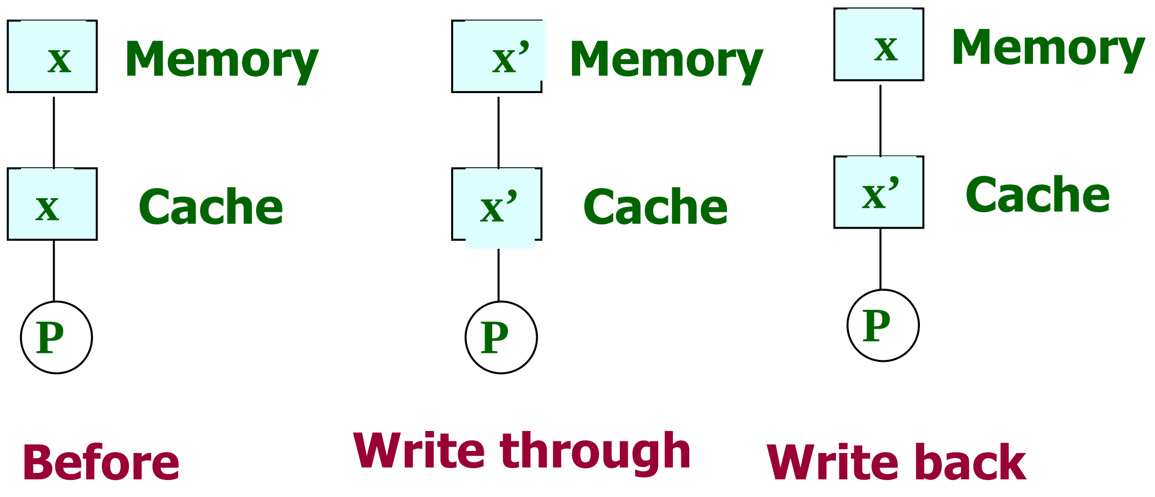


Figure 1- Write policy [106]

Directory based protocol

There are two main protocols to provide cache coherence: the snooping and directory based protocol. The snooping protocol broadcast information between caches and the main memory to provide cache coherence. This protocol will not be covered removed . Directory based protocols are a set of cache coherence cache coherence protocols, that is, protocols which ensure cache coherence or memory coherence between multiple nodes of multiprocessor or distributed shared memory systems[111]..A directory is a data structure that maintains information on the processors that share a memory block and on its states [107]. Instead of broadcasting to all caches like the snooping protocol, the directory based protocol selects only the needed cache. This improves the performance in the case where there are a lot of processors not requiring the data. There are three types of directory based protocols: full-map directories, limited directories and chained directories.

Centralized vs. distributed directory

The directory can be centralized or distributed. When centralized, the directory contains all the cache states and presence bits in one location. This could lead to a bottleneck problem where there is a large search to be made. Thus, centralized directory does not scale well. Figure 2 shows a centralized directory protocol. In a distributed directory, each memory contains a directory. Figure 3 shows a distributed directory base protocol.

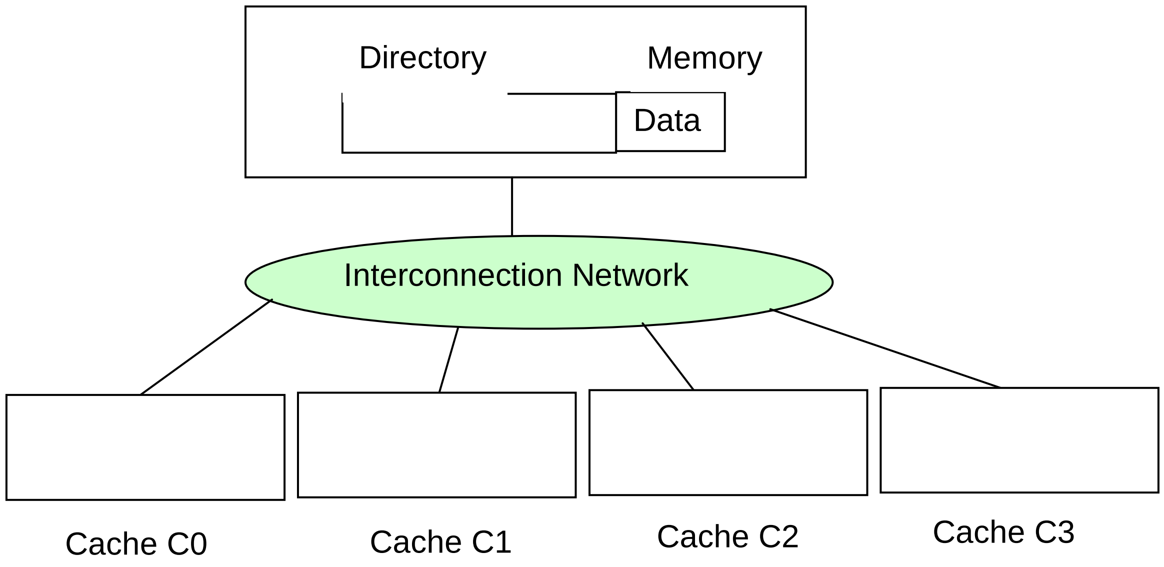


Figure 2: Centralized directory [106]

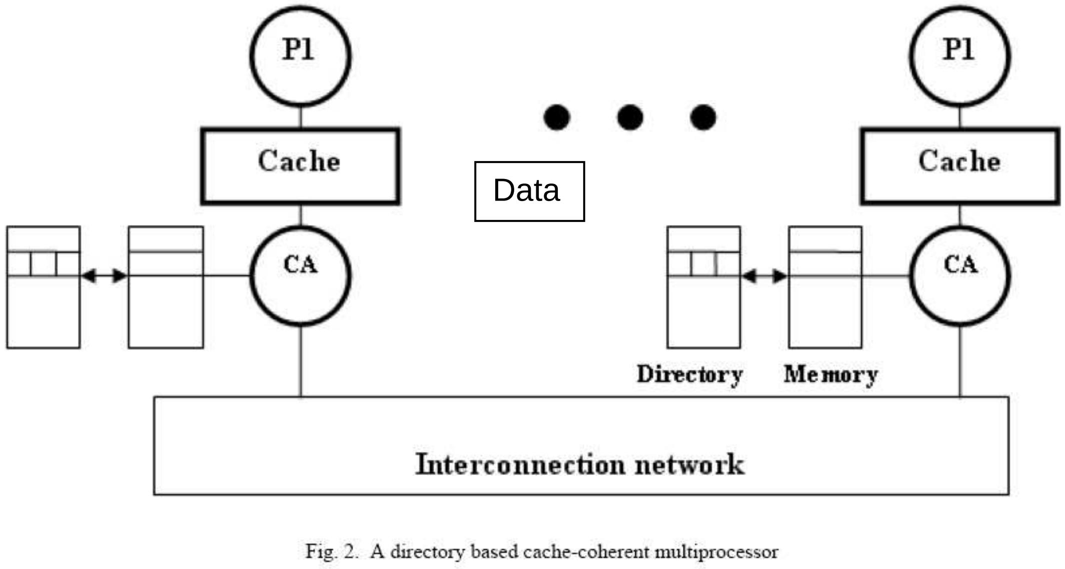


Figure 3: Distributed directory [109]

Full-map directories

The principal of a full-map directory is that the directory holds all the information about who holds t the data in their cache. The first bit of an entry in a directory contains the state of the memory block. This can be empty, copied or dirty. A state is empty when there is no data copied to its caches. While a state copied means that when a processor make a read, the data is copied to its cache and the state of the memory block is changed to "copied". When a processor writes to the shared memory, the memory module issue an invalidation request to the other processor containing the data. The processors acknowledge the invalidation request and the shared memory marks the block as dirty. There are also presence bits that indicate which caches hold a copy of the data. Figure 4 shows the relationship between the directory and the shared memory in full-map directories while figure 5 shows the connection between different caches in that type of directories. Also, a big disadvantage of full-map directories is that it doesn't scale well because it contains all the cache information.

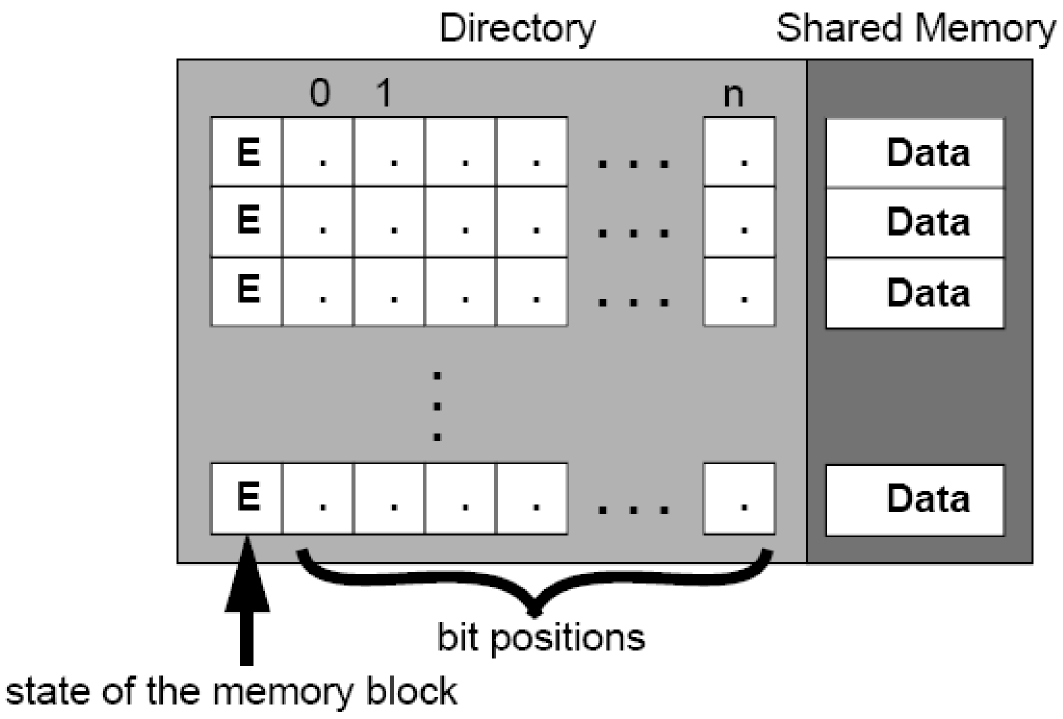


Figure 4: Full-map directory [110]

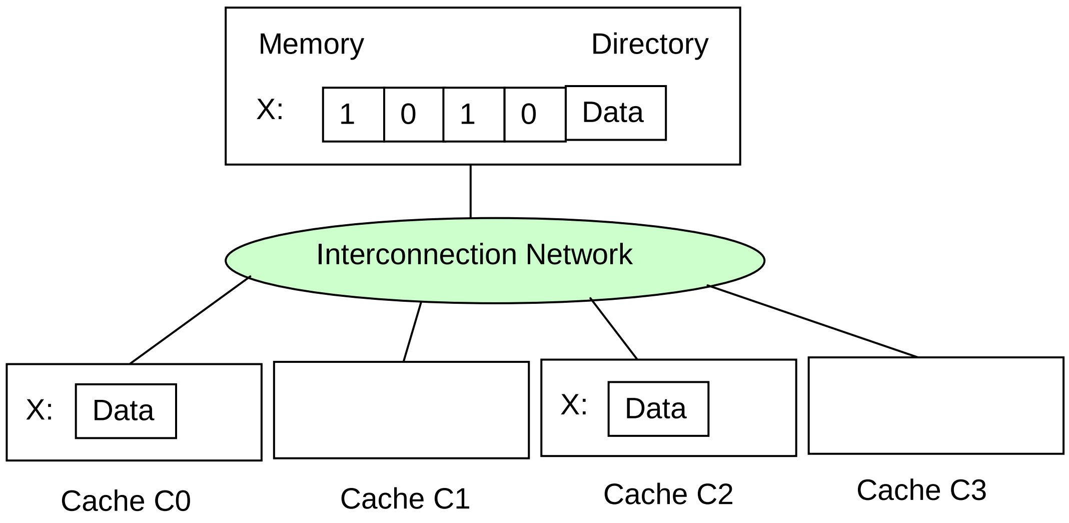


Figure 5: Full-map directory protocol [106]

Limited directories

To solve the scalability problem of the full-map directories, a limited directory is used. This directory has a limited amount of cached copies at the same time by using pointers to caches. If we take figure 7 example, there are two pointers in the directory one pointing to C0 and the other to C2. If C1 would need a copy of X in his cache, it would make a request to the memory and the memory would change the directory pointer of C0 or C2 to C1. Let's change C2 pointer to C1 for this example. If C1 modified X, then only C0 and C1 would be updated since C2 is no longer associated to the directory. Also, like full-map directory the state bit would be turned into dirty. C2 would need to read the memory to have the updated value and the directory would return back to copied state. Figure 6 shows the association between the directory and the shared memory in the limited directories protocol.

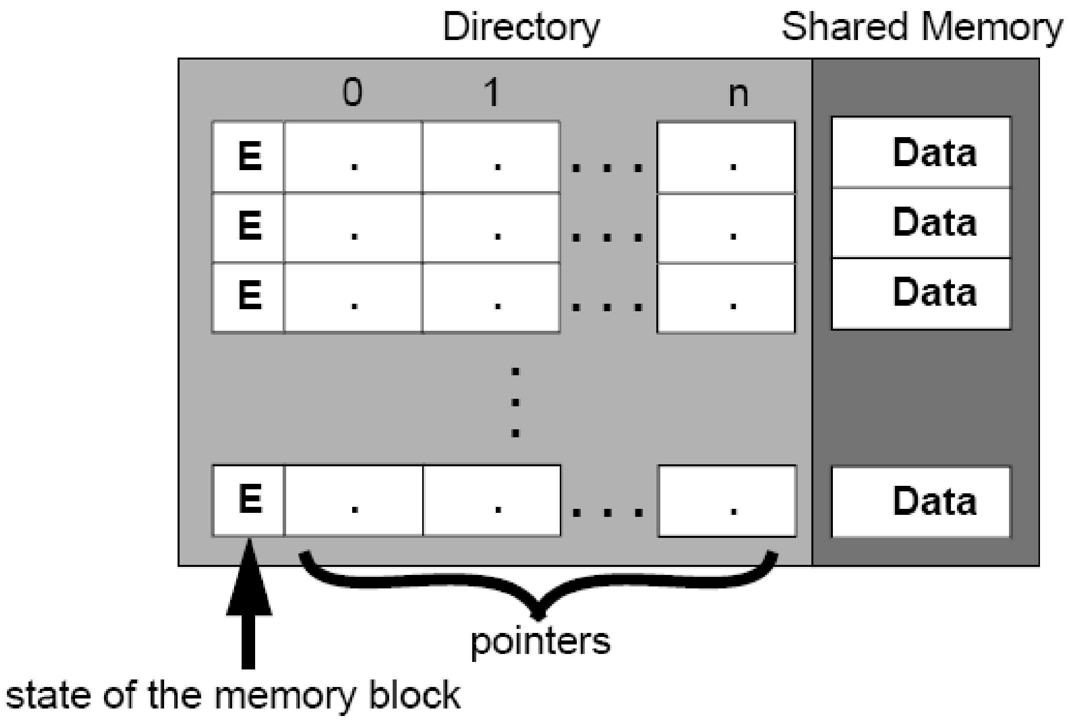


Figure 6: Limited directory [110]

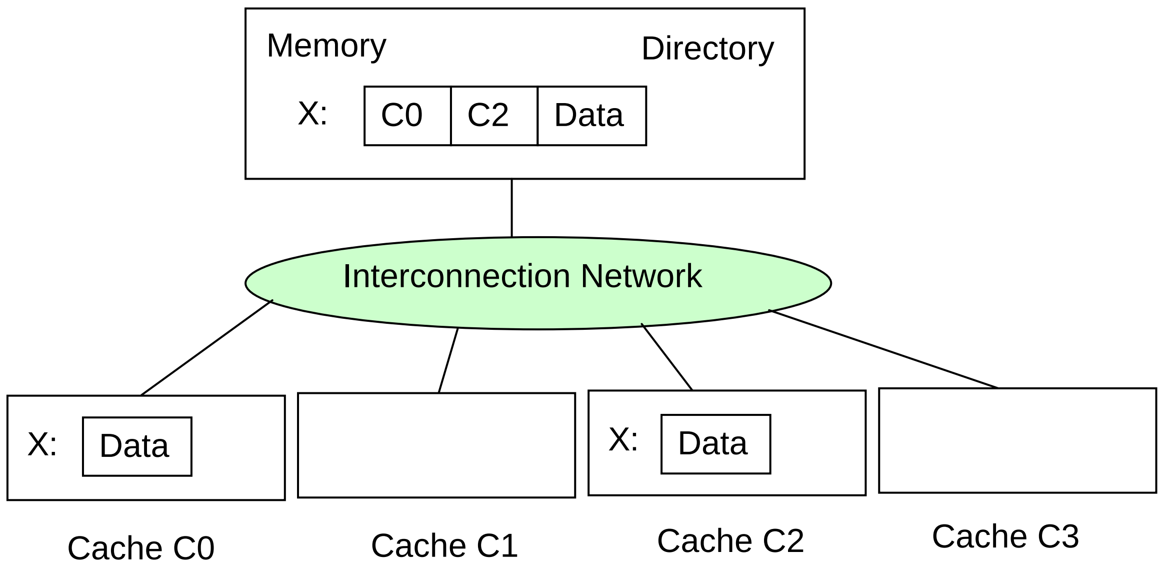


Figure 7: Limited directory protocol[106]

Chained category

The problem with the limited protocol is that it has a restricted number of copies in the directory. The chained directory protocol improves the scalability by letting caches handle pointers. The first pointer is in the directory and it points to the latest cache who issued a read operation. Then that cache points to the previous one who read the content in the memory and so on until the last one has a pointer that points to a terminator. Figure 8 shows how the pointers are linked with the shared memory and figure 9 shows how the caches are pointing to each other. The write operation is a little bit more complicated in this protocol. If processor 3 for example write to location X then C3 indicates the block as read only, issues a write request and stalls P3. Then the shared memory sends an invalid request to all the caches using the pointers. Once this is done, the shared memory receives an invalidation acknowledgement, marks the block as dirty and sends write permission for processor 3[110]. One of the drawbacks is that it's a sequential walk-through.

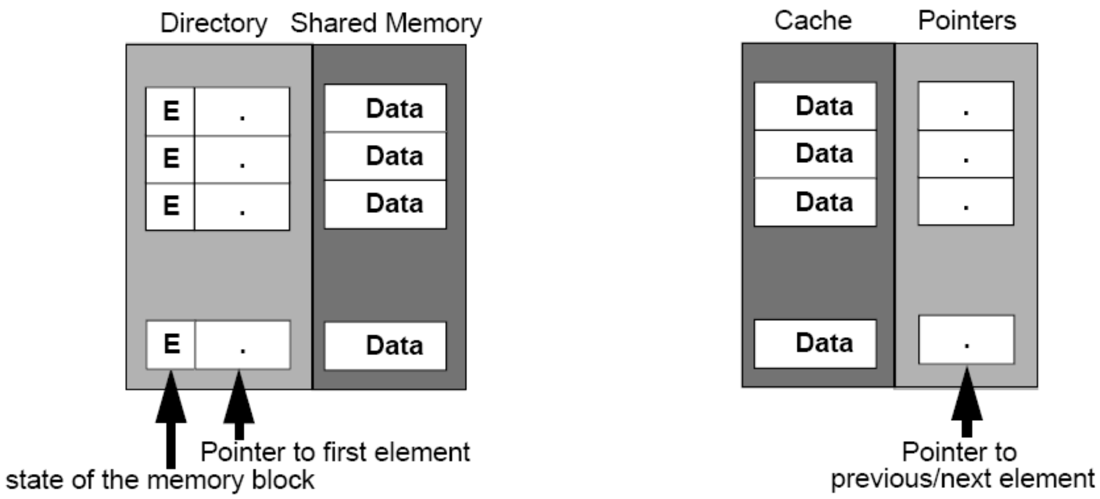


Figure 8: Chained directory protocol [110]

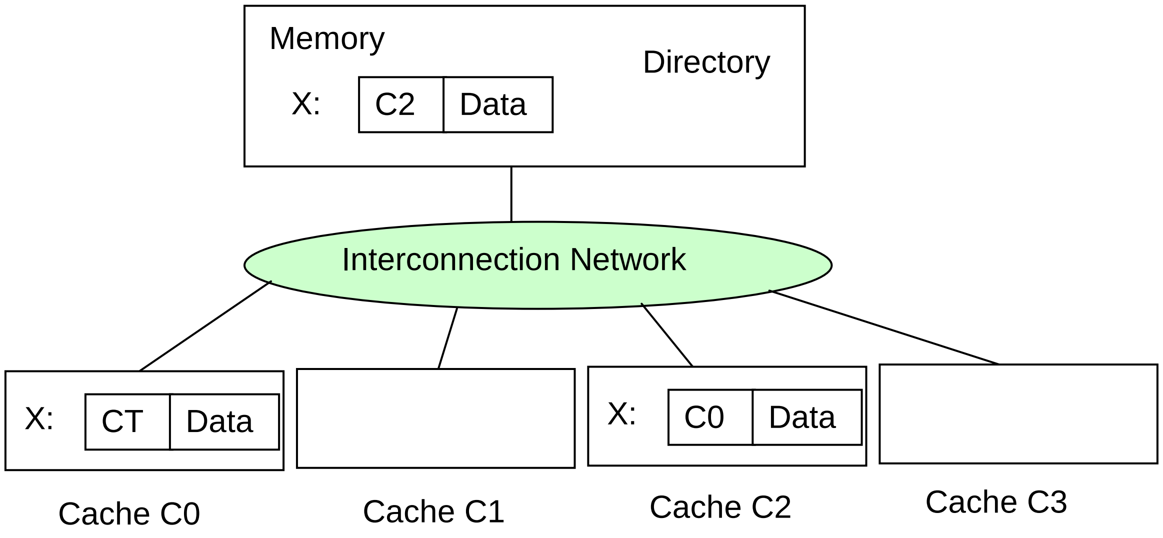


Figure 9: Chained directory protocol [106]

Scalable Coherent Interface (SCI)

The scalable coherent interface protocols are based on a doubly linked list of distributed directory [106]. We can easily identify caches by adding a tag bit to every caches and memory. The tags tell who is the previous and next element in the list. In figure 10, the directory state is initially set to “uncached” while all the caches are set to invalidate. Once a cache tries to read the memory it set the cache previous tag to the memory and next tag to the old head. The memory set its tag to "cached". Another feature is that the head can have the read/write priority by purging the other caches. Figure 11 shows this process.

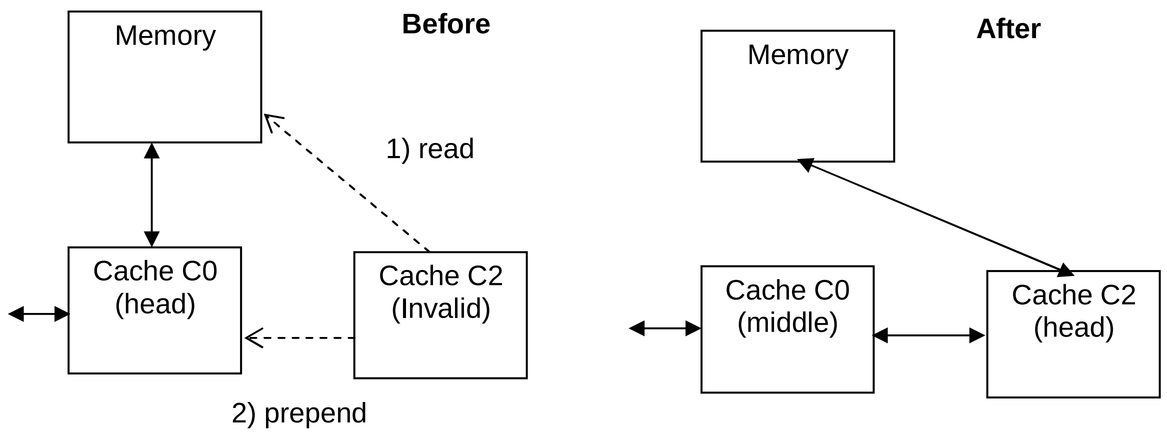


Figure 10: Sharing list addition(SCI) [106]

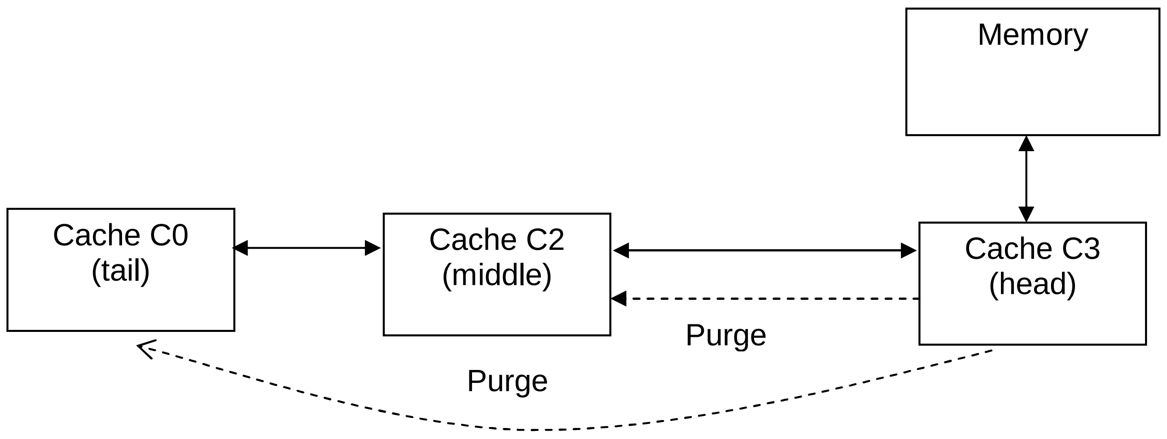


Figure 11: Head purging(SCI) [106]

Stanford Distributed Directory (SDD)

The Stanford distributed directory protocol is based on a singly linked list of distributed directories [106]. This is similar to SCI where each cache contains a tag to the previous element in the list. Unlike the SCI it doesn't contain tag to the next cache in the list. Because of that, adding and removing caches to the list are handled differently. When there is a read-miss, the requester sends a read-miss message to the memory[106]. The memory then updates his status and sends that signal to the old header. Once the old header receives the signal, it sends the data with the address as a read-miss reply. Then the requester become the new head and point to the to old header. Figure 12 shows this.

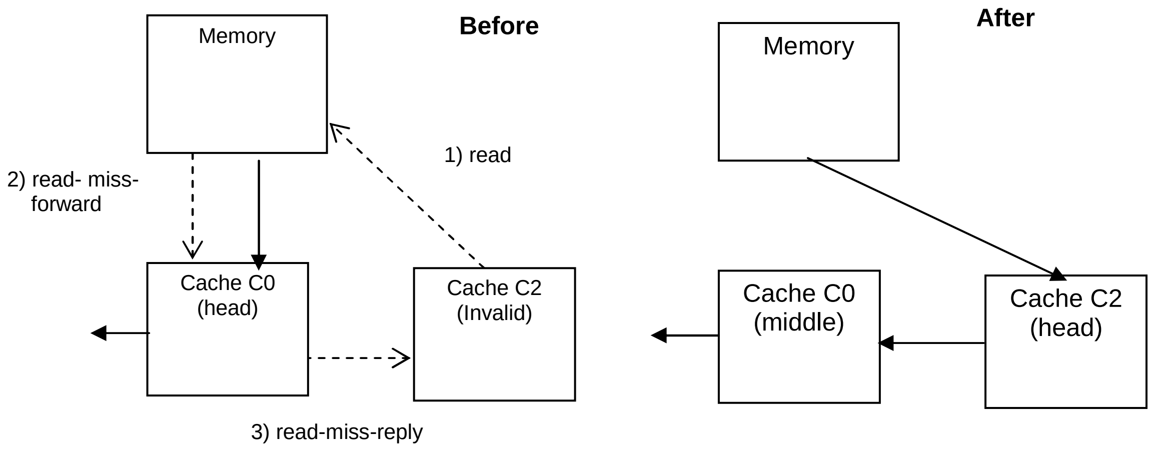


Figure 12: Sharing list addition (SDD) [106]

For a write-miss situation, the process is similar then the read-miss except that once the old header receives the write-miss request it forwards them to the other caches and invalidate itself. Once the other caches receive that signal, it sends a reply to the new header and invalidates itself also. This way the new header has exclusive read/write. Figure 13 shows this process.

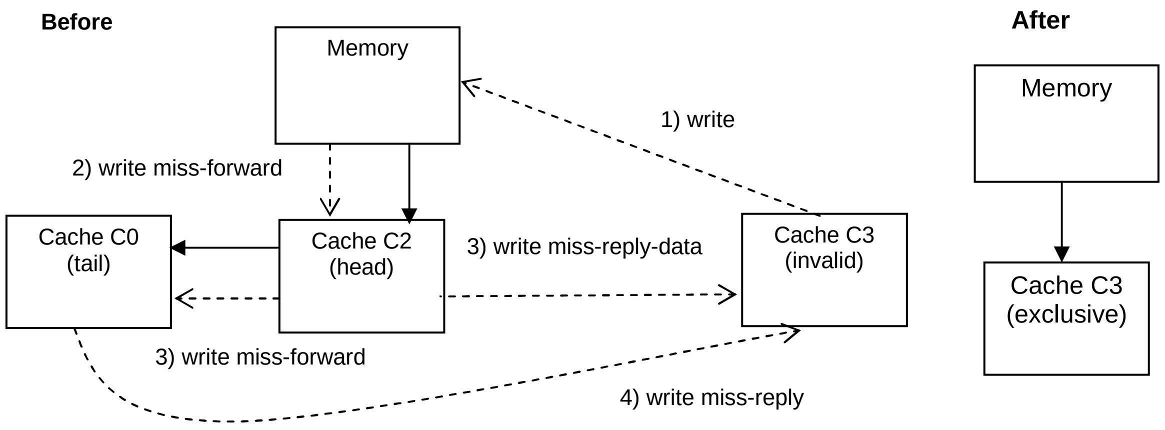


Figure 13: Write miss sharing list removal (SDD) [106]

Summary

This lecture first covered writing policies. There are two policies: write through and write back. The write through policy states that when a processor is updating its cache then it must update also the shared memory. On the other hand, the write back policy only updates its cache when the processor is writing to it. The main advantage of write back is performance because it doesn't need to write to the shared memory. However, it can have poor integrity.

The other subject invoked in this lecture was the directory based protocol. There are three different category of that protocol: the full-map directories, limited directories and chained directories. The full-map directories record all the presence bits of all the processors in its directory. This protocol doesn't scale well because it takes all the processors even the ones that doesn't have a copy of the data. The limited protocol has pointers that point to the cache that contains data. However, the number of pointers is limited, thus some cache can not be synchronized with the shared memory because it was bumped by another processor. The chained directories protocol puts a linked list with pointers inside each cache that has data copied in it. Each cache points to another cache until it find the terminator pointer. This protocol scales well but it has a sequential walk-through.

Paper 1 Cache Coherence for Shared Memory Multiprocessors Based on Virtual Memory Support

Lecture #15a: Message Passing Models

Course: Computer Architecture III
Lecturer: Miodrag Bolic
Scribe: Dimitri Elkodsi – 3218299 & Sadek Hamdan - 3597245
Date: October 15, 2007

Introduction

The message-passing model of parallel computation has emerged as an efficient and well-understood paradigm for parallel programming. The message passing model is one of several computational models for conceptualizing program operations providing alternative methods for communication and movement of data among multiprocessors (compared to shared memory multiprocessor systems). It is defined as a set of processes having only local memory, the processes communicate by sending and receiving messages while on the other, the transfer of data between processes requires cooperative operations to be performed by each process (a send operation must have a matching receive).

A message passing architecture is used to communicate data among a set of processors without the need for a global memory. The basis for the scheme is that each processor has its own local memory and communicates with other processors using messages. The elimination of the need for a large global memory together with its synchronization requirement, gives message passing schemes an edge over shared memory schemes.

In this lecture, we will eventually discuss different aspects of message passing systems including a software programming model, hardware implementation, examples of message passing systems, message passing interface (MPI), message passing routines and concluding with a comparison between MPI and OpenMP

Explanation

Model of a Message passing Multicomputer

Figure 1 shows the main components of a message passing multiprocessor architecture. There are n nodes in the figure and each node consists of a processor and a local memory. Each processor has its own address space. Nodes communicate with each other by links (called external channels) and via an interconnection network, normally a static-type network.

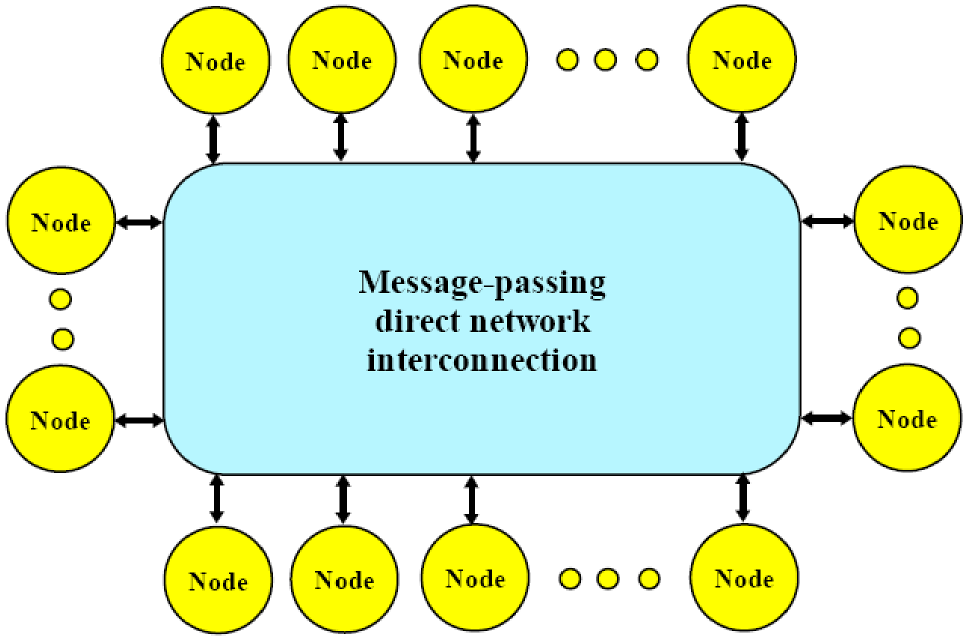


Figure 1: Message passing system[118]

Communication among nodes is typically realized via point to point or direct connections called links or channels. Such networks are called direct or static networks and can be represented by a communication graph in which vertices correspond to the nodes of the multicomputer and edges represent channels.

Generic node architecture

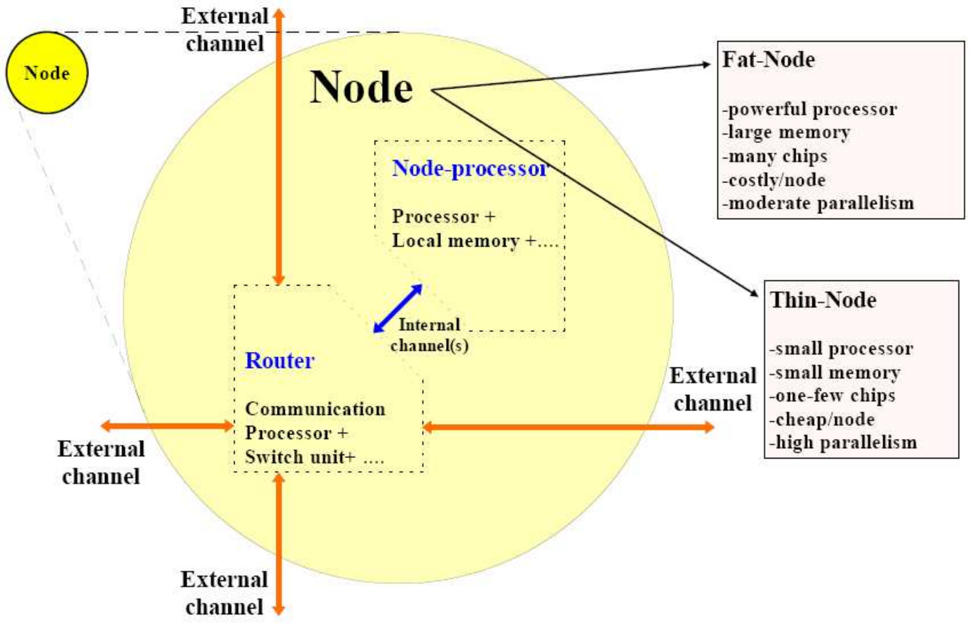


Figure 2: Generic node architecture[118]

A message passing system typically combines local memory and the processor at each node of the interconnection network. There is no global memory so it is necessary to move data from one local memory to another by means of message passing. This is typically done by send/receive pairs of commands, which must be written into the application software by a programmer.

Therefore, the structure of the node of message passing architectures consists of three main components as shown in the figure 2:
* Computation processor + private memory (PE)
* Communication processor
* Router (or switch unit)

Each node in this mesh network has a computation processor and a router. Each processor has access to its own local memory and can communicate with other processors using the direct network interconnection. The role of both the communication processor and the router is to organize communication among the nodes of the multicomputer.

2.2 Generic organization model

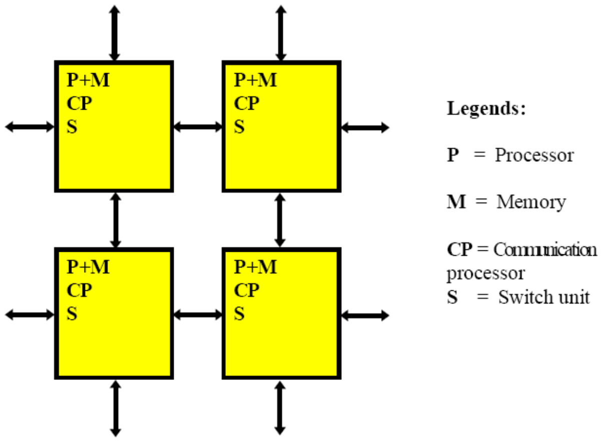


Figure 3a: First generation[118]

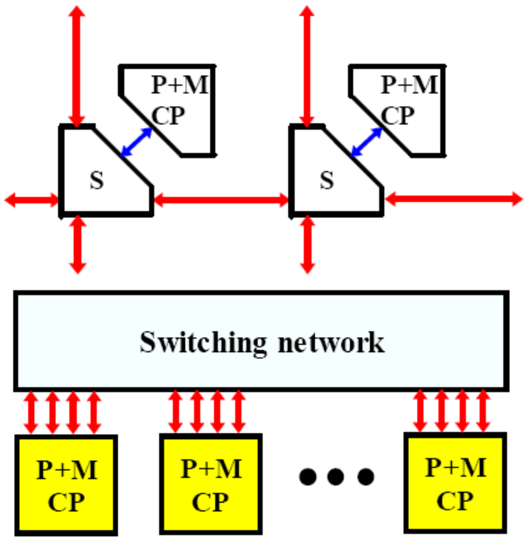


Figure 3b: Decentralized (2nd generation) [118] Figure 3c: Centralized (2nd generation) [118]

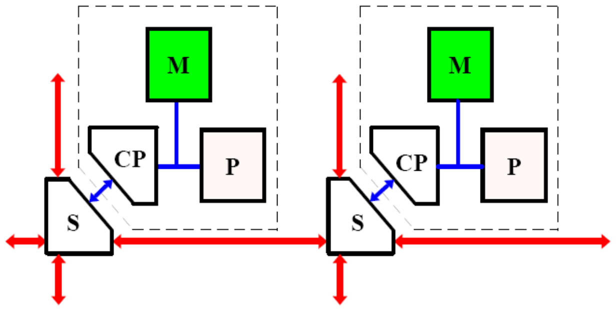


Figure 3d: Third generation[118]

Based on the organization of interconnection networks and nodes, three generations of multicomputers can be distinguished. In first generation message passing computers shown in figure 3a, the processing elements are directly connected, so there are no communication processors or switch units. Their functions are provided by the processing element via low level software layers.

Hence, in the second generation multicomputers independent switch units are introduced and separated from the processor. On one hand, the figure 3b represents the decentralized generic interconnection scheme. On the other, the figure 3c represents the centralized generic connection scheme. What's more, in the third generation message passing architectures shown in figure 3d; there are three main components and for each component there is a separate processor.

Software programming model

In the contrary of the shared memory system, the message passing machine handles the problems of communication between processors in a more explicit manner. In consequence, the processors will communicate by sending messages to each other as explained above, instead of communicating through shared variables. These messages and the programmer take care of all the coherency and synchronization problems inherent in a multiprocessor machine.

A message passing system has the following properties:
• Complete computer as building block, including I/O
• Programming model: directly access only private address space (local memory)
• Communication via explicit messages (send/receive)
• Communication integrated at I/O level, not memory system, so no special hardware
• Resembles a network of workstations which can actually be used as multiprocessor systems.

Message passing program

On one hand, each processor has SEND and RECEIVE functions available to the programmer. On the other, each processor runs a program that passes messages back and forth to finish a job.

In a message passing system, there is no shared space so the memory space is segmented for each processor. Concerning the transfer of data, we will be dealing with a movement of that data from one memory space to another as shown in Figure 4. So, if Processor 1 needs to have "Mydata," Processor 0 must send it to Processor 1, which places it in its own memory space.

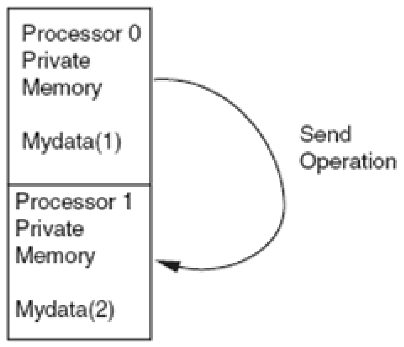


Figure 4: Transfer of data from one memory space to another. [112]

The transfer of data from one memory space to another will deal with a coherency problem, but since the programmer knows where the data was sent, he/she can make sure to update the data correctly. As a result, the programming of a message passing machine is much more difficult than traditional sequential programming.

Code example:

The following example is related to an array summing example for a message passing machine. The problems consist of adding n elements of an array in parallel.

The program is written using several languages, providing three additional operations: INITIALIZE - SEND- RECEIVE that allows processes to communicate with each other.

The INITIALIZE primitive assigns a number to each processor in the system, assigns the total number of processors. The SEND primitive takes a memory buffer and sends it to a destination node, as to other processors. The RECEIVE primitive accepts a message from a source node and stores it in a specified memory buffer. The basic programming model used in message passing architectures is based on the idea of matching a send request on one processor with a receive request on another. In such scheme, send and receive are blocking; that is, send blocks until the corresponding receive is executed before data can be transferred.

```
INITIALIZE; //assign proc_num and num_procs

if (proc_num == 0) //processor with a proc_num of 0 is the master,
//which sends out messages and sums the result
{
 read_array(array_to_sum, size); //read the array and array size
 //from file
 size_to_sum = size/num_procs;

 for (current_proc = 1; current_proc < num_procs; current_proc++)
 {
  lower_ind = size_to_sum * current_proc;
  upper_ind = size_to_sum * (current_proc + 1);
  SEND(current_proc, size_to_sum);
  SEND(current_proc, array_to_sum[lower_ind:upper_ind]);
 }

 //master nodes sums its part of the array
 sum = 0;
 for (k = 0; k < size_to_sum; k++)
  sum += array_to_sum[k];

 global_sum = sum;
 for (current_proc = 1; current_proc < num_procs; current_proc++)
 {
  RECEIVE(current_proc, local_sum);
  global_sum += local_sum;
 }

 printf("sum is %d", global_sum);
}
else //any processor other than proc_num = 0 is a slave
{
 sum = 0;
 RECEIVE(0, size_to_sum);
 RECEIVE(0, array_to_sum[0 : size_to_sum]);
 for (k = 0; k < size_to_sum; k++)
  sum += array_to_sum[k];
 SEND(0, sum);
}

END;
```

Array summing code [112]

The following will briefly explain the operation of each processor:

Proc 0

1- Sends the appropriate part of the array to be summed
2- It will perform local addition on its own portion of the array
3- Receive results from other processors
4- Print

Other processors:

1- They will receive the portion of the array to add
2- Add
3- Send the result on the appropriate partition.

As a result, we are dealing with an explicit communication, blocking applications; so the messages have to wait to be processed either receiving or sending the data.

Description of the program:

Proc 0:

1- Read all array, send the appropriate portion of the array for each processor in the network.
2- Perform local sums and store eventually in global sum.
3- Wait for the number of processors to send their own local result -> Receive operations
4- If Receive is blocking, it has to wait to a local sum to be receive, and the incremented sum will be added it to global sum

This programs requires different way of thinking, it consists of a blocking application, which decide in a way when to send the data (Send) and when to expect data (Receive)

Message passing architecture

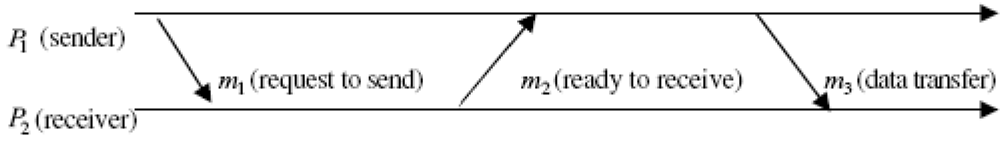


Figure 5: Blocking send/receive handshaking protocol. [113]

As we can realize from the figure 5 shown above that the implementation of the send/receive among processes requires a three-way protocol. In this case, the sending process issues a request-to-send message to the receiver process. The latter stores the request and sends a reply message back. Then, the sender process receives the reply and finally transfers the data.

We can realize that message passing implementations uses non-blocking operation in order to avoid the drawbacks of the three-phase protocol. In this case, send appears immediately to the user program. However, the message is buffered by the message layer until the network port is available. Only then, would the message be transmitted to the recipient. In there, the message is again buffered until a matching receive is executed.

Example implementing a blocking send/receive handshaking protocol

The following example shows message passing with two processors P1 and P2 which will let us see how data would be handled while working with blocking receives and blocking send.

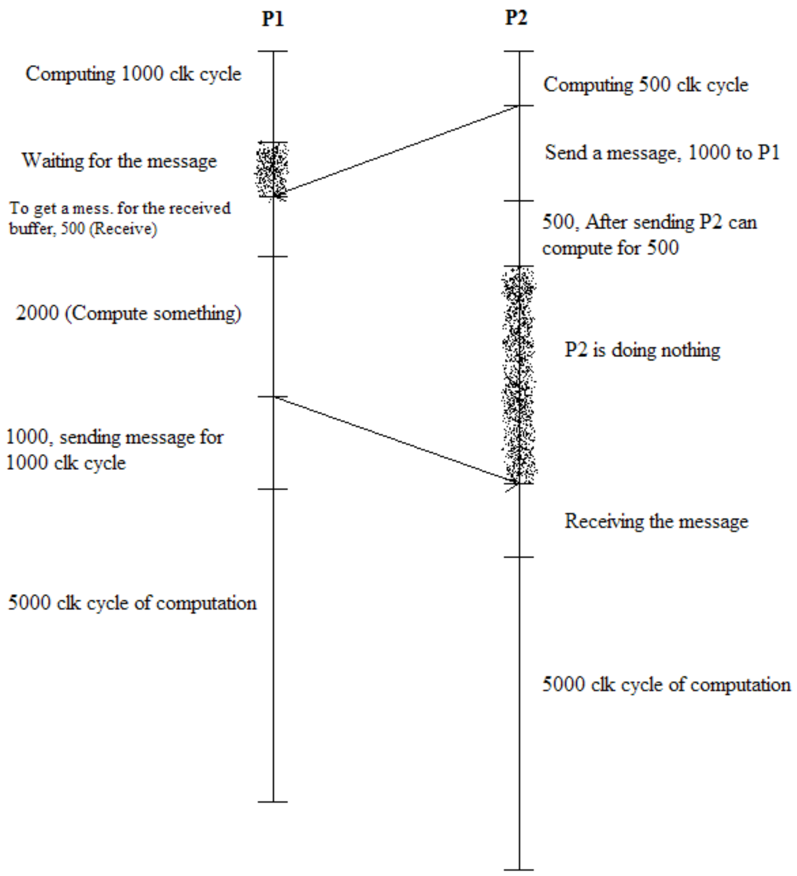

P1
P2
Computing 1000 clk cycle
Waiting for the message
To get a mess. for the received buffer, 500 (Receive)
2000 (Compute something)
1000, sending message for 1000 clk cycle
5000 clk cycle of computation
Computing 500 clk cycle
Send a message, 1000 to P1
500, After sending P2 can compute for 500
P2 is doing nothing
Receiving the message
5000 clk cycle of computation

Figure 6: Example of a Blocking send/receive protocol

Therefore there is 3000 cycles which are waiting in P2 (computed 10500 cycle), in the contrary of P1. Eventually, the problem is to allocate the resources which involve scheduling.

From the examples that we dealt previously, we can deduct that these programs accomplishes the same task as the shared memory program. In the Code example, the processor on the system that is the master (processor 0) sends out parts of the array to the other processors to sum.

As a result, the SEND and RECEIVE operations are blocking, so program execution cannot proceed until they are completed. Then the master waits for the processors to send the sum back, creates a global sum, and exits.

Message passing interface – MPI

Message Passing Interface (MPI) [113] is a specification of a library of routines for writing a practical, portable, efficient and flexible message passing program. In any parallel programming architecture especially the distributed memory parallel programming model, these routines can be used for data movement, global computation, and synchronization.

The reasons for using the MPI:

The reasons for using the MPI can be summarized as follows – more details are available in [114]:

- **Standardization** - MPI is the only message passing library which can be considered a standard. It is supported on virtually all high performance computers platforms. Practically, it has replaced all previous message passing libraries.

- **Portability** - There is no need to modify your source code when you port your application to a different platform that supports (and is compliant with) the MPI standard.

- **Performance Opportunities** - Vendor implementations should be able to exploit native hardware features to optimize performance. There is many factors which can affect an MPI application's performance:

    - *Platform / Architecture Related*: CPU clock speed, number of CPUs, memory and cache configuration, operating system characteristics

    - *Network Related:* Hardware such as routers, protocols (TCP vs. UDP).

    - *Application Related:* Algorithm efficiency and scalability, communication to computation ratios, load balance, memory usage patterns, I/O, message size used, types of MPI routines used: blocking, non-blocking, point-to-point, collective communications.

    - *MPI Implementation Related*: Message buffering, sender-receiver synchronization – polling/interrupt.

- **Functionality** - Over 115 routines are defined in MPI-1 alone.

- **Availability** - A variety of implementations are available, both vendor and public domain.

Groups and Communicators:

A group is an ordered set of processes. Each process in a group is associated with a unique integer rank. Rank values start at zero and go to N-1, where N is the number of processes in the group. In MPI, a group is represented within system memory as an object. A group is always associated with a communicator object.
A communicator is an object that encompasses a group of processes that may communicate with each other. All MPI messages must specify a communicator. In the simplest sense, the communicator is an extra "tag" that must be included with MPI calls. The concept of communicator is introduced in MPI to achieve this safe communication requirement. A communicator can be accessed via a handle of type MPI_COMM. The default communicator PI_COMM_WORLD contains all the MPI tasks.

Hello world example

On each processor the following program runs. At run time, you specify how many PEs you require and then your code is copied to each PE and run simultaneously.

At first the idea that the same code must run on every node seems very limiting. This is not at all the case as shown later.

The easiest way to see exactly how a parallel code is put together and run is to write the classic "Hello World" program in parallel. In this case it simply means that every PE will say hello to us. Let's take a look at the code to do this.

Hello World C Code:

```
#include <stdio.h>
#include "mpi.h"

main(int argc, char** argv)
{

  int my_PE_num;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_PE_num);
  printf("Hello from %d.\n", my_PE_num);
  MPI_Finalize();
}
```

Output of the previous example:

```
Hello from 5.
Hello from 3.
Hello from 1.
Hello from 2.
Hello from 7.
Hello from 0.
Hello from 6.
Hello from 4.
```

There are two issues here that may not have been expected. The most obvious is that the output might seem out of order. The response to that is "what order were you expecting?" Remember, the code was started on all nodes practically simultaneously. There was no reason to expect one node to finish before another. Indeed, if we rerun the code we will probably get a different order. Sometimes it may seem that there is a very repeatable order. But, one important rule of parallel computing is don't have to assume that there is any particular order to events unless there is something to guarantee it.

Message-passing routines

Environement management routines

MPI environment management routines [113] are used for an assortment of purposes, such as initializing and terminating the MPI environment, querying the environment and identity.
The most commonly used environment management routines are [114]:

MPI_Init: Initializes the MPI execution environment. For C programs, MPI_Init may be used to pass the command line arguments to all processes, although this is not required by the standard and is implementation dependent.

MPI_Comm_size: Determines the number of processes in the group associated with a communicator. Generally used within the communicator MPI_COMM_WORLD to determine the number of processes being used by your application.

MPI_Comm_rank: Determines the rank of the calling process within the communicator. Initially, each process will be assigned a unique integer rank between 0 and number of processors - 1 within the communicator MPI_COMM_WORLD. This rank is often referred to as a task ID. If a process becomes associated with other communicators, it will have a unique rank within each of these as well.

MPI_Finalize: Terminates the MPI execution environment.

P-T-P communication routines

MPI point-to-point operations between two processes can be implemented by send/receive technique by using the ranks of the two processes.
The MPI point-to-point messages can be in either blocking or non-blocking mode.

Blocking:

A blocking send routine will only "return" after it is safe to modify the application buffer (your send data) for reuse. Safe means that modifications will not affect the data intended for the receive task. Safe does not imply that the data was actually received; it may very well be sitting in a system buffer.
A blocking send can be synchronous which means there is handshaking occurring with the receive task to confirm a safe send.
A blocking send can be asynchronous if a system buffer is used to hold the data for eventual delivery to the receive.
A blocking receive only "returns" after the data has arrived and is ready for use by the program.
The routines for a blocking send and receive are:
MPI_Send (&buf,count,datatype,dest,tag,comm)
MPI_Recv (&buf,count,datatype,source,tag,comm,&status)
Where:

- &buf is a program (application) address space that references the data that is to be sent or received.
- count is the number of data elements of a particular type to be sent.
- datatype: char, int, float, double, long…
- source is the originating process of the message. Specified as the rank of the sending process. This may be set to MPI_ANY_SOURCE to receive a message from any task.
- destination is the process where a message should be delivered. Specified as the rank of the receiving process.
- tag identify a message. Send and receive operations should match message tags. For a receive operation, MPI_ANY_TAG can be used to receive any message regardless of its tag.
- comm is the communication context, or set of processes for which the source or destination fields are valid. The predefined communicator MPI_COMM_WORLD is used by default.
- &status is the source of the message and the tag of the message.

The program below uses the blocking communication mode:

```
#include <stdio.h>
#include "mpi.h"
main(int argc, char** argv)
{
        int my_PE_num, numbertoreceive, numbertosend=42;
        MPI_Status status;
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &my_PE_num);
        if (my_PE_num==0)
        {
                MPI_Recv( &numbertoreceive, 1, MPI_INT, MPI_ANY_SOURCE,
        MPI_ANY_TAG, MPI_COMM_WORLD, &status);
                printf("Number received is: %d\n", numbertoreceive);
        }
        else
                MPI_Send( &numbertosend, 1, MPI_INT, 0, 10, MPI_COMM_WORLD);
        MPI_Finalize();
}
```

Processor 0 do one blocking receive of an integer from any processor, in the parameters passed to MPI_Recv routine, MPI_ANY_SOURCE and MPI_ANY_TAG are used for the source and the tag. In this case processor 0 can receive any message from any processor in the MPI_COMM_WORLD communicator. After that, it prints the integer received.
Any other processor just sends an integer to processor 0.
At the beginning and before using any MPI routine, a MPI_Init is needed. Then, the routine MPI_Comm_rank described above to get the rank of the process, is used. And at the end a MPI_Finalize is needed.

Non-blocking

Non-blocking send and receive routines behave similarly; they will return almost immediately. They do not wait for any communication events to complete, such as message copying from user memory to system buffer space or the actual arrival of message.
Non-blocking operations simply "request" the MPI library to perform the operation when it is able. The user can not predict when that will happen.
It is unsafe to modify the application buffer (your variable space) until you know for a fact the requested non-blocking operation was actually performed by the library. There are "wait" routines used to do this.
Non-blocking communications are primarily used to overlap computation with communication and exploit possible performance gains.
The routines for a blocking send and receive are:
MPI_Isend (&buf,count,datatype,dest,tag,comm,&request)
MPI_Irecv (&buf,count,datatype,source,tag,comm,&request)
MPI_Wait (&request,&status)

Where:
&request: Since non-blocking operations may return before the requested system buffer space is obtained, the system issues a unique "request number". The programmer uses this system assigned "handle" later (in a WAIT type routine) to determine completion of the non-blocking operation.
The program below uses the non-blocking communication mode:

```
#include "mpi.h"
#include <stdio.h>
int main(int  argc, char *argv[])
{
int numtasks, rank, next, prev, buf[2], tag1=1, tag2=2;
MPI_Request reqs[4];
MPI_Status stats[4];

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

prev = rank-1; next = rank+1;
if (rank == 0) prev = numtasks - 1;
if (rank == (numtasks - 1)) next = 0;

MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[0]);
MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[1]);

MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]);

 { do some work }

MPI_Waitall(4, reqs, stats); MPI_Finalize();
}
```

This program is used in every processor of a ring network to send its rank and to receive the two ranks of the two neighbor nodes. Each processor gets the size of the group (the number of nodes in the network) and its rank using the environment management routines.
First the processor tries to receive two messages from the previous and the next node in the network with different tags to uniquely identify the message, and stores them in different elements of an integer buffer. The execution of the program does stop here since it is a non-blocking send.
After that the processor first sends its rank to the previous and the next nodes in the network with different tags to uniquely identify the message. The execution of the program does stop here since it is a non-blocking send.
The MPI_Waitall routine forces all the processors (4) to wait until all the messages has been sent and received.

Collective Communication Routines:

Collective operations in MPI are those operations that are applied to all members of a communicator's group. All processes are by default, members in the communicator MPI_COMM_WORLD, [113].

Types of Collective Operations:

Synchronization:
In some cases parallel tasks are required to synchronize with each other at a given point during the execution. Members of a group may need to wait at a synchronization point until all tasks reach the same point. Synchronization in MPI can be achieved using message passing and barrier operations.

Data Movement:
Broadcast: MPI_Bcast(buffer, n, data_type, root, communicator)
This function must be called by all members of the communicator's group using
the same arguments for the root and communicator. The contents of the root's buffer will be copied to the buffers of all tasks.

Collective Computation:
Global Combine (Reduction): MPI provides the following reduction function in which the result returns only to the root:
MPI_Reduce(sbuf, rbuf, n, data_type, op, rt, communicator)
The reduction operator is applied to the data given in the send buffer of each task
in the communicator's group. The result will be returned only to the receive buffer of the root.
Many-to-Many Reduction: A variant of the global combine operation is the many-to-many reduction operation in which the result is returned to all members of the group. MPI provides the following function for this operation:
MPI_Allreduce(sbuf, rbuf, n, data_type, op, communicator)
As in MPI_Reduce(). The result of the reduction appears in the receive buffers of all members of the communicator's group.

Hardware implementation model

Since the building block of a message passing machine is a complete system itself, this system could be built with a cluster of workstations on a LAN or off-the-shelf processors on a shared bus. Even though, there will be no special hardware, proven, off-the-shelf processors. Although the network topology is more important due to larger, more frequent data transfer which involve more hardware attributes, like nodes that route traffic, or intelligent interfaces to other processors.

A message passing machine could use the same topology as the shared memory machine. The following figures show an example of a network topology that could be used for either shared memory or message passing systems, we can realize in the following configuration that it allows transactions to be executed in parallel (as opposed to the shared bus configuration).

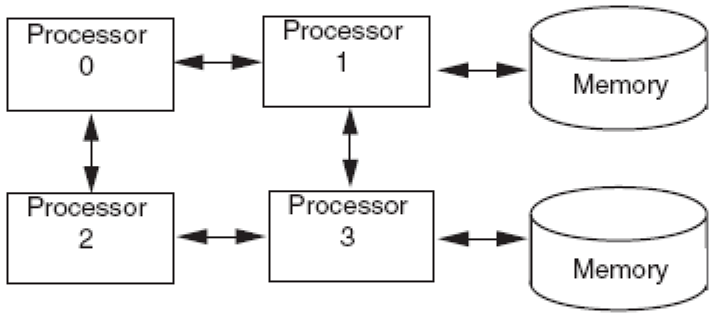


Figure 7: Parallel configuration [112]

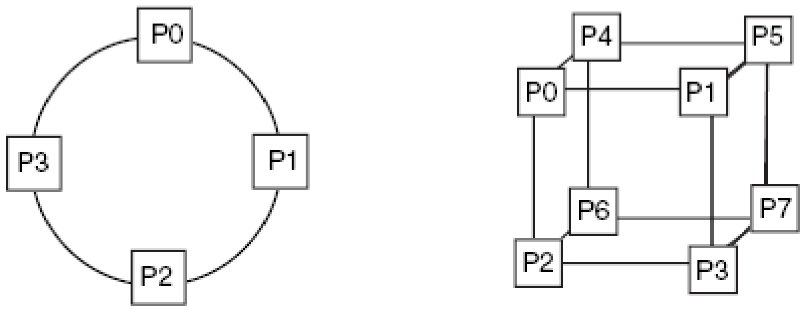


Figure 8 [112]: Ring and cube Network topologies [112]

Comparison between MPI and OpenMP

For MPI
It is known to give good performance for large numbers of processors and for larger problems and on a variety of distributed memory systems. It is portable across most parallel systems. It requires no special compiler and no special hardware but can make use of high performance hardware. What's more, it is very flexible as it can handle just about any model of parallelism with no shared data which mean that you don't have to worry about processes "treading on each other's data" by mistake. It forces you to do things the "right way" in terms of decomposing your problem. MPI is easier to build than scalable shared memory machines. On one hand, it maps closely to highly scalable architectures. The Programming model more removed from basic hardware operations. Om the other, the coherency and synchronization is the responsibility of the user, so the system designer need not worry about them. Easier to understand how the data is distributed and where the time is spent.

Against MPI
All-or-nothing parallelism so it is difficult to incrementally parallelize existing serial codes. There is no shared data so it requires distributed data structures. It could be thought of assembler for parallel computing so you generally have to write more code. Partitioning operations on distributed arrays can be messy. Moreover, it consists of large overhead: copying of buffers requires large data transfers (this will *kill* the benefits of multiprocessing, if not kept to a minimum). The programming is more difficult, complex to read, write and maintain. Blocking nature of SEND/RECEIVE can cause increased latency and deadlock issues.

For OpenMP
OpenMP code is initially much easier to write, modify and maintain. Once one code is satisfactory; it should be easy to parallelize other codes in a similar way. Incremental parallelism which can parallelize existing serial codes one bit at a time. It has a quite simple set of directives. The partitioning operations on arrays are very simple. No message starts up cost and can cope with irregular / data dependent communication patterns

Against OpenMP
It requires proprietary compilers and shared memory multiprocessors. Uses shared data and can't handle models like master/slave work allocation. In general, it is not scalable (more synchronization points), too easy to overlook necessary (or un-necessary) synchronizations. Not well-suited for non-trivial data structures like linked lists, trees etc
And there is a lack of tools inhibits quick program development. Portability is limited to subset of parallel systems with (virtual) shared memory. It can be harder to diagnose performance problems than in MPI so the communication costs are not easily measurable.

OpenMP or MPI?

It depends what you want to do, and what platforms you want to do it on. MPI is great for regular domain problems on 1000 processors. OpenMP is great for highly irregular problems on a small shared memory system. Most peoples needs lie somewhere in between. MPI codes often employ simple, brute force algorithms because it's too hard to code up anything more sophisticated as the scalability isn't everything.

Summary

Shared memory systems may be easier to program, but are difficult to scale up to a large number of processors. If scalability to larger and larger systems (as measured by the number of processing units) was to continue, systems had to use message passing techniques. It is apparent that message passing systems are the only way to efficiently increase the number of processors managed by a multiprocessor system. There are, however, a number of problems associated with message passing systems. These include communication overhead and difficulty of programming. In this lecture, we discussed the architecture and the network models of message passing systems. We explained briefly both the hardware implementation and software programming model; we also discussed the message passing routines. We concluded with a contrast between shared memory and message passing systems.

Paper 1: MPI: A Message Passing Interface.

Lecture #15b: Message Passing Models

Course: Computer Architecture III
Lecturer: Miodrag Bolic
Scribe: Miqdad Jaffer (2966605) & Kevin Mastrogiacomo (2946450)
Date: November 1, 2006

Introduction

.
A message passing architecture is used to communicate data among a set of processors without the need for a global memory. The intention is that each processor has its own local memory and communicates with other processors with messages. Message passing has a distinct advantage over a shared memory scheme, since there is no extra requirement of a large global memory and the corresponding load on synchronization. This lecture discusses three models for message passing systems, they are:

- Hardware Model
- Programming Model
- Message Passing Interface

Explanation

Hardware Model

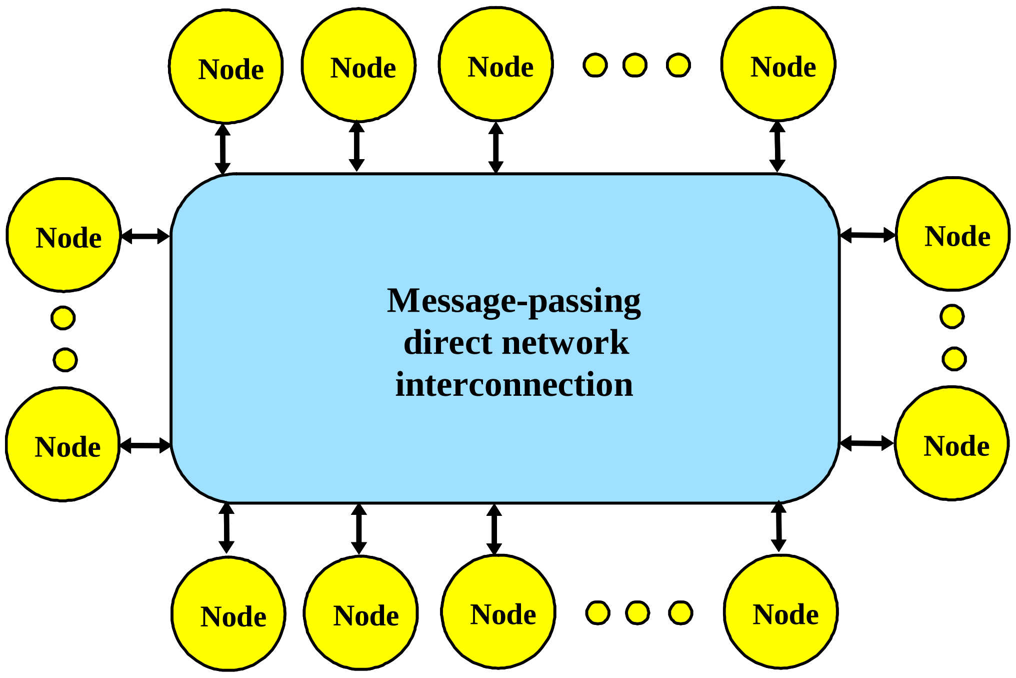


Figure 36 - Generic Model Of A Message-passing Multi Computer [124]

A message passing network is composed of nodes and the connection mechanisms between them. The node itself is what is most important. Each node is comprised of two main components as it relates to message passing. They are the node processor and the router.

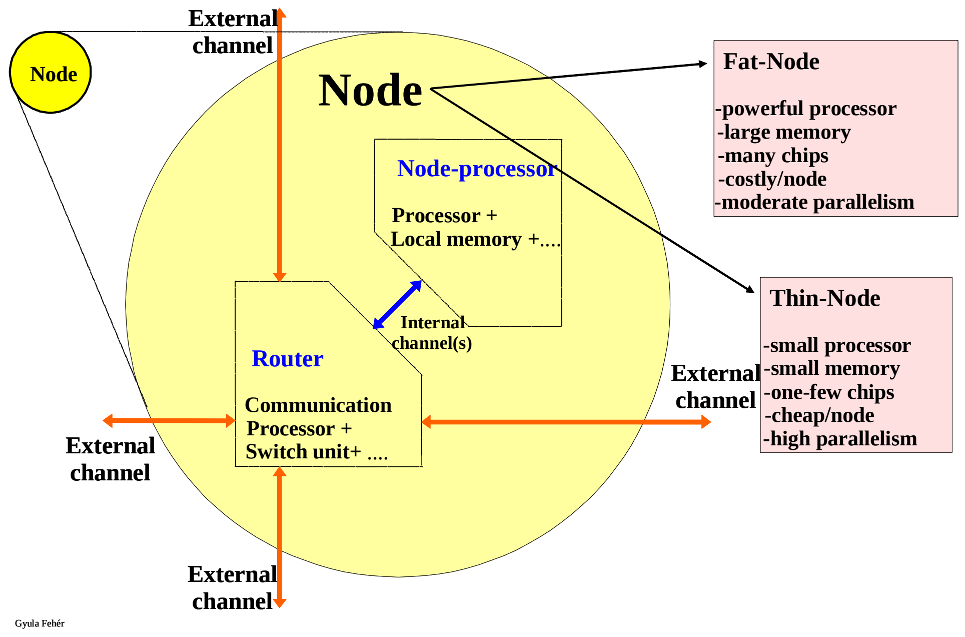


Figure 2 - Generic Node Architecture [124]

The node processor can be broken down into the computational processor and local memory. The router is made up of a communications processor and the switching unit. The communication between the two units is facilitated through the use of a shared internal channel. The router also has to maintain external communication with neighbouring nodes; as such it has a number of external channels to these other nodes.

Depending on the type of system desired, there are two methodologies used when developing nodes, coarse and fine grained.

Coarse Grained Approach

This method accepts the idea that a majority of the processing should be done on the node, and the amount of data that is sent through external node communication should be reduced [127]. Also called the fat node, it accepts the following general principles, in terms of node composition:

- Powerful processor

- Large memory
- Many chips
- Costly node
- Moderate degree of parallelism

Fine Grained Approach

In contrast the fine grained approach accepts that each node is a single process. The immediate benefit to this approach is that the cost of each node is significantly reduced, along with which the communication overhead is minimized. The minimization occurs because data being sent to each node is small and communication is relatively quick [127]. Communication of large amounts of data would result in more overhead because of the architecture of the processing element. Also referred to as the thin node, it applies the following properties:

- Small processor
- Small memory
- One – few chips
- Cheap node
- High degree of parallelism

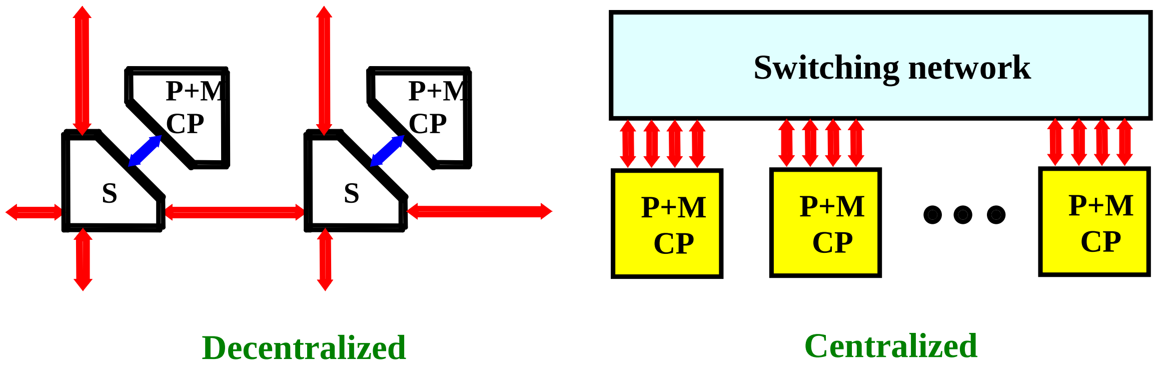


Figure 3 - Generic Organization Model [124]

The switching unit within the can use a mesh network where each router component has four (4) connections to the external system. However, for message passing a point to point connection system can be established using different routing strategies and different network topologies. Two possible solutions are the centralized and decentralized patterns.

The nodes of the centralized solutions are composed of both processor and memory. They use a switching network to route messages from one processing element to another The decentralized theory follows the design from Figure 3, where the switching unit is part of the node and the not a component of the network.

Properties of the Hardware Based Message Passing Model [120]
The hardware model used by MPI can be formed using complete computers. The computers can represent individual nodes. The computers must have a method of communication or an I/O port, in the case of complete computers this is Ethernet port. Having the communication reduces the cost, because there is no specialized hardware needed. Communication between processors can be integrated at the I/O level and not at the memory level, again eliminating the need for specialized hardware. Communication is achieved through SEND and RECEIVE messages being sent to the various computers / nodes. When the nodes are initialized they will allocate a small amount of memory used for message passing. This memory is used when performing parallel operations across computers/nodes.

Programming Model
The programming model for message passing, takes the initial premise that multiple processors are used to reduce to the total amount of work on any one processor, then surmises that the only way for this to be a functional alternative is to have some form of communication between the processors. In order to facilitate this requirement, we apply the principles of the message-passing paradigm. Instead of using a large global memory space, as in a shared memory scheme, we allow each processor to complete its task and then *send* the necessary information to a processor that is waiting to *receive* it. [123]

Once the information is passed correctly, the main processor (master) can complete the requirements of the process. The program below attempts to use the message passing principles to solve a simple problem, summing all the elements of an array of size n.

```
INITIALIZE; //assign proc_num and num_procs
if (proc_num == 0) //processor with a proc_num of 0 is the master,
//which sends out messages and sums the result
{
        read_array(array_to_sum, size); //read the array and array size from file
        size_to_sum = size/num_procs;
        for (current_proc = 1; current_proc < num_procs; current_proc++)
        {
                lower_ind = size_to_sum * current_proc;
                upper_ind = size_to_sum * (current_proc + 1);
                SEND(current_proc, size_to_sum);
                SEND(current_proc, array_to_sum[lower_ind:upper_ind]);
        }
        //master nodes sums its part of the array
        sum = 0;
        for (k = 0; k < size_to_sum; k++)
        sum += array_to_sum[k];
        global_sum = sum;
        for (current_proc = 1; current_proc < num_procs; current_proc++)
        {
                RECEIVE(current_proc, local_sum);
                global_sum += local_sum;
        }
        printf(“sum is %d”, global_sum);
}
else //any processor other than proc_num = 0 is a slave
{
        sum = 0;
        RECEIVE(0, size_to_sum);
        RECEIVE(0, array_to_sum[0 : size_to_sum]);
        for (k = 0; k < size_to_sum; k++)
        sum += array_to_sum[k];
        SEND(0, sum);
}
```

```
END;
```

Figure 4 - Message Passing Program [120]

There are four main functions that are used in the programming model, three of which are utilized in the above program; these functions are [120]:

- INITIALIZE
- SEND(receiving_processor_number, data)
- RECEIVE(sending_processor_number, data)
- BARRIER(n_procs)

At first INITIALIZE is used to assign a number to each processing element in the system. It also assigns the total number of processors to the variable num_procs. Once the processors have been initialized the SEND function is used to send *data* to the processor with number *sending_processor_number*. That processor must execute the RECEIVE command, to wait for the data. Here the receiving processor specifies the *sending_processor_number* and waits for *data* to be sent from that processor. BARRIER is a synchronization function that forces the program to wait for *n_procs* to reach this specific line of code.

With the basic understanding of these functions, we can correctly analyze what the program is doing. The structure of the program can be analyzed as follows:

- Initialize operation
  - Assign number of processors and number to each processor
- Processor 0
  - Reads the whole array
  - Determines the size that each processor must sum
  - Send the sizes of the array to appropriate processors
  - Performs its local addition
  - Stores its sum in the global sum
  - After this it has to wait for N processors to send their local results
  - It keeps repeating the RECEIVE message for N-1 processors
    - RECEIVE is blocking
    - Every time a value is received from the other nodes, it is added to the global sum
  - The processor than outputs the global sum
- If the processor is not processor 0
  - Each processor has to RECEIVE its size_to_sum
  - Performs local sum addition
  - Send its sum to processor 0

The programmer must know when to send, when to receive and what to send and receive. As an example, processor 1 cannot start adding until it receives the portion of the array it must sum. Once this is received only then can the local sum be calculated.

An important note to take away from the program is that when the RECEIVE function is called, the program enters a blocking state. This ensures that the data is received before the program carries on with the next operation

The process of sending and receiving requires that the message travel through the network, as such there may be an associated delay at some of the nodes, because of this we cannot guarantee the timing, the issue of deadlock may occur because of this, however this concept is beyond the scope of this lecture.

Message Passing Interface

The final model we will examine is the message passing interface (MPI). It much like the OpenMP library allows us an easier way to implement message passing within our system's design. The MPI library provides us with the following benefits [122]:

- **Standardization** - MPI is the only message passing library which can be considered a standard. It is supported on virtually all HPC platforms. Practically, it has replaced all previous message passing libraries.
- **Portability** - There is no need to modify your source code when you port your application to a different platform that supports the MPI standard.
- **Performance Opportunities** - Vendor implementations should be able to exploit native hardware features to optimize performance.
- **Functionality** - Over 115 routines are defined.
- **Availability** - A variety of implementations are available, both vendor and public domain.

Though there are a multitude of functions already defined, knowing the basic four (4), which have been defined previously, INITIALIZE, SEND, RECEIVE, SYNCHRONIZE (BARRIER), the message passing interface can be implemented.

Communicators

The MPI Library introduces the concept of communicators, the purpose of which is to provide a named set of processes for communication. The system allocates unique tags to processes. The library provides the means for all processes to be numbered from 0 to n-1, where n is the total number of processes, as well as the mechanism to construct libraries (i.e. application creates communicators).

One specific communicator is called MPI_COMM_WORLD. MPI uses communicators and groups to define which collection of processes may communicate with each other. The MPI_COMM_WORLD provides functions (split, duplicate, etc.) for creating communicators from other communicators. There are also functions (size, my_rank, etc) that can be used for finding out about all the processes within a communicator. [122]

Blocking vs. Non-Blocking

There are two main concepts for the send and receive communication, they are blocking and non-blocking. The former was examined in the programming model, this is where the RECEIVE function waits for a result before continuing. This concept of waiting is referred to as blocking. Since the process is blocked from performing any other action while it awaits a result or function to complete.

This concept can also be applied to the SEND function, where the process awaits acknowledgement that the data has been received so that it may continue. However, if the send is always blocked there can be many situations where the execution of the program is slowed unnecessarily. As this is the case for a majority of the time, the concept of non-blocking SEND exists. In this situation, the program executes and continues to execute without the need for an acknowledgement to continue. This has the advantage of speeding up execution, however, we must ensure that the reliability of the network is one in which we can take advantage of this feature, without causing the unsuccessful execution of our program.

Summary

Comparison
In comparison to the hardware model we can note certain advantages the programming model (which utilizes the MPI library) has. However, the programming model is also not without its own disadvantages, both are outlined below:

Advantages [120]

- Easier to build than scalable shared memory machines
- Easy to scale (but topology is important)
- Programming model more removed from basic hardware operations
- Coherency and synchronization is the responsibility of the user, so the system designer need not worry about them.

Disadvantages [120]

- Large overhead: copying of buffers requires large data transfers (this will *kill* the benefits of multiprocessing, if not kept to a minimum).
- Programming is more difficult.
- Blocking nature of SEND/RECEIVE can cause increased latency and deadlock issues.

Paper 1 Tutorial

Paper 2 MPI Communicator

Lecture #16a: Message Passing Architectures and Routing

Course: Computer Architecture III
Lecturer: Miodrag Bolic
Scribes: Ben Cuthbert (2942436), Matthew Fransham (2933464)
Date: November 3, 2006

Introduction

Previous lectures dealt with message passing between processors with little discussion about the details of the interconnection network that connects the processors. This lecture deals with how routing and switching occurs within the network.

The three main types of routing algorithms are deterministic, oblivious, and adaptive. Each one has its own disadvantages and advantages. An algorithm with higher intelligence has higher hardware requirements, but a less intelligent algorithm, with lower hardware demands, does not take the state of the network into consideration when sending packets. The three types of routing algorithms will be discussed in detail.

Switching techniques are also introduced. Circuit-switching and packet-switching are the two main types of switching techniques. Circuit-switching is used in such applications as telephone networks, while the most well-known use of packet-switching is the internet [133]. A comparison and analysis of these techniques can be found in Section 5.

Explanation

1. Functionality Required for Connecting Devices

In order to connect multiple devices together, the system requires the following functionality:

- Routing;
- Arbitration; and
- Switching.

Routing is the technique used for a message to select a path over the network channels. It is possible that there could be multiple paths from a source to a destination within a network, and it is the responsibility of the router to determine which path to take. If the entire path that the message will take is determined before sending the message, it is referred to as *centralized* routing. *Distributed* routing involves each node deciding which channel to use to forward the message. Centralized routing requires knowledge of the entire network, while distributed routing requires each node to know only the status of its neighbours. [128]

Arbitration involves determining when a packet can be sent. When packets request a shared resource at the same time, the arbitrator determines which packet is allowed to be sent. For all arbitration decisions there will be a winning packet and at least one losing packet. The winning packet will be granted the shared resource. In a lossless arbitration scheme, the losing packet(s) will be buffered, while in a lossy arbitration scheme the losing packets will be dropped [4]. Some of the issues that should be considered are priority and fairness [129]. Higher priority packets should be allowed to send packets first, but lower packets should also be granted send access at some point in order to prevent starvation.

Switching is the process by which paths are allocated to packets [4]. One switching technique is to reserve an entire path from source to destination and send all packets along this pre-determined path. This technique is called circuit switching. Another technique, packet-switching, involves sending smaller parts of the messages individually throughout the network. [128]

2. Shared-Media Network vs. Switched-Media Network

A network could be classified as either shared-media or switched-media. In a shared-media network, all nodes share a common network media, such as a bus. In this type of network, arbitration is extremely important since only one node should be allowed to send messages at a time. Arbitration should be the first process performed when a node wants to send a message. Routing and switching are straightforward in a shared-media scheme. For example, with a bus, a node broadcasts its messages to all other nodes on the bus. It is up to the other nodes to determine whether the destination address in the packet matches their address. Routing and switching processes are minimal.

In a switched-media network, all nodes are connected through a switch fabric. Paths between source nodes and destination nodes are dynamically established within the switch fabric. Aggregate bandwidth in a switched-media network is higher than in a shared-media network since many source nodes can be sending information at the same time (as long as they use independent paths) [4]. Routing is the first process that should be performed in a shared-media network when a node sends a message. This will determine the path that the message will take through the network. Arbitration will only be performed if two messages arrive at the same switch at the same time. Switching will occur after routing and arbitration.

There are advantages and disadvantages to the above-mentioned network types. Shared-media networks have low cost, but a global arbitration scheme is needed which could result in a bottleneck. Also, the time required for a node to send a message increases as additional nodes are added to the network, since there is more competition for the shared connection. A switched-media network is more scalable than the shared-media network and they allow for concurrent communication to occur. However, with this increase in performance comes a higher cost. [4]

3. Routing Algorithms

The three main types of routing algorithms are [130]:
Deterministic;
Oblivious; and
Adaptive.

Descriptions of the algorithms can be found in the subsections below.

3.1 Deterministic Routing

Deterministic routing algorithms route all messages along a predefined path from a source to a destination. This is the simplest form of routing algorithm and usually has poor performance. Deterministic routing does not consider alternate routes between nodes to avoid problems such as network congestion. However, with less intelligence comes the advantage of less hardware requirements and cheaper cost.

An example of a deterministic routing algorithm is the Greedy routing algorithm. The Greedy algorithm will always send a message from one node to another by taking the shortest possible path. There is no consideration about network congestion.

3.2 Oblivious Routing

Oblivious routing is one step above deterministic. A path between a source and a destination is chosen from a set of possible paths, but this choice is made blindly. Oblivious does not consider such factors as network congestion either.

Examples of oblivious routing techniques include uniform random and weighted random. Uniform random involves choosing a path between the source and the destination from all possible choices, where each path is given an equal probability of being chosen. Weighted random is similar to uniform random except the shorter paths are given higher probability of being chosen than the longer paths.

3.3 Adaptive Routing

Adaptive routing techniques involve changing the path on which a message is sent depending on the current state of the network. The state of the network can be affected by such factors as network congestion or the failure of a part of the network [132]. Also, if the buffer of a neighbouring node is full, the sending node will try to forward the message to a different neighbour. The objective is to balance the load on all channels in the network. Adaptive routing is the most intelligent routing algorithm, but this intelligence requires increased hardware requirements.

An example of how adaptive routing considers the state of the network when sending a message can be seen in Figure 1 below. Node A wants to send a message to node B. Node A first sends the message to node 1. Node 1 recognizes that the input buffer to node 3 is full, so it sends the message to node 2. Node 2 then forwards the message to the destination.

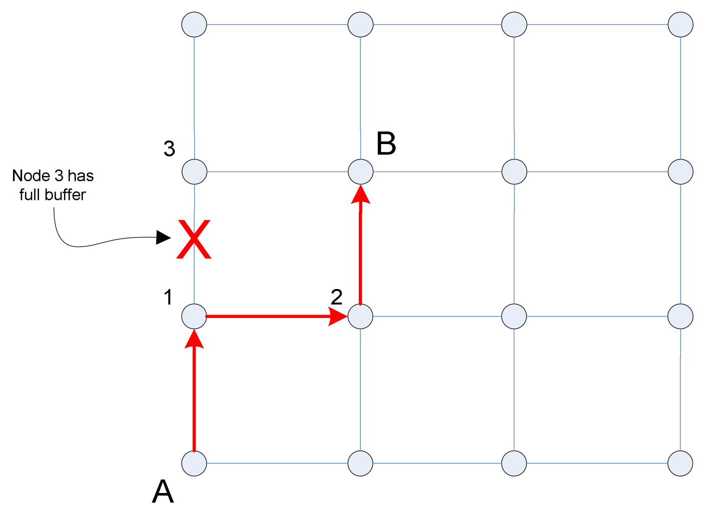


Figure 37: Example of Adaptive Routing

4. Routing Algorithm Analysis

4.1 Metric Definition

When performing a comparison between different network topologies, traffic flows and routing algorithms, metrics for the channels of the network, c, and for the network as a whole need to be measured or calculated. These metrics will be presented next, followed by an example.

A network channel is defined as a link between nodes x and y. The three characteristics of a network channel are its width, w_c, measured as the number of parallel signals the channel can contain; its frequency, f_c, measured as the rate at which bits are transported over a channel signal; and latency, t_c, measured as the time required for a bit to travel from x to y. Using the width and frequency of the channel, the bandwidth of the channel can be defined as $W = w_c * f_c$.

A network, which consists of a number of nodes and channels connecting those nodes, has two main metrics that are used for comparison: the maximum channel load, γ; and the throughput, Θ. The maximum channel load is determined by the channel that carries the largest fraction of traffic for a particular traffic pattern. The maximum value that the channel load can achieve is equal to the bandwidth of that channel. The throughput is the rate at which the network can accept data on an input port, in bits per second. This is calculated as $\Theta = W/ \gamma$.

4.2 Example

The parameters defined above will now be used in comparing the performance of four different routing algorithms that were described in the last section: greedy, uniform random, weighted random and adaptive. The comparison will be performed for a bidirectional ring network topology with eight nodes.

One of the most important tasks that must be performed when comparing different routing algorithms is determining the most realistic model (or pattern) for the traffic that the network will encounter. For this example, a tornado traffic pattern will be used. For a network of k nodes, the tornado traffic pattern is characterized by each node, N_i, sending messages to node N_j, where $j = \left(i + \left\lfloor \frac{k-1}{2} \right\rfloor \right) \bmod k$. In our example network, k = 8, so j = (i + 3) mod 8.

This example will generalize the channel width and channel frequency making the channel bandwidth W. The examples will assume an ideal channel, meaning that the channel latency is 0.

The direction of message passing for the greedy routing algorithm for this example is shown in Figure 38.

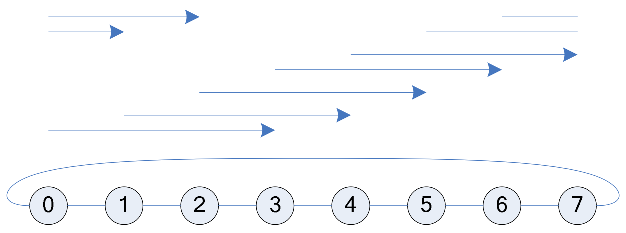


Figure 38: Tornado Traffic Flow - Greedy Routing Algorithm

The greedy routing algorithm utilizes the network channels in a clockwise direction 8/8's of the time. The maximum number of messages on a channel is 3. Therefore the maximum channel load is 3 * 8/8 = 3.

With a generalized channel bandwidth of W, the throughput for the greedy routing algorithm is Θ =W/3.

The direction of message passing for the uniform random, weighted random and adaptive routing algorithms for this example is shown in Figure 39.

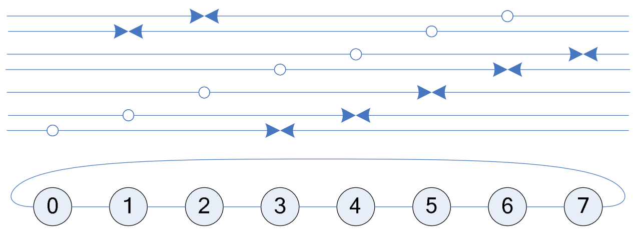


Figure 39: Tornado Traffic Flow - Other Routing Algorithms

The uniform random routing algorithm utilizes the network channels in a clockwise direction 4/8's of the time and in a counter-clockwise direction 4/8's of the time. The maximum number of messages on a channel when in the clockwise direction is 3 and the maximum number of messages on a channel when in the counter-clockwise direction is 5. Therefore the maximum channel load is 5 * 4/8 = 5/2, since 5 > 3.

With a generalized channel bandwidth of W, the throughput for the uniform random routing algorithm is Θ =W/(5/2) = 2W/5.

The weighted random routing algorithm utilizes the network channels in a clockwise direction 5/8's of the time and in a counter-clockwise direction 3/8's of the time. The maximum number of messages on a channel when in the clockwise direction is 3 and the maximum number of messages on a channel when in the counter-clockwise direction is 5. Therefore the maximum channel load is 3 * 5/8 = 15/8 or 5 * 3/8 = 15/8.

With a generalized channel bandwidth of W, the throughput for the weighted random routing algorithm is Θ =W/(15/8) = 8W/15.

The ideal adaptive routing algorithm will ensure that the channel load throughout the network is equally distributed. Since it has just been shown that the weighted random routing algorithm produces a maximum channel load in the counter-clockwise direction that is equal to the maximum channel load in the clockwise direction, the best maximum channel load that the adaptive routing algorithm can achieve is equal to the weighted random routing algorithm.

With a generalized channel bandwidth of W, the throughput for the adaptive random routing algorithm is Θ =W/(15/8) = 8W/15.

Table 3 shows a comparison of the results for the routing algorithms, assuming a channel bandwidth W = 1. As can be seen in the results, the weighted random and adaptive routing algorithms provide the best throughput for the example network.

Routing Algorithm	Throughput on Example Network
Greedy	0.33
Uniform Random	0.40
Weighted Random	0.53
Adaptive	0.53

Table 3: Summary of Routing Algorithm Results

5. Switching

5.1 Definitions

A message is the logical unit of communication between nodes [130]. A message can be of variable length; theoretically infinite length. To be transmitted through an interconnection network though, it is desirable that the size of the transmission be fixed, or at least have a maximum. To accomplish this, the message is broken into packets, the size of which is determined by the routing scheme and the network implementation [130].

The packet is the basic unit containing the destination address for routing the packet [130]. A packet can also contain a sequence number so that the destination can reorder any packets that are not received in order [130]. Some switching schemes divide a packet even further into flits, whose length is affected by the network size [128]. Unlike packets, all flits do not contain the destination address. Instead the flits are preceded by header flits that contain routing information and a sequence number [128].

Other factors that determine the size of packets and flits are the channel bandwidth, router design and network traffic [128].

5.2 Circuit Switching

Circuit switching consists of setting up a dedicated connection between the sender and receiver upon request, and pending availability. After communication between the two nodes has been completed, the reserved path is torn down to allow other nodes to use the channels. To initiate a connection, a header packet is sent to setup and reserve a communication path between the communicating nodes. The header packet is used to perform all routing, arbitration and switching tasks before communication commences. These tasks only need to be performed once for a connection. This reduces the amount of overhead present in each packet since the destination address is only included in the header packet and not in subsequent data packets. This also reduces latency, since no decisions need to be made after the connection has been established. [131]

The disadvantages, like the advantages of this mechanism, are due to the exclusive access to a path of channels between two nodes. The bandwidth reserved on the channel is static, irrespective of the bandwidth needs of the communicating nodes [128]. This could be an inefficient use of the resources of the network if the communication flow is in bursts or if it is low bandwidth. Also, all requests for the channel while the channel is already reserved are queued until the connection is torn down or the requests are dropped and must be resent by the initiator [131].

To try to reduce the amount of wasted bandwidth, it is possible to have pipelined circuit switching [131]. In this scheme data packets can be sent immediately after the header packet. Although these packets will incur some delay, this scheme does increase the amount of bandwidth utilized since there is no time of inactivity between the header packet and the message.

5.2 Packet Switching

Packet switching involves performing routing, arbitration and switching on a per-packet basis [131]. A message is divided into packets and each packet is sent individually. It is common that packets from the same message will be sent from the source to the destination along different paths. Due to this, it is possible that packets arrive at the destination out of order. To remedy this problem, an end-to-end message assembly scheme is required [128].

One of the main advantages of this technique is that resources are better shared than in circuit switching. In circuit switching, a path from a source to a destination is reserved and other nodes cannot use that path until the connection is torn down. Packet switching allows multiple nodes to use the same switch at the same time. However, there are some disadvantages as well. Since each node in the network is responsible for routing the packets (distributed routing), they must have knowledge of the state of the network. Communicating this status information causes additional overhead. There is also extra overhead with the end-to-end message assembly scheme. Despite any drawbacks, packet-switching is a very popular switching technique and is used in the internet. [132]

There are two main types of packet-switching: store-and-forward and cut-through switching. Store-and-forward works on a per-packet basis, while cut-through sub-divides packets even further into flits. These techniques are discussed further below.

5.2.1 Store-and-Forward Switching

Store-and-forward switching works at the packet level. A packet is sent to a switch, which will wait for the entire packet to be received before sending it to the next node. The main issue with this technique is that each node must contain enough buffer space to be able to store an entire packet [128]. However, there is still the advantage of dealing with messages in chunks, and not with the entire message.

5.2.2 Cut-through Switching

The advantage of cut-through switching is that it works at the flit level of a packet. Cut-through switching does not wait for the entire packet to be received before sending it along to the next hop. The chunks of the packet will be sent as soon as they are received, unless there is too much network congestion to do so. The two types of cut-through switching are virtual cut-through and wormhole. The difference between these techniques is their strategies for dealing with buffering messages. If a packet (or parts of a packet) cannot be sent due to network congestion (or a similar problem) virtual cut-through switching will buffer a complete packet at the switch. Therefore, the switch requires enough buffer space to store a complete packet, just as in store-and-forward. The wormhole technique is capable of storing only parts of a packet at a switch. This allows the switches to use smaller buffers. However, if a back-log occurs and a packet becomes buffered at several switches, that packet may monopolise the resources at many switches, blocking other network traffic. This disadvantage is similar to that seen with circuit-switching. If the links are not busy and flits can be forwarded as soon as they are received, the wormhole technique works best since it does not require larger buffers. However, if links are busy and a lot of buffering occurs, virtual cut-through is a better solution; it provides a nice compromise between store-and-forward and wormhole.

5.2.3 Analysis of Switching Techniques

The following parameters must be considered when analyzing a switching technique:

D:	Distance (number of hops);
F:	Flit Length;
L:	Packet Length; and

W: Channel Bandwidth (bits/s).

First, the store-and-forward technique will be considered. Figure 40 shows a sample of the performance.

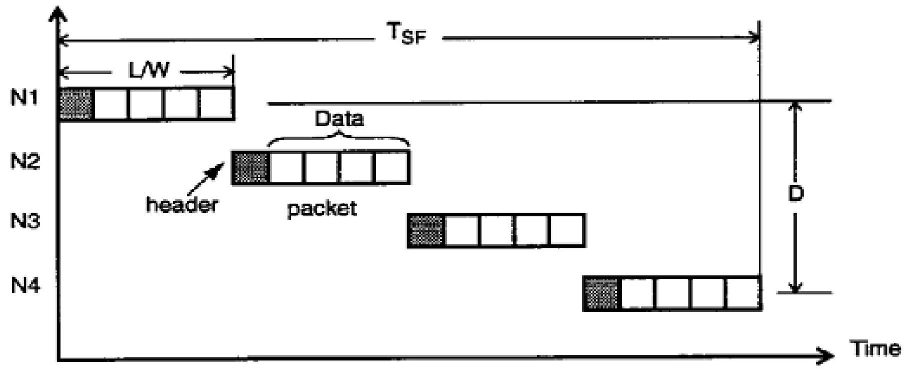


Figure 40: Performance of Store-and-Forward Switching [130]

As can be seen, a packet will not be delivered to the next node until the entire packet has been received. The latency for the store-and-forward technique is:

$T_{SF} = (D + 1) * L/W$

In contrast, Figure 41 shows the performance of the wormhole technique.

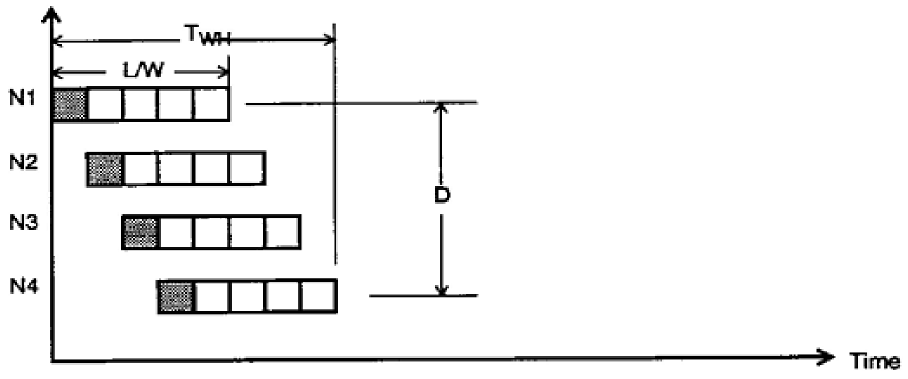


Figure 41: Performance of Wormhole Switching [130]

It can be seen that the latency for wormhole switching is much less than for store-and-forward since the nodes begin sending packets before they have been completely received. The formula for the latency of wormhole cut-through switching is:

$T_{WH} = L/W + D*F/W$

This formula corresponds to the length of time to send one entire packet, plus the time to send one flit per hop.

Summary

This lecture discussed the functions performed within an interconnection network that allows communication between nodes. The main functions are routing, arbitration, and switching. Routing is the process by which a path through the network is chosen for a packet or message, arbitration decides when a packet or message can be granted a resource, and switching is the process by which paths are allocated to a packet or message.

Various routing algorithms were discussed and contrasted. These techniques include deterministic, oblivious, and adaptive. Adaptive is the most intelligent technique and considers the state of the network when sending a packet, but its hardware requirements are also higher.

Different switching techniques were also introduced. Circuit-switching is a technique that reserves a path between a source and a destination. This technique is used in telephone networks. Packet-switching performs routing, arbitration, and switching on a per-packet basis. There are different forms of packet-switching including store-and-forward and cut-through. These techniques were also discussed.

This lecture provided a good overview of routing and switching techniques along with their respective advantages and disadvantages. Choosing the best algorithm depends highly on the application and available resources.

Paper 1 Impact of Virtual Channels and Adaptive Routing on Application Performance [134]

Paper 2 A Survey and Comparison of Wormhole Routing Techniques in Mesh Networks [135]

Lecture #16b: Message Passing Architectures and Routing

Course: Computer Architecture III
Lecturer: Miodrag Bolic
Scribe: David Au, Mumtaj Kamarujaman
Date: October 15, 2007

Introduction

In a message passing system, messages passed between nodes should be delivered efficiently and effectively. When sending a packet, there are several possible paths. There must be a way to determine which route the data will take. Routing provides a set of operations that are needed to compute a path. The routing can be performed at the source, destination, or while along the path. Network arbitration determines whether or not messages can be sent along a specific path at a point in time. Arbitration resolves the issue of nodes requesting resources at the same time. Switching determines how paths are allocated to nodes. Messages (in order by arbitration) are sent through the path. Paths can be established one fragment at a time or in their entirety. The design of a network can determine the kind of algorithms, arbitration, or switching that is required or more suitable to transport messages between network nodes.

Explanation

1. Routing
Routing algorithms can be categorized by the method in which they select their paths as [135]a:

Deterministic
Oblivious
Adaptive

1.1 Deterministic Routing

A deterministic routing algorithm is pre-set configuration of paths. For each source and destination, there is only one path for the packets to take. The path is specified in advance and does not change. Deterministic routing does not take into account the path diversity of the underlying topology, thus the design can offer poor load balancing. However, they are easy and inexpensive to implement and are inherently deadlock-free. The simple deterministic approach may be used in a complex system where a potentially more efficient algorithm would be more difficult to design. Destination-tag routing on butterfly networks and dimension-order routing for tori and mesh networks are two examples of deterministic algorithms.

1.2 Oblivious Routing

An oblivious algorithm considers a set of paths and chooses between them. Deterministic routing algorithms are a subset of oblivious routing algorithms. Oblivious routing can provide some load balancing, for example, by randomly distributing the traffic uniformly across a set of paths. However, it does not take into account the state of the network in its decision making. It is easy to compute channel loads (y) from the traffic pattern in oblivious routing because the algorithms give linear channel load functions. This linearity makes computing the worst-case traffic pattern for an oblivious algorithm relatively easy.

1.3 Adaptive Routing

Adaptive routing algorithms take into account the state of the network in order to make routing decisions. Traffic is routed through paths that are under the lowest load. Figure 1 shows an example of adaptive routing in a mesh network. In this example, node 12 wants to send a packet to node 5, but knowing that the path between nodes 4 and 8 is under load, the algorithm will adapt to this traffic and route the packet through another path to avoid the load.

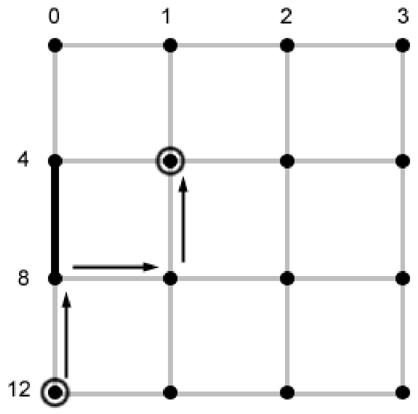


Figure 1 Adaptive routing example diagram.

2. Routing Definitions

The bandwidth, W, of a channel is $w_c * f_c$, where w_c (width) is the number of parallel signals in the channel and f_c is the rate of which each bit is transported at each signal.

The channel load, γ, can be equal to or less than the channel bandwidth. The maximum channel load, γ_{max}, is determined by the channel that carries the largest fraction of the traffic.

The throughput of a network, Θ, is the data (in bits per second) that the network accepts per input port. Throughput can be calculated as $\Theta=W/\gamma$.

3. Example of Routing Algorithms

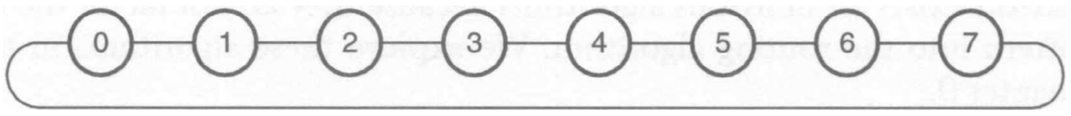


Figure 2 An 8-node ring network [135]a.

Consider an 8-node ring network (Figure 2) with a tornado traffic pattern. That is, each node n sends a packet to node (n+3)mod8. Figure 3 illustrates the configuration and characteristic of this kind of network traffic.

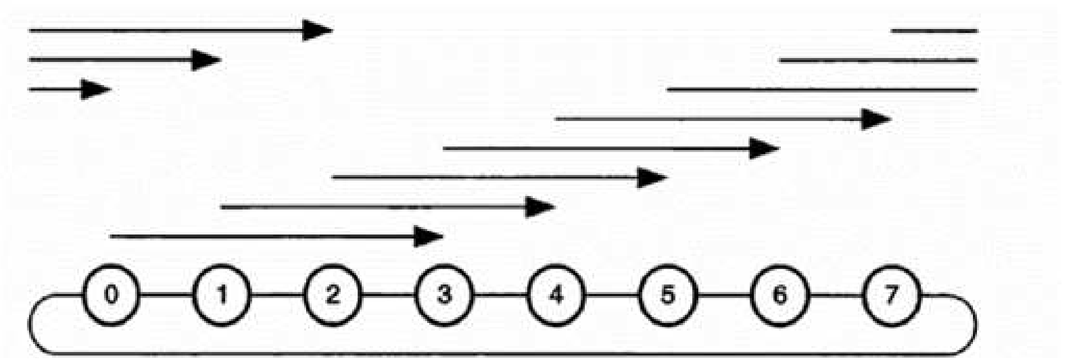


Figure 3 An 8-node ring network with tornado traffic [135]a.

Various routing algorithms can be applied to this network. We will look at the example with these algorithms:

Greedy
Uniform random
Weighted random
Adaptive

We will assume that the packets cannot backtrack while traversing through nodes.

3.1 Greedy

The packets will traverse the shortest direction around the ring. The paths for routing packets are predefined. For example, a packet will always route from node 0 to node 3 in the clockwise direction, while a packet from node 0 to node 5 will traverse in the counter-clockwise direction. If the distance is the same in both directions, the direction can be picked randomly.

The greedy algorithm can utilize all 8 of 8 network channels at any time in a single direction (clockwise for example). The maximum number of messages on each channel is 3. Thus, the maximum channel load of is 3 x (8/8) = 3. The throughput of the greedy algorithm is Θ=W/3. Assuming channel bandwidth, W = 1, the throughput of the greedy algorithm would be 0.33 for this example.

3.2 Uniform random

In this algorithm, the packets traverse to its destination node in a randomly chosen direction, where there is an equal probability for the packet to travel in either direction. This algorithm is an example of an oblivious routing algorithm.

Since there is equal probability for the packet to travel in either direction, then for this example, the probability of the uniform random algorithm sending messages clockwise is 4 of 8 times and likewise for counter-clockwise direction. If the number of messages traveling in a clockwise direction is 3 and 5 in the counter-clockwise direction, the maximum channel load can then be computed as 5 x (4/8) = 2.5. We use 5 in the calculation for maximum channel load because 5 > 3. The throughput of the uniform random algorithm is Θ=W/2.5. Assuming channel bandwidth, W = 1, the throughput of the uniform random algorithm would be 0.40 for this example.

3.3 Weighted random

The direction is picked randomly, but the probability is weighted on the short and long directions. The short direction is weighted with the probability of 1 – Δ/n and the long direction is weighted Δ/n, where n is the total number of nodes in the ring network (there are 8 nodes) and Δ is the shortest distance between the source and destination nodes. This is an example of an oblivious routing algorithm.

Given 3 messages traveling in the clockwise direction and 5 in the counter-clockwise direction, with a 5/8 weighting on clockwise messages and 3/8 weighting on counter-clockwise messages, the maximum channel load can be computed as 3 x (5/8) = 1.875 or 5 x (3/8) = 1.875. The throughput of the weighted random algorithm is Θ=W/1.875. Assuming channel bandwidth, W = 1, the throughput of the weighted random algorithm would be 0.53 for this example.

3.4 Adaptive

For this example of adaptive routing, we will send the packet in the direction with the lowest load on the local channel and the decision on the direction is applied once and at the source node. The load can be approximated by observing the length of the queue on the channel or how many packets it has transmitted over a recent interval.

Of the previous computed algorithms used in this example, the weighted random algorithm gives the best maximum channel load (1.875, lower load on a channel is better; less traffic on a single channel). Thus, the best adaptive routing algorithm to use in this particular network is the weighted random algorithm.

The adaptive routing algorithm would yield a throughput of Θ=W/1.875. Assuming channel bandwidth, W = 1, the throughput of the adaptive algorithm would be 0.53 for this example.

3.5 Results of the Routing Example

Application of the example routing algorithms on the 8-node ring network with tornado traffic pattern yield these throughput results:

Algorithm	***Throughput***
Greedy	0.33
Random	0.40
Weighted random	0.53
Adaptive	0.53

Table 1 Throughput of the routing algorithm examples [135]a.

As you can see from the results, the choice of routing algorithm for a network makes an impact on its performance. In this example, the average throughput of the network was affected.

4. Interconnection between many devices
Interconnection is classified into 2 types:
Shared Media Networks
Switched Media Networks

4.1 Shared Media Networks

In shared media networks the devices share common resources. The transmission between the devices is either bidirectional at different time (half duplex) or bidirectional at same time (full duplex). As shown in Figure 4, when node 1 is sending data to node 3, the path is occupied. Simultaneously if node 2 wants to send data to node 3, node 2 has to wait until the path is not occupied and send its data when the path is clear.

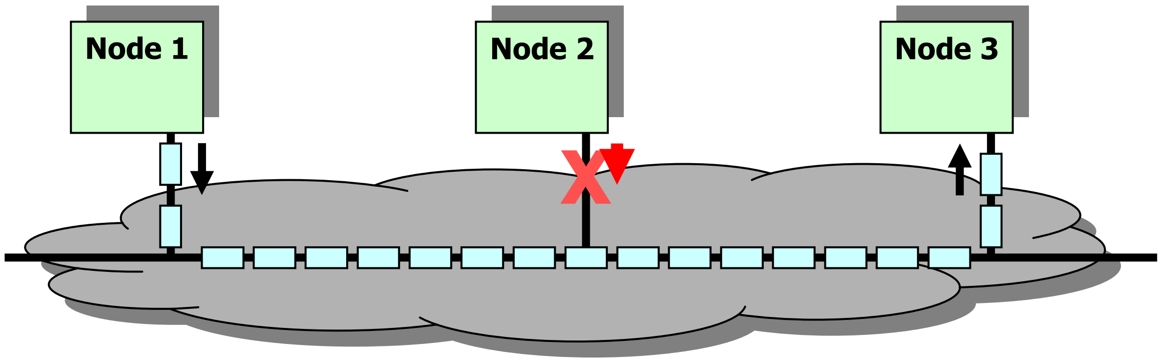


Figure 4 Example of a shared media network: bus topology [135]b

4.2 Switched Media Networks

Part of network that are disjoint are shared through switching. There are two types of switches passive and active. Passive switches connect one point to another. Active switches makes connections between source and destination dynamically. Bandwidth of switched media networks are greater than the bandwidth of shared networks.

Shared Media Network	***Switched Network***
When the number of devices increase aggregate network bandwidth *<u>does not scale.</u>* Requires global arbitration network.	When the number of devices increase aggregate network bandwidth *<u>scales</u>* to it.

5. Switching
Switching means connection of non adjacent nodes in a switched communication network.

Switching is classified into 2 types:
Circuit switching
Packet Switching

5.1 Circuit Switching

The path that the circuit is going to take is predetermined by routing, and when to take is decided by arbitration. Eg. telephone communication. There is a dedicated path between caller and the receiver. System decides which path in advance. During connection setup, connection is established, hence packet always uses same path, thereby reducing latency. After sending a request to send the packet, that packet is being sent to the destination.

Problem: The only problem with circuit switching is that the network is taken since it is a dedicated line.
Solution: we can pipeline messages.

5.2 Packet Switching

Switching is done based on a packet. Network bandwidth is also shared by packets. Packet switching is further classified into the following:
Store and forward switching
Cut through switching

***Store and Forwarding* [135]c**	***Cut Through[135]c***
Processes the entire packet before forwarding to the destination address. Hence it takes more time. But prevents collisions and erroneous packet from circulating in the network.	Only looks at the destination address to which the packet has to be forwarded to. No error control.

5.2.1 Store and Forward Switching

In this mechanism, though flits are being received by the buffer initially, the entire packets are being stored in the buffer and only then are forwarded. Packet transmission multiplies as the hop count increases. We don't need a buffer of message size, but rather we do need a buffer of packet size.

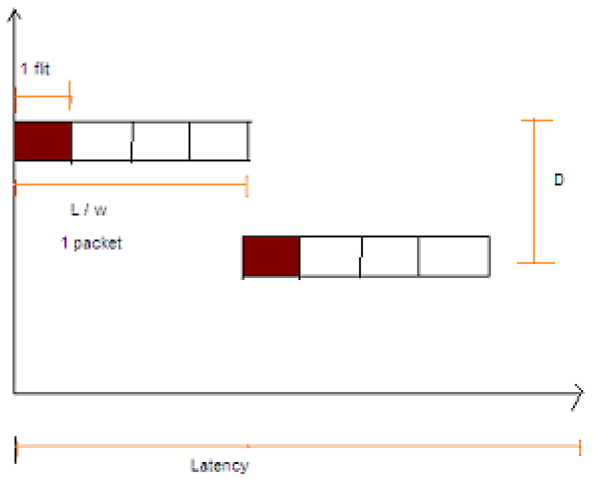


Figure 5 Store and forward switching timing diagram

$T_{SH} = (D + 1)(L/W)$

L → Packet size
L/W → latency of packet
W → bandwidth
D → # of channels

6.2.2 Cut through switching

When the header is receives the bits of the packet are starting to get transmitted. There is no wait for the entire packet unlike with store and forward switching. The following figure shows the transmission of the packets. The latency of cut through switching mechanism is given by T_{sh}.

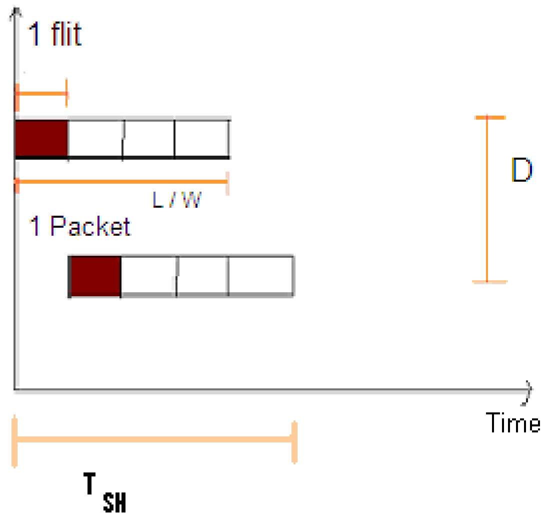


Figure 6 Cut through switching timing diagram

$T_{SH} = (D \times (F/W)) (L/W)$

L → Packet size
L/W → latency of packet
W → bandwidth
D → # of channels

There are 2 types of cut through switching:
Virtual cut through
Wormhole

Summary

5.2.3 Difference Between Virtual Cut Through and Wormhole

Virtual Cut Through	***Wormhole***

Control flow is based on flits. Store and forward Packets can be larger than buffers	Control Flow is based on packets Only forwarding no storing. We have busy link problem. Since buffers for data packets Buffers must be sized to hold entire packet

Paper 1: Adaptive Routing Algorithm for Lambda Switching Networks [135]d

Paper 2: Fault Tolerance of adaptive routing algorithms in multicomputers [135]e

Lecture #17: Flow Control and Deadlock

Course: Computer Architecture III
Lecturer: Miodrag Bolic
Scribes: Vijay Narasimhan and Jean-Sébastien Lauzon
Date: November 8, 2006

Introduction

1 This lecture provided perspectives on implementation of flow control in interconnection networks. Key concepts included:

- Packet-buffer flow control
- Flit-buffer flow control
- Backpressure
- Deadlock characterization, avoidance, detection, and recovery

Explanation

2 Packet-Buffer Flow Control - Review

2.1 Store-and-forward

In this flow control method, data is sent one packet at a time, and the nodes along the path must wait until a packet has been completely received before forwarding a packet to the next node. Two resources must be allocated before a packet can be forwarded: the packet must have the exclusive use of the channel and a packet-sized buffer is needed at the end of the channel [106].

2.2 Virtual cut-through

In this type, the nodes along the path forward a packet to the next node without waiting for it to be completely received. Like store-and-forward, the packet must have the exclusive use of the channel and a packet-sized buffer is needed at the edge of the channel. It only differs from store-and-forward by forwarding flits of the packet as soon as they arrive. This technique reduces latency; however, since the checksum of the packet cannot be computed until the entire packet is received, virtual cut-through switching is a disadvantage for reliability [137]. Flits are forwarded as soon as they are received, therefore only the last node can verify the checksum, i.e. erroneous messages can be forwarded over several nodes.

3 Flit-Buffer Flow Control

3.1 Wormhole

In wormhole flow control, large packets are broken into small pieces called **flits** (flow control digits). Wormhole operates like cut-through, but the channel and the buffers allocated to flits rather than packets.

Figure 42 demonstrates wormhole flow control. When the first flit of a packet arrives at the node, it must get hold of three resources: a virtual channel for the packet, one flit buffer, and one flit worth of channel bandwidth.

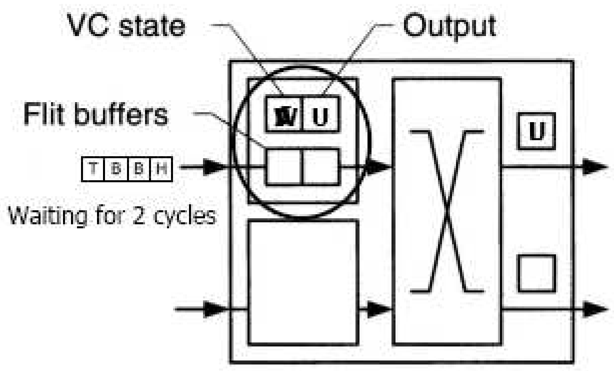


Figure 42. Wormhole flow control [138]

A virtual channel must include the following information [137]:
The output channel of the current node for the next hop of the route
The state of the virtual channel (idle, waiting for resources, or active)
Pointers to the flits of the packet buffered on the current node
The number of flit buffers available on the next node

Compared with cut-through, wormhole uses the buffer space more efficiently because only a small amount of flit buffers are required per virtual channel. The saving of buffer space comes at the expense of blocking [137]. Only the flits of one packet can use the channel bandwidth. When a flit can not get hold of a buffer, the channel goes idle.

What will happen if the channel is busy, or if the buffer is full? **Figure 43** demonstrates four methods for resolving these issues.

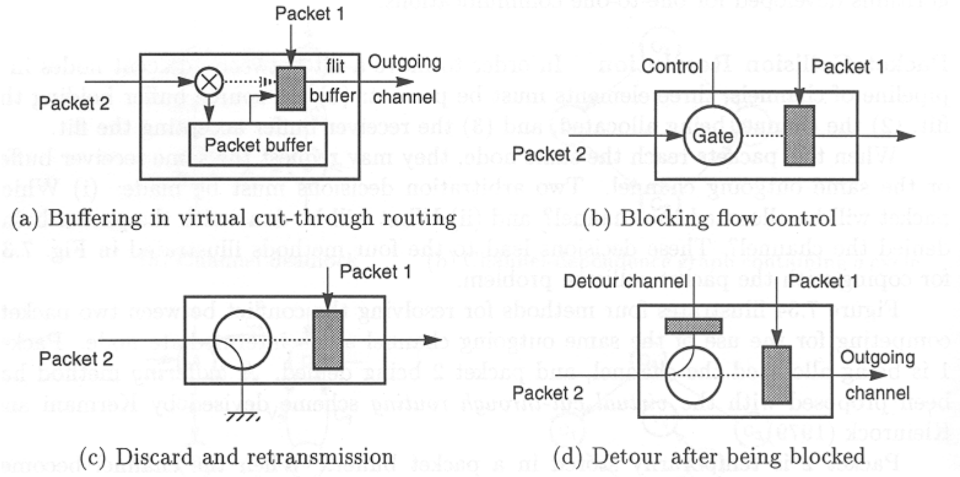


Figure 43. Methods to resolve congestion [138]

Buffering in virtual cut-through routing. If packet 1 is blocking the channel, queue packet 2 into the buffer. Buffer allocation should be done for case of blocking only. If there is no blocking, forward the flits through the channel.

Blocking flow control. If packet 1 is blocking the channel, block packet 2 until the channel is free.

Discard and retransmit. If packet 1 is blocking the channel, drop packet 2 and retransmit. It is up to the higher layer to retransmit the packet.

Detour after being blocked. If packet 1 is blocking the channel, send packet 2 along another route. Adaptive routing can be done with this type of hardware, but this method can cause deadlocks and livelocks (to be treated later).

3.2 Virtual Channel Flow Control

Virtual channel is very similar to wormhole flow control, but it associates several virtual channels on a single physical link. This solves the blocking problem of the wormhole flow control by allowing other packets to use the available channel bandwidth that would be idle when one of the packets blocks. This permits virtual channel flow control to have a substantially higher throughput than wormhole flow control [137]. A virtual channel, a free flit buffer, and a flit worth of channel bandwidth are needed to forward a packet. Virtual channels are competing to send their flits across the same link. Unlike wormhole, these flits are not guaranteed to have access to the necessary channel bandwidth.

Figure 44 (a) demonstrates packets being sent with wormhole flow control. In this situation, packet A is blocked at node 1. It is unable to forward its packet because it is unable to get hold of the channel. Channels p and q are idle because packet B is blocked at node 2 and not using the bandwidth of the links.

Figure 44 (b) demonstrates packets being sent with virtual channel flow control. There are 2 virtual channels on the physical link. Packet B is blocked at node 2 and it is using the first virtual channel. Packet A is able to get hold of the second virtual channel and use the unused bandwidth of the link to forward its packet to node 3.

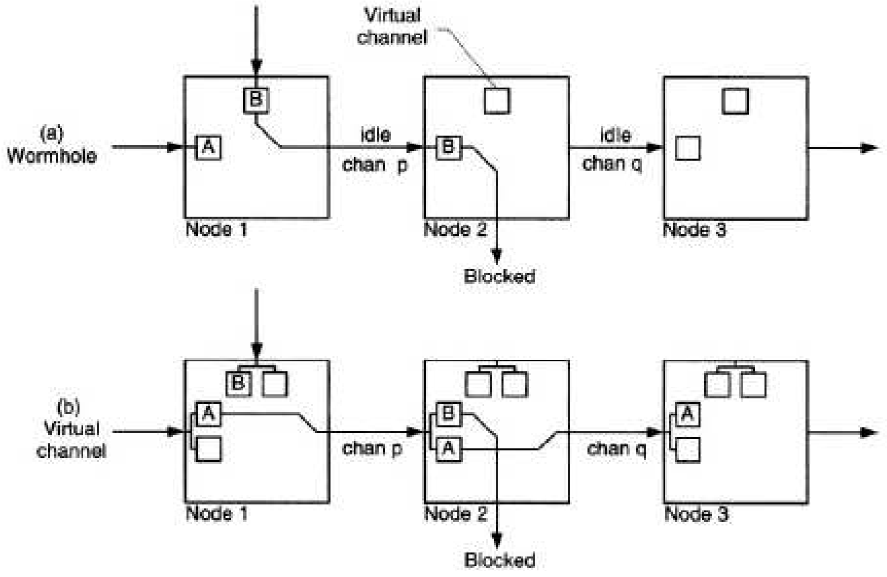


Figure 44. Example of wormhole and virtual channel flow control [137]

Figure 45 demonstrates how sharing bandwidth of a bottleneck link reduces both upstream and downstream bandwidth. Packet A and packet B share the bandwidth on a bottleneck link p. Since the flits of the two packets are interleaved at the output, they can only use 50% of the bandwidth each (1/2 flit cycle of the channel). Because link p is limiting the bandwidth of both packets arriving at full speed, packets leaving node 2 using the downstream link are forced to half of the rate. Because of blocking, the packets going through the upstream links are also throttled to half the rate of p. However, packets using a different virtual channel could use the idle bandwidth of the throttled link.

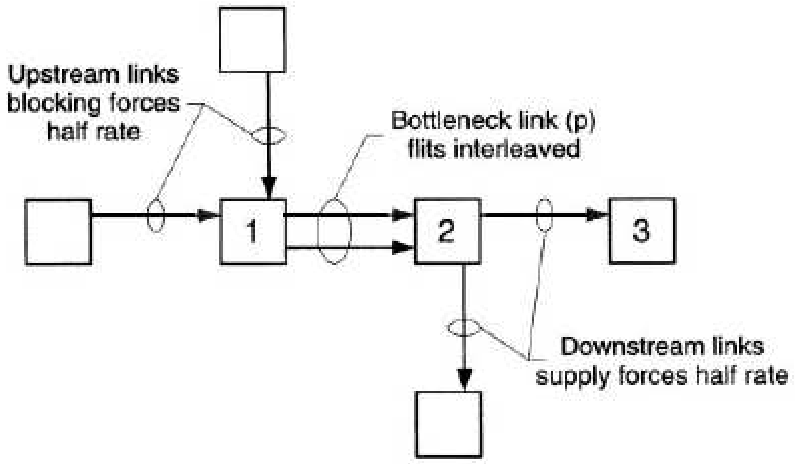

Upstream links blocking forces half rate
Bottleneck link (p) flits interleaved
1
2
3
Downstream links supply forces half rate

Figure 45. Sharing bandwidth of a bottleneck link

A flow control method that uses buffering needs to be able to communicate the availability of the buffers. A downstream node informing an upstream node to stop sending flits (or packets) because all of its buffers are full is called backpressure. Three low level buffer management techniques that provide backpressure are available [106]: credit-based, on/off, and ACK/NACK.

4.1 Credit-based

With the credit-based technique, the router keeps track of the number of free flit buffers in each virtual channel downstream. This number corresponds to a credit. Each time an upstream router sends a flit, it takes a buffer space downstream; consequently it spends a credit (decrement the count of free flit buffers). When the credit count is zero, all buffers are full and no flits can be forwarded. When a downstream router forwards a flit, it frees a buffer space and sends a credit to the upstream router (increment the count of free flit buffers). A typical credit exchange (assuming two nodes have full flit buffers) is shown in Figure 46. The credit count for node 1 is also shown.

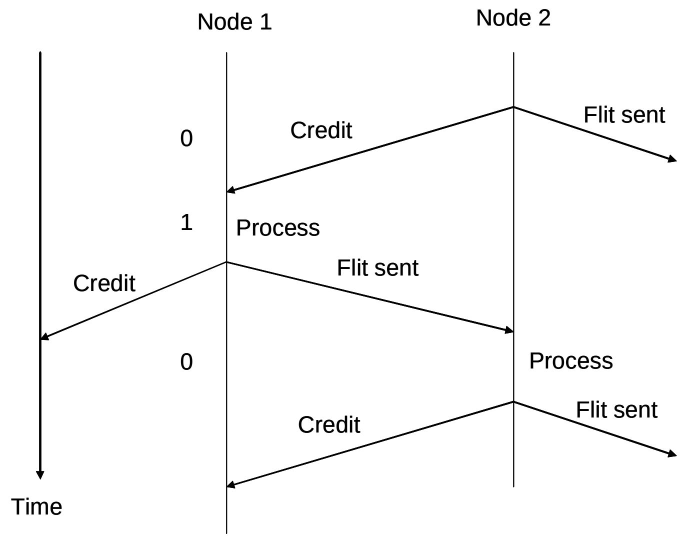


Figure 46. Credit-based flow control exchange

4.2 ON/OFF

In this technique, the upstream node has one control bit that represents its state. The state is ON if the upstream node is allowed to send and OFF if it is not. An OFF signal is sent to the upstream node when its state is ON and the amount of free buffers falls under the OFF threshold. An ON signal is sent to the upstream node when its state is OFF and the amount of free buffers rises over the ON threshold. A signal is only sent upstream when it is necessary to change its state, therefore, this method can reduce the amount of upstream control data being sent [137].

4.3 ACK/NACK

With the ACK/NACK technique, an upstream buffer forwards its flits as soon as they are available. If the downstream node has free buffers it will accept it and reply with an acknowledgment **(ACK)**. If the downstream node does not have a free buffer it will drop the flit and reply with a negative acknowledgment (**NACK**). An upstream nodes needs to keep a copy of the flit until it receives an acknowledgment. If the upstream node receives a negative acknowledgment, it resends the flit until it receives an acknowledgement. There is no need to keep the state indicating the downstream buffer availability in the upstream node because the flits are sent assuming an optimistic approach (i.e. the upstream node assumes that the downstream node can receive the flit).

4.4 Comparison

Table 4 compares the characteristics of the three buffer management and backpressure techniques.

	Credit-based	**ON/OFF**	**ACK/NACK**
Upstream node state	Number of free buffers (**credits**)	Control bit (**ON** to send, **OFF** to idle)	No state
Upstream signalling	One signal per flit (very high signalling)	If there are enough buffers the upstream, signalling is very low	One signal per flit (very high signalling)
Used	When there are a small number of flit buffers	When there are a large number of flit buffers	Rarely (too inefficient)

Table 4. Comparison between credit-based, ON/OFF and ACK/NACK technique

5 Deadlock

5.1 Overview

If you have ever been driving downtown during rush hour, you may have been caught in traffic because of the situation illustrated in Figure 47. On these one-way streets, each car is stuck because some other car is blocking it.

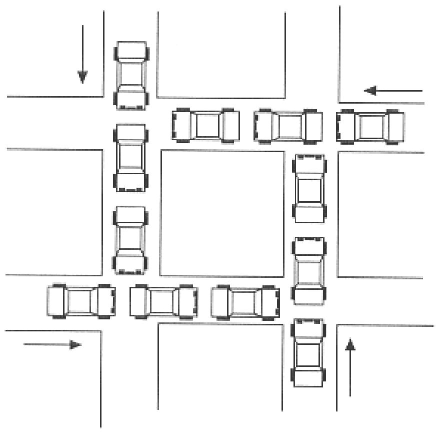


Figure 47. Deadlock in downtown [140]

Consider now a four-node circuit-switching network with unidirectional channels, as illustrated in Figure 48. Recall that in such a network, physical channels are allocated to a particular connection when the transaction starts. Let us observe what happens when node 0 wants to communicate with node 3 (connection *A*) and, at the same time, node 2 wants to communicate with node 1 (connection *B*).

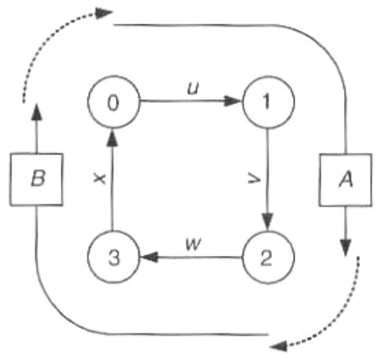


Figure 48. Deadlock in a circuit-switching network [137]

A captures channels *u* and *v* and *B* captures *w* and *x*. Now, both connections can not proceed. *A* waits for *B* to release channel *w* while *B* waits for *A* to release channel *u*. This situation has brought the network to a grinding halt; no connections can be established.

The term **deadlock** describes both situations illustrated above. Deadlock is defined as a situation in which a group of **agents** are unable to progress because they are waiting on another agent to release some mutually exclusive **resource** [137]. In the traffic example, the agents are cars, and the resources are spaces on the road. In the network example, the agents are connections, and the resources are physical channels.

We can also define agents and resources for networks with other types of flow control, as shown in Table 5. In both packet- and flit- buffer flow control, the agents are packets. At each node in packet-buffer networks, the packet may capture and hold onto a buffer if there is no space in the buffer of adjacent nodes. A packet can capture and hold onto several virtual channels in a flit-buffer network when it gets blocked because the flit buffer at each node is not large enough to hold the entire packet.

Network	Agent	Resource
Circuit Switching	Connection	Channel
Packet-Buffer	Packet	Packet Buffer
Flit-Buffer	Packet	Virtual Channel

Table 5. Agents and resources in various types of flow control [137]

Beyond physical resources, deadlock can also occur in a network due to operations on some higher layer [137]. Consider, for example, the distributed memory system depicted in Figure 49.

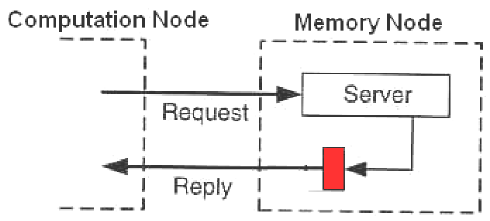


Figure 49. Protocol deadlock (adapted from [137])

Here, the computation node needs to fetch data from the memory at some remote node. Suppose we need to add two operands. The code in the computation node could read something like:

```
SEND(mem_node,addr_A); //Ask for data from addr_A
SEND(mem_node,addr_B); //Ask for data from addr_B

RECEIVE(comp_node,a); //Put the first reply in A
RECEIVE(comp_node,b); //Put the second reply in B

c = a+b; //Add and store in local variable c
```

The computation node makes a request to the server at the memory node to retrieve operand A. The server receives the request and puts the requested data in its reply buffer. It then waits for this reply buffer to clear before accepting any more requests. However, the computation node does not clear the reply buffer right away; instead, it makes a second request, this time for operand B. Since the server does not accept the request, the computation node hangs on this instruction. The computation node is waiting for its request to be processed, and the memory node is waiting for its reply to be processed. Neither of these will happen, so we have a form of deadlock known as **protocol deadlock**, which originates from factors external to the network itself.

Deadlock is a critical problem in interconnection networks (INs). As we have seen from these examples, it can totally cripple communications. To understand and deal with deadlock effectively, we must understand the necessary conditions for deadlock to occur and develop strategies to prevent, avoid, or detect and recover from deadlock in the network. These issues will be discussed in the following sections.

5.2 Characterizing Deadlock

For deadlock to occur, three conditions must prevail in the network [141]:

Mutual exclusion. There must exist a resource that can be acquired by only one agent at a time. In a network, such mutually exclusive resources could include physical channels or buffers.

No preemption. No agent is able to forcibly take a resource from a waiting agent. Preemption may arise in networks with some notion of priority, but, in many cases, preemption simply is not possible.

Cyclical hold-and-wait. If we assign indices to some set of agents, $\{A_0, A_1, \ldots A_{n-1}\}$, each A_i must be waiting for a resource held by $A_{i+1 \bmod n}$. In other words, an agent must hold a resource and be waiting for a resource held by another agent, and this must occur in a cycle.

In the networks we will examine in this section, the first two conditions will always hold. It is also straightforward to see that network agents can exhibit the hold-and-wait condition. In circuit switching, while the connection is being established, the connection holds the channels from the source to some intermediate node as it waits for channels to become free from that intermediate node to the destination. In packet switching, each packet holds on to buffers at one or more nodes while the network bandwidth or buffers at subsequent channels are occupied.

How, then, can we see if the hold-and-wait is cyclical? For networks with a few nodes, a resource dependence graph and a resource allocation graph can help. In a **resource dependence graph**, we draw an arrow from any resource that an agent can hold to any resource for which the agent could be waiting at the same time. Clearly, in our example from Figure 48, since the network is unidirectional, we draw an arrow between adjacent channels in the direction of network flow, as shown in Figure 50.

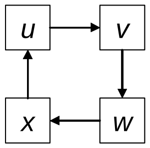


Figure 50. Example resource dependence graph

A cycle in this graph shows that deadlock *may* occur; it only shows potential hold-and-wait conditions. To see if a deadlock has actually occurred given a certain network scenario, we use a resource allocation graph. Graphically, we denote each agent with a circle and each resource with a square. If an agent holds a resource, we draw an arrow directed from the resource to the agent that holds it. If an agent is waiting for a resource, we draw an arrow directed from the agent to the resource it is waiting for. A resource allocation graph for the example in Figure 48 is shown in Figure 51. There is clearly a cycle (shown in red). *A* is waiting for a resource held by B, and B is waiting for a resource held by *A*.

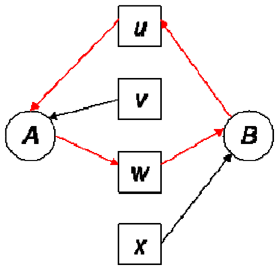


Figure 51. Example resource allocation graph

The three conditions we outlined are always necessary conditions for deadlock to arise in the network [141]. They are only sufficient, however, if there is no auxiliary resource that the agent can capture, for example, an alternate pathway or buffer [137]. Pursuing alternative pathways that do not ultimately lead to the destination node can lead to a situation called livelock; however, we will leave discussion of this to a later section.

5.3 Deadlock Prevention

Deadlock prevention involves breaking one of the three necessary conditions so that deadlock can never occur. For example, in the 3-stage Clos network of Figure 52, if we consider the inputs and outputs of the network to represent the same nodes, then any pair of them can communicate without blocking resources that prevent the others from communicating. Thus, the entire network can be busy with no hold-and-wait states. Deadlock prevention requires a lot of hardware redundancy (in our example, the Clos network has a number of switches proportional to the square of the number of nodes) and is seldom used outside of the context of switching networks [143].

5.4 Deadlock Avoidance

Deadlock avoidance is currently the most common strategy used in INs [3]. In avoidance techniques, physical resources may in fact show some dependence (unlike networks that use deadlock prevention techniques). However, routing rules are imposed such that cycles can never arise. Usually, partial or total ordering is imposed on the resources, and agents are mandated to only acquire resources in ascending order [106]. When there are multiple resources at each node, say in packet-switching networks with multiple packet buffers or virtual channels per node, then we can assign a resource from each node to a certain **resource class** within the order.

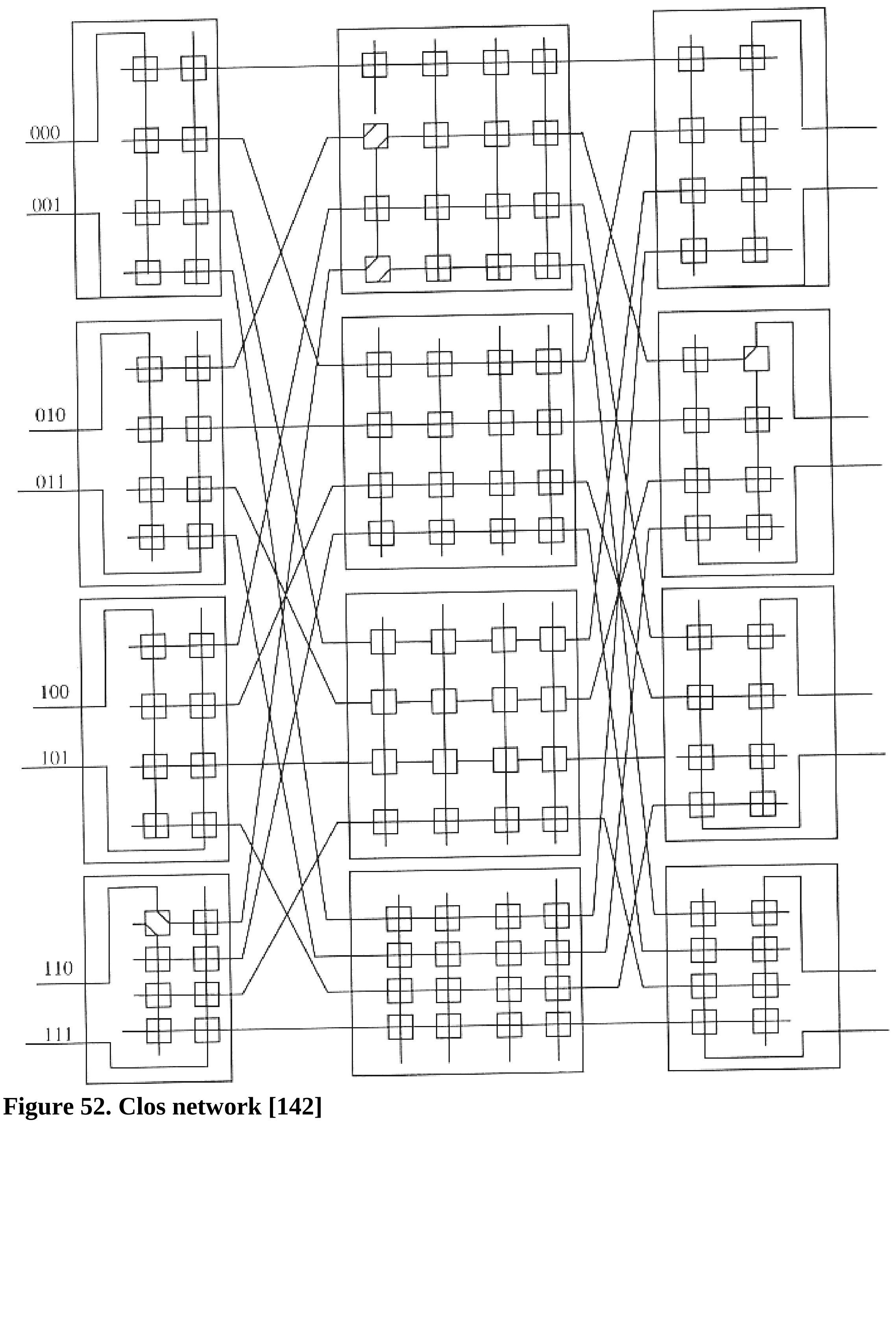


Figure 52. Clos network [142]

There are several ways of ordering the resources. In **distance ordering**, the resource a packet captures at each node depends on the number of hops the packet has taken to get to that node [2]. For example, consider a four-node packet-buffer network with four buffers per node. Messages starting from each node are assigned to buffer class 0. After they have traveled one hop, they are assigned to class 1. After they have traveled two hops, they are assigned to class 2. Finally, after three hops, they are assigned to class 3. This situation is shown in Figure 53.

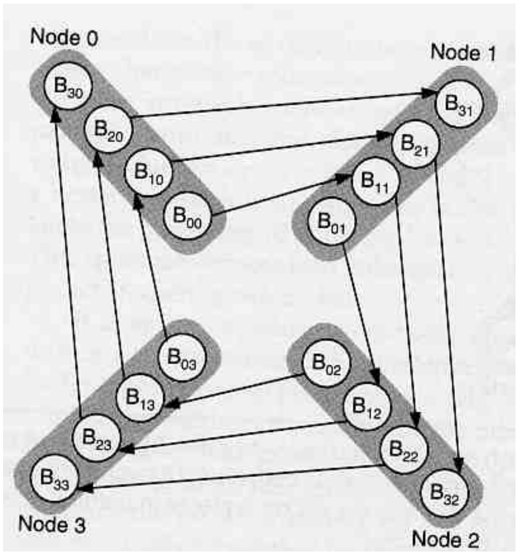


Figure 53. Resource dependence graph for network with distance ordering [2]

All of the nodes can communicate with each other, but there are no cycles in this graph. Thus, the ordering and routing rules have successfully prevented deadlock. The cost of this method is a significant increase in the number of buffers of each node. In fact, [2] states that the number of resources at each node is proportional to the diameter of the network. In larger networks, the expense of this form of avoidance may not be feasible.

Another type of ordering which requires only two buffers at each node is one which uses dateline classes [137]. In Figure 54, we have the resource allocation graph for a 4-node flit-buffer ring network with two virtual channels. The agents are designed to always request VC0 first. If a wants to establish a channel between u and x, and b between w and v, we again have a deadlock. However, if the connections could deterministically have figured out which of the two channels to take, deadlock would have never happened.

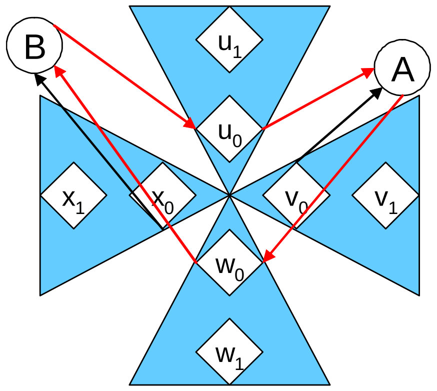


Figure 54. Resource allocation graph for flit-buffer network

There is a cycle in the resource dependence graph of this network, as shown in Figure 55. We could easily solve this problem by placing a dateline between nodes w and x, as shown in Figure 56. Messages start routing on class 0, and switch to class 1 as they cross the dateline. Thus, cycles are broken and deadlock is avoided. An interesting note with dateline avoidance is that messages will apply an unbalanced load to the system, sending more messages along VC0 than along VC1 [137]. Thus, some virtual channels are underutilized, diminishing performance.

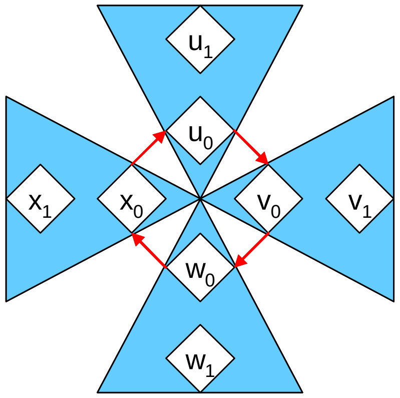


Figure 55. Resource dependence graph for flit-buffer network

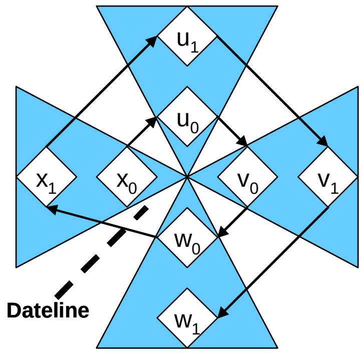


Figure 56. Resource dependence graph for flit-buffer network with dateline

In both distance and deadline routing, we see that many more buffers are needed at each node. An alternative approach is to restrict the routes that can be taken between nodes (i.e. by imposing deterministic, deadlock-free routing). One example of this approach is dimension-order routing in mesh networks [137]. For example, consider the X-Y routing algorithm used in the 2D mesh of Figure 57. Here, the links in the mesh are ordered from left to right going bottom to top (0, 1, 2, 3, 4, 5), then from right to left going bottom to top (6, 7, 8, 9, 10, 11), then from bottom to top going left to right (12, 13, 14, 15, 16, 17), and finally from top to bottom going left to right (18, 19, 20, 21, 22, 23). Again, we impose the restriction that resources can only be acquired in ascending order. Thus, packets will always travel in the X direction first, and then in the Y direction. This gives rise to the name X-Y routing. In the illustration, the route {0, 1, 16} is permitted (since it is in ascending order), but the alternate route {0, 14, 3} is not.

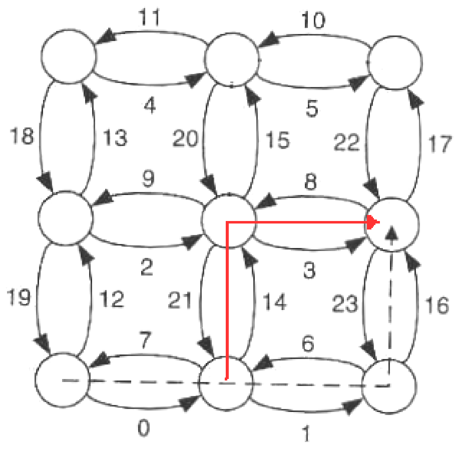


Figure 57. Dimension-order routing in a 2D mesh (adapted from [137])

In sum, when we use deadlock avoidance, we impose some kind of ordering on the resources in the network. For a cycle to occur, some agent must hold a resource with a higher order and be waiting for a resource with a lower order, but this is a contradiction since our rules disallow it. Thus, we conclude that there can be no cycles. Deadlock avoidance usually implies hardware redundancy (through additional buffers, for example) or some loss of resource utilization (by restricting routing, for example).

5.5 Deadlock Detection and Recovery

Some of the performance lost with deadlock detection and avoidance can be preserved if we allow deadlocks to happen. This means that buffer bank sizes can be restricted and fully adaptive routing can be used. Our goal will thus be to detect deadlock quickly when it occurs and then correct the situation before the network becomes totally crippled.

Deadlock detection involves identifying when the network is deadlocked [141]. It could involve checking for cycles in the resource allocation graph, but this involves scanning the entire graph in the worst case and could thus be very costly. Instead, many deadlock detection algorithms are designed conservatively; they will always find deadlocks, but may sometimes have “false positive” results, i.e. incorrectly identify network situations as deadlock, such as networks that are still operational but heavily congested. To accomplish this, we associate a timeout with each agent, and if the timeout has passed without the agent making any progress, the situation is corrected.

Deadlock recovery is the process of getting the network out of a deadlocked state. In **regressive recovery**, agents that have timed out are simply removed from the network and retransmitted. This fits well with the drop-and-retransmit method of flow control. For example, in a circuit switching network, each node checks a timeout on a partially constructed channel. If the timeout passes, the channel is torn down, from the intermediate node all the way back to the source, and a sort of negative acknowledgement, or NACK, is sent along as the path is torn down to alert the source node that the channel was not successfully created and should be re-established later.

With flit-buffer flow control, we could append empty flits to the end of each message so that the source will only be able to send the entire packet if there are virtual channels all the way to the destination. We can show why this is important. If the number of flits is such that the entire packet can be sent before the packet reaches the destination (Figure 58), the source node has no way of knowing if deadlock has occurred, and detection must be undertaken by the intermediate nodes. This may result in many false positives if the network is congested. This will in turn cause needless re-transmission of message, which will clog the network further. If, on the other hand, we pad the packet with empty flits such that the source can not send the final flit until the destination has successfully received a flit (Figure 59), the source can abandon the transmission and send messages to the intermediate nodes to flush their buffers if the final flit is resident in the source for too long. The source knows that if the destination had received a flit, then there exist virtual channels all the way from the source to the destination, and deadlock could not have occurred. Although this deadlock detection strategy adds some overhead, we can assume that it will not unnecessarily block too many resources since the number of flits in a packet is roughly the same size as the diameter of the network [2].

Progressive recovery detects deadlock in much the same way, but, if deadlock is detected, the packet is not removed from the network, but simply given an “escape route” [137]. A simple example would be backup buffer at each node in a packet-buffer network. If a node, through timeouts, detects that a packet may be deadlocked, it stores this packet in the backup buffer until some later time when the network clog has cleared. Another strategy is to store “normal packets,” i.e. those which are not deadlocked, in the backup buffers so that deadlocked packets are delivered quickly [143]. The performance of these algorithms is generally higher since the message does not need to be resent from the source; however, some hardware redundancy is required.

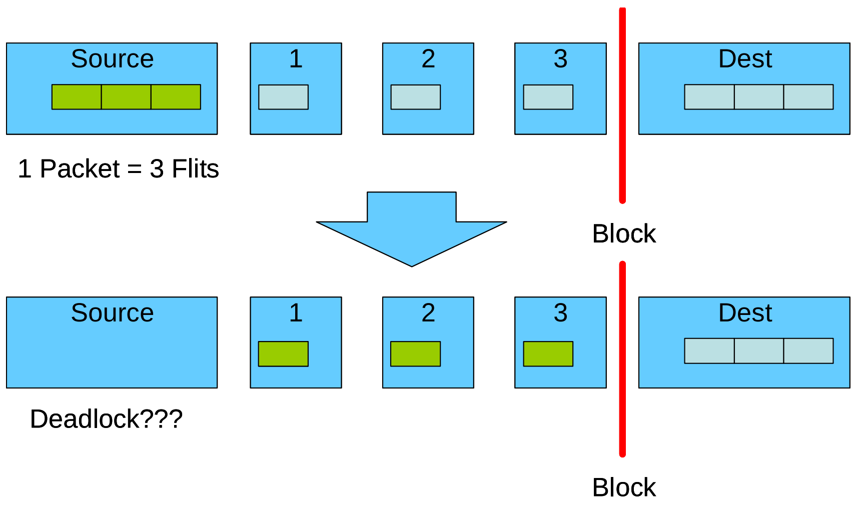


Figure 58. Flit-buffer network with potential deadlock

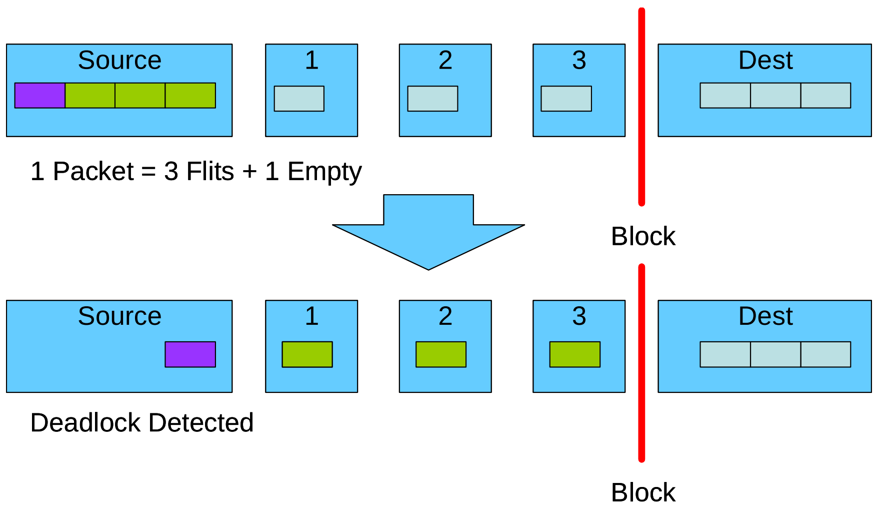


Figure 59. Flit-buffer network with deadlock detected

5.6 Livelock

In flow control methods that use dynamic re-routing or dropping, there is a danger that a packet may be forwarded or retransmitted indefinitely; this is called **livelock**. It can also be dealt with using counter-based timeouts. For example, if we add a flit to each packet that counts the number of times the packet has been misrouted, once a node detects that this count has passed a certain threshold, it will give this packet priority over other packets and route it correctly so that it may reach its destination.

Summary

Deadlock can occur whenever agents can cyclically hold and wait for resources in a network. It can be dealt with by prevention, avoidance, or detection. Each solution has a cost of either increased hardware redundancy or reduced performance due to routing limitations. Networks with deflection or drop-and-retransmit flow control are also susceptible to a condition called livelock in which packets travel in the network never reach their destination. Counters can be used to ensure that packets can only be misrouted a certain number of times.

Paper 1: Flit-Reservation Flow Control [144]

Paper 2: FC3D: Flow Control-Based Distributed Deadlock Detection Mechanism for True Fully Adaptive Routing in Wormhole Networks [143]

Lecture#18: Network on Chip

Course: Computer Architecture III
Lecturer: Miodrag Bolic
Scribe: Steve Lamontagne, Mathieu Laroche
Date: November 10th, 2006

Introduction

This lecture gives a peek at a new approach to System on Chip design, called Network on Chip (NoC). NoC is believed to be the next trend in VLSI designs. This new approach will allow different heterogeneous cores to interact with each other through a chip-level network. Anything starting from processor cores, memories, DSP cores and even proprietary IP blocks can in theory be connected to the switching network and relay information to other blocks. The bus topology currently used on SoC will soon become inefficient due to the higher amount of traffic from an increased number of IP cores on silicon.

Explanation

Why Network on Chip

It has been predicted that the number of IP/uP cores on a chip could increase drastically in the future, even over a hundred. IP cores usually interface with the exterior by using an on-chip bus such as: Avalon, Wishbone or AMBA. . Adding devices to a bus increases parasitic capacitance, degrading electrical performance. [151] Parasitic capacitance increases the delay needed for a transistor to stabilize. [152] This will affect the bus performance. Buses aren't able to sustain the increased traffic, pushing for new on-chip interconnects topologies. Global clocks are also becoming a thing of the past:

"Synchronization of future chips with a single clock source and negligible skew will be extremely difficult, if not impossible." [150]

NoC embraces the concept of locally synchronous, globally asynchronous which can save a considerable amount of power. It is especially true for heterogonous networks. Modularity is also a major advantage for NoC. IP reuse and ease of debugging, considering the size, makes NoC a very attractive solution. A network on chip should consist of at least the following elements: resources, connection topology, a routing technique, switches, network interfaces, an addressing system, a communication protocol and finally a programming model such a message passing of shared memory.

Network on Chip, an Overview

This lecture proposes several network topologies such as: scalable programmable integrated network, torus, folded torus, octagon and butterfly fat-tree. It also suggests a NoC with a mesh topology. This topology is often called Chip-Level Integration of Communicating Heterogeneous Elements (CLICHÉ), implying the processing elements and other cores are different from one another. This mesh network connects different processor cores, memory, DSP cores and other IP blocks. These blocks are each connected to one routing node, giving them access to the rest of the network. All the resource blocks that don't support the networking scheme will have to use a wrapper to adapt the incoming and outgoing messages.

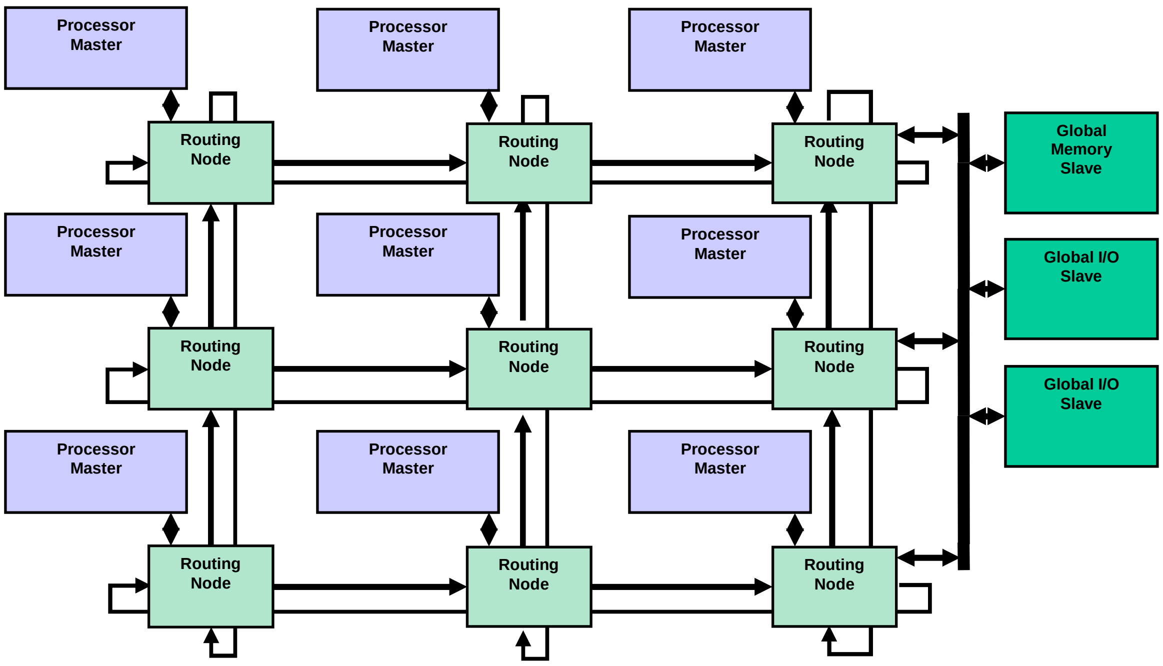


Figure 60 NoC example [151]

NoC uses the same communication paradigm as a conventional network. This means the concept of layers can be applied to raise the level of abstraction. This approach allows the decoupling of communication from the computation. NoC has a physical, data-link, network and transport layer. The physical layer ensures sampling synchronisation, voltage levels and proper connectivity. The data-link layer provides reliable data transfer between the switches. This physical network might not be reliable, so the data-link layer must provide certain mechanisms to address the problem. Contention and arbitration are also taken care of at this level. This will ensure fairness and a higher throughput. The networking layer is described in the next paragraph of this article. Finally, the transport layer decomposes and reconstructs the information. This layer is responsible for congestion control, admission control, packet retransmission and the packet size.

Although NoCs might be the next step in chip design, new challenges will arise. Network contention, silicon size of the network and IP interfacing with the network are problems that designers will have to address. Designers will have to find ways to optimize the resource utilization at the application level. Middlewares can certainly help to reach this goal if well implemented.

Switching and Routing

Switching and routing are fundamental elements for any network and NoC are not different. Although switching on a NoC uses the same paradigm as a conventional switch, it also has to submit to other rules specific to the IC industry. Since the network is in silicon, it is important to minimize the area and power consumption. For the same reasons, buffers should be the smallest possible without compromising performance too much. The switch uses one of the switching techniques such as: store-and-forward and virtual cut-through. Wormhole and deflection routing can also be used.

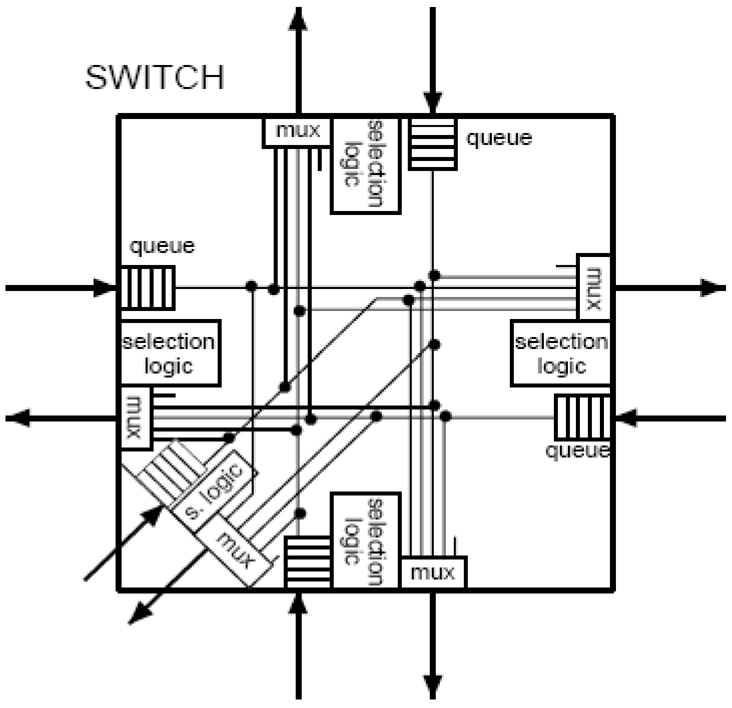


Figure 61 Switching element [150]

Figure 2 shows an example of a switching block a NoC could have. The switch is connected to four other switches and one resource. Buffers are also needed to queue the messages coming in the switch.

A good example: Blue Gene Solution

To better understand the design and requirements of complex NoC systems, we would take a look at the number one ranked supercomputer on the Top500 list of June 2006 [146]. The IBM Blue Gene solution is the result of an IBM supercomputing project begun over five years ago dedicated to building a new family of supercomputers optimized for bandwidth, scalability and the ability to handle large amounts of data while consuming a fraction of the power and floor space required by today’s high performance systems. The world’s fastest supercomputer is installed at the Department of Energy’s/National Nuclear Security Administration’s Lawrence Livermore National Laboratory (LLNL). The fully configured 64 rack, 130,000 PowerPC® Blue Gene system at LLNL has achieved an astonishing 367 peak teraflops [147].

The reasons of being the fastest supercomputer are its efficiency and the ultrascalable performance it can achieve. This was done by using a very large number of nodes that would consume low power. Blue Gene utilizes IBM PowerPC® embedded processors, embedded DRAM and system-on-a-chip techniques that allow for integration of all system functions including compute processor, communications processor, three cache levels, and multiple high speed interconnection networks with sophisticated routing onto a single ASIC[147].

We can see in Figure 62 how the system is made from the chip to the cabinets containing the racks. Each compute card has four PowerPC® 700Mhz processors that can perform four floating point operations per cycle. The programmer can decide to use both for computing or one dedicated to handling message passing operations.

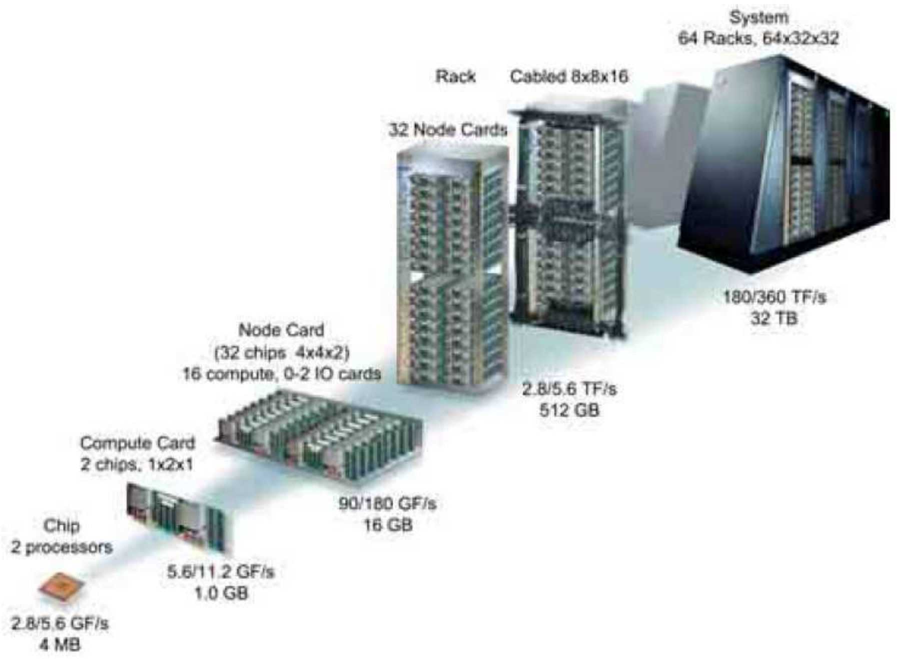


Figure 62 Blue Gene components [146]

The nodes are interconnected through five different networks:

3D Torus (175MBps in each direction)
Low latency
High bandwidth interconnections with its six nearest neighbors
Support point-to-point communication
Effective for applications with locality of communication

Collective network (350 MBps, 1.5 µsec latency)
Speed up commonly used MPI collective communications constructs

Global Barrier/Interrupt
Quickly synchronizes state across all processors in the system

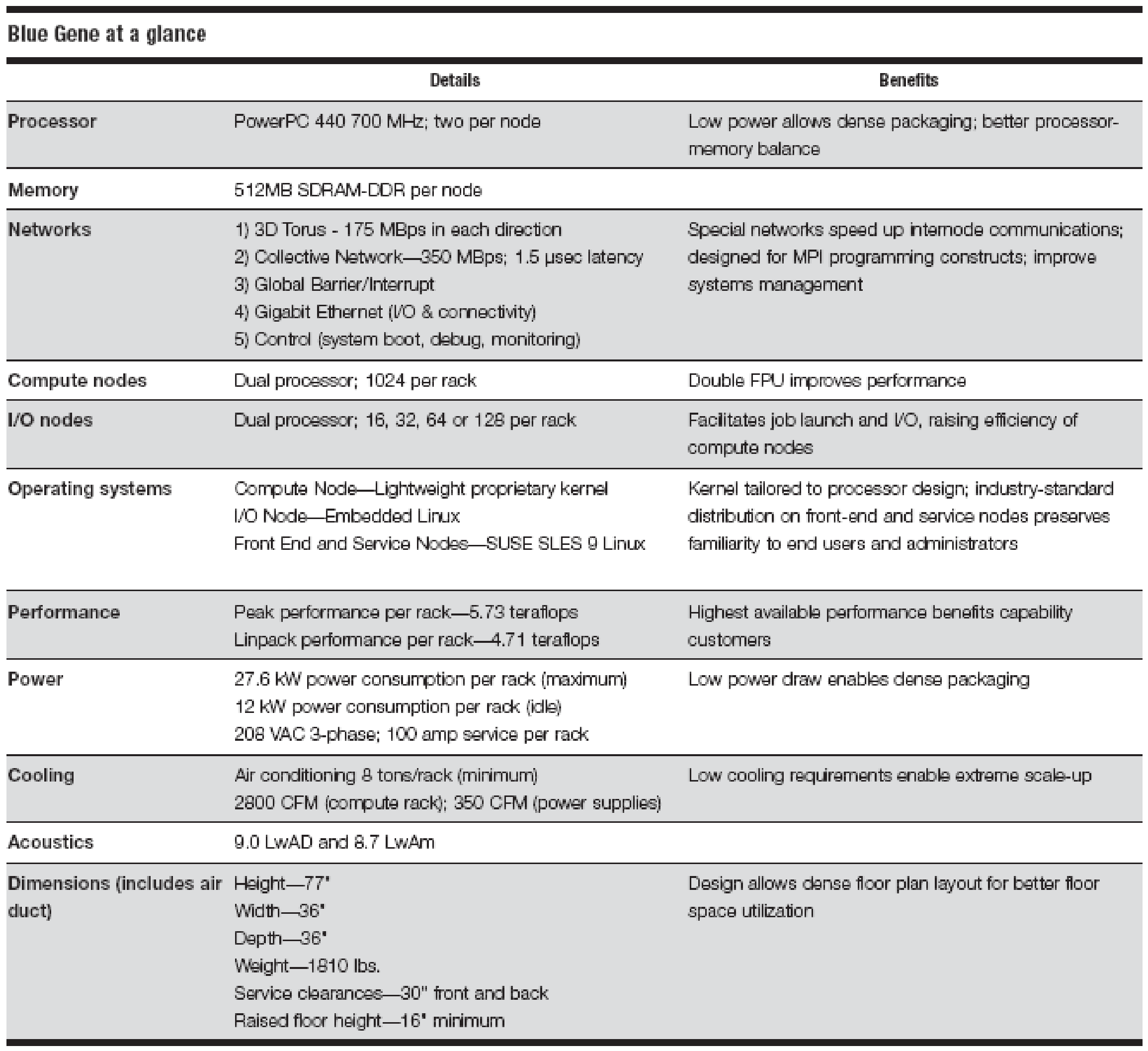
Gigabit Ethernet
I/O and connectivity

Control
System boot
Debug
Monitoring

The usage of different networks optimizes the efficiency of the communication between the nodes and results in better efficiency of the system.

The front-end node runs on SUSE SLES9 Linux®. The programmer can compile, debug or run programs. Those one can be in standard IBM XL Fortran, C and C++. It also includes a terminal where the administrator can modify system configurations, initialize and deal with the operation of Blue Gene.

IBM supercomputer addresses two big problems with supercomputer: high cost and high power consumption. In creating a new paradigm of supercomputers optimized for massive parallelism, the new architecture offers an unprecedented scalability and an ability to handle large amounts of computation while consuming a fraction of the power and floor space required by today's fastest systems [148]. By regrouping a team of smart people and using innovation, it was possible to design a powerful system with high parallelism to provide the possibility to do complex calculations in different fields (molecular dynamics, hydrodynamics, etc.).



Blue Gene at a glance

	Details	Benefits
Processor	PowerPC 440 700 MHz; two per node	Low power allows dense packaging; better processor-memory balance
Memory	512MB SDRAM-DDR per node	
Networks	1) 3D Torus - 175 MBps in each direction 2) Collective Network—350 MBps; 1.5 µsec latency 3) Global Barrier/Interrupt 4) Gigabit Ethernet (I/O & connectivity) 5) Control (system boot, debug, monitoring)	Special networks speed up internode communications; designed for MPI programming constructs; improve systems management
Compute nodes	Dual processor; 1024 per rack	Double FPU improves performance
I/O nodes	Dual processor; 16, 32, 64 or 128 per rack	Facilitates job launch and I/O, raising efficiency of compute nodes
Operating systems	Compute Node—Lightweight proprietary kernel I/O Node—Embedded Linux Front End and Service Nodes—SUSE SLES 9 Linux	Kernel tailored to processor design; industry-standard distribution on front-end and service nodes preserves familiarity to end users and administrators
Performance	Peak performance per rack—5.73 teraflops Linpack performance per rack—4.71 teraflops	Highest available performance benefits capability customers
Power	27.6 kW power consumption per rack (maximum) 12 kW power consumption per rack (idle) 208 VAC 3-phase; 100 amp service per rack	Low power draw enables dense packaging
Cooling	Air conditioning 8 tons/rack (minimum) 2800 CFM (compute rack); 350 CFM (power supplies)	Low cooling requirements enable extreme scale-up
Acoustics	9.0 LwAD and 8.7 LwAm	
Dimensions (includes air duct)	Height—77" Width—36" Depth—36" Weight—1810 lbs. Service clearances—30" front and back Raised floor height—16" minimum	Design allows dense floor plan layout for better floor space utilization

Figure 63 Blue Gene at a glance [147]

Article Synopsis: Networks on Chips A New SoC Paradigm

In the article “Networks on Chips: A New SoC Paradigm” [150], the authors discuss about a new design approach of SoCs by borrowing models, techniques, and tools from the network design field. Both SoCs and Network designs have similar requirements such as reliability, performance and energy bounds. This is why they suggest using the micro network stack paradigm (Figure 64).

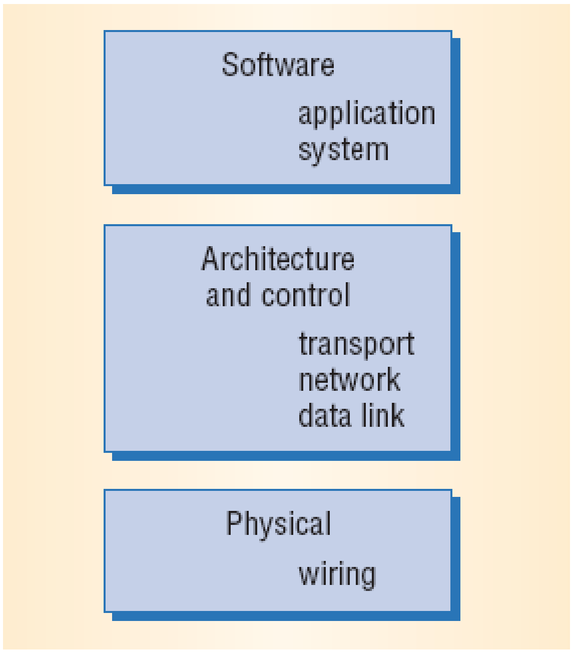


Figure 64 Protocol stack from which the micronetwork stack paradigm can be adapted. Bottom up, the layers span increasing design abstraction levels. [150]

One major difference between SoCs and wide area networks is the energy consumption issue which has to be taken into account for SoCs when designing such system. Another unique characteristic to SoCs is the design-time specialization. As opposed to macroscopic networks, the fabric of silicon is designed from scratch so only the abstract end node interface needs standardization.

This architecture is usually used in SoCs due to its simplicity. The problem with this approach is that only one device can access the bus causing delays as other are waiting for their turn. For this reason, it is not very scalable when multiple processors are connected to it. Energy efficiency is another issue as all the data are sent to every processor and would need a lot of energy to achieve broadcast all over the bus, especially with several hundreds of nodes:

“Future integrated systems will contain tens to hundreds of units generating information that must be transferred. For such systems, a bus-based network would become a critical performance and power bottleneck.” [150]

The direct and indirect networks offer a more scalable solution. Several nodes are connected together for fast communication and use a router to send data with direct link to other groups of processors. A major advantage of direct network is the total communication bandwidth also increases when the number of nodes in the system increases. In an indirect or switch-based network, the communications between any two nodes has to be carried through a number of switches. Each of those nodes has a network adapter that connects to a network switch which is used to setup a connection path.

Since we already discussed the control layers (data link, network and transport) earlier, we will discuss the software layers of the paradigm. The lower layer of that part is the system layer. Authors of the article believe that system software will be designed as a modular distributed system. Each programmable component will be provided with system software to support its own operations, manage its communications with the micronetwork, and interact effectively with neighbouring components' system software. It should provide hardware abstraction.

The highest layer, the application software, should preserve portability of the application over different platforms and provide some intelligence to leverage the distributed nature of the underlying platform.

Summary

Finally, the authors will end their article believing that a layered-micronetwork design methodology will likely be the only path to mastering the complexity of SoC designs in the years to come.

Paper 1: A Network on Chip Architecture and Design Methodology

Paper 2: "Performance Evaluation and Design Trade-Offs for Network-on-Chip Interconnect Architectures" [154]

Lecture #19: Scheduling and Dependence

Lecturer: Miodrag Bolic
Scribes: Benoit Pagé-Guitard, Anne-Audrey Loé
Date of Lecture: November 15, 2006

Introduction

This lecture provides an overview of scheduling issues involved in distributing tasks on multiprocessor systems. This is one of the most complex and often unsolvable problems in such systems. Indeed, the complexity involved in dealing with algorithms that would schedule tasks optimally is far too great, thus forcing us to take a heuristic approach when implementing scheduling in real-world applications.

In order to cover this rather vast topic as efficiently as possible, the discussion is grouped into the following subjects: the scheduling model, performing scheduling without considering communication, communication models, performing scheduling while considering communication, and heuristic algorithms.

Explanation

Scheduling Model

When considering the scheduling of tasks on a multiprocessor system, we need to take into account the following components of the scheduling model: program tasks, the target machine, and the schedule itself.

The program tasks can be defined in terms of a system (T, <, D, A) as follows:
T = {t1, …, tn} : a set of tasks to be executed
< : the partial order which specifies the precedence of tasks (i.e. t1 before t2, t4 before t5 and t6, etc.)
D : a matrix defining the amount of data that is required to be transferred to and from each respective task in order to achieve successful execution of the system (i.e. total data transferred between tasks t4 and t6, t4 and t5, etc.)
A : the amount of computations required for each task
These tasks represent the work that needs to be performed.

The target machine, on the other hand, can be defined as a set of m processing elements P_i having speed S_i. Also, these processing elements are connected between each other in a predefined fashion, with the link speed between any two PEs being defined as $R_{i,j}$, where i and j denote the identification number of each respective PE.

Finally, the schedule can be defined as a function that indicates, for each task, which PE it will execute on and at which time its execution will begin. Determining this schedule is the entire point of the scheduling exercise.

Also, it is important to define two measures for calculating execution time and communication delay [155]. First, the execution time of task t_i on processing element P_j is defined as: $\mathbf{A_i / S_j}$. Secondly, the communication delay between tasks t_i and t_j running, respectively, on processor elements P_v and P_w is defined as: $\mathbf{D_{ij} / R_{vw}}$

When analyzing a scheduling system, we will wish to consider both the quality of the schedule and the quality of the scheduler. [155]

Scheduling Without Communication Delay

It is practical to first investigate how we would go about scheduling tasks if communication delays could be ignored (i.e. made to be zero).

To do this, we'll first introduce the concept of a Gantt chart, followed by introducing in-forest and out-forest task graphs. Finally, we'll look at an algorithm for scheduling in/out-forest graphs while ignoring communication delay.

3.1 Gantt Chart

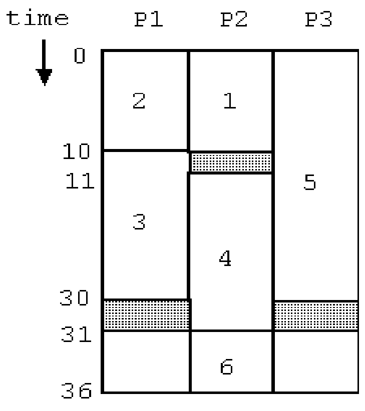


Figure 1: A Gantt chart [155]

The layout of such a chart is simple. The individual processing elements are indicated on the horizontal axis, and the time values on the vertical axis. The various tasks (numbered 1 through 6) are scheduled in their respective timeslots on the three processors. The actual positioning of these tasks is a result of applying one of many scheduling algorithms. Additionally, the grey areas between tasks represent communication delays (although we are abstracting these for now).

3.2 In/out-forest

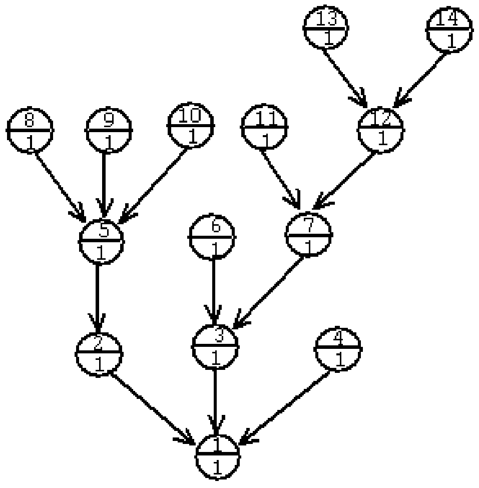


Figure 2: An in-forest [156]

An in-forest is a task graph for which any node has at most one successor. Similarly, an out-forest is a task graph for which any node has at most one predecessor. [155]

Generally speaking, the shape of the graph (collapsing inwards, or growing outwards) is sufficient to give a clue as to whether we are dealing with an in-graph or an out-graph.

The use of such graphs lies in visualizing the dependence relation between various sub-tasks in a large execution block.

In/out-forest algorithm

The algorithm for scheduling In/Out-Forests is as follows [155]:

The priority of each node is defined as its height in the graph (the vertical distance from the lowest node)

Whenever a processor becomes available, assign to it the next available task which has the highest priority

This algorithm schedules in/out-forests in the most efficient way (given an arbitrary number of processors). It is thus useful for handling large number of sub-tasks that have clearly-defined relations amongst themselves.

It should be noted that this algorithm will not work for “opposing” forests (i.e. forests that are neither in nor out). It can even be proven that scheduling such forests is impossible.

Communication Models

There are three primary models used to evaluate the total completion time of a parallel program when considering communication delay.

Two of these models (Model A and B) consider the completion time as being the sum of the execution time (without communication) and the communication delay, and the third (Model C) considers the completion time as being the schedule length (as illustrated in a Gantt chart) after tasks have been scheduled with communication delay considered.

Model A

In this model, the total communication delay is defined as the communication delay per message, multiplied by the amount of edges on a task graph with nodes on different CPUs. [155]

Basically, it considers a task graph as a whole and identifies every edge which connects tasks that are not executing on the same processing element. The number of edges identified, multiplied by the communication delay per edge (considered to be constant) will give the total communication delay.

The total program completion time is thus: Execution time + Total communication delay
Model B

In this model, the total communication delay is defined as the number of processors for each task for which that processor is not the one that computes said task, but does compute at least one of the immediate successors of this task. [155]

For example, let’s consider task A as being a successor to task B. If task A is not executed by the same processor as task B, then there will be a communication delay. We need to repeat this check for every processor on every task in the graph. The total will be the communication delay.

Again, the total program completion time is: Execution time + Total communication delay

Model C

In this model, we consider the total execution time of the system including communication in one swift procedure.

The model is simply a representation of communication as being an asynchronous operation that occurs at the same time as processing. For example, if task A has just finished executing on PE1, and task B is scheduled to be executed on PE2 while task C is simultaneously scheduled for PE1, task C can start executing immediately while communication takes place towards PE2 for task B. Once communication is complete, task B can immediately begin execution on PE2.

If we render the complete schedule on a Gantt chart, we can consider the schedule length of this Gantt chart as being the total program completion time.

Scheduling With Communication Delay

There exist two algorithms for scheduling with communication delay. One considers a task graph which is either an in-forest or an out-forest, and the other considers a task graph which is an interval order.

Choosing the right algorithm depends on choosing the correct way for representing the task graph at hand. This problem is outside the scope of the material.

Algorithm for scheduling in/out-forests

The algorithm for scheduling in/out-forests while considering communication delay can be found in [155].

Algorithm for scheduling interval order graphs

The algorithm for scheduling interval order graphs while considering communication delay can be found in [155].

Heuristics

Given the complexity of optimal algorithms, it is often impractical to employ them in real-world scheduling systems. Thus, we make use of heuristics to approximate an optimal solution. A heuristic can be judged both on its precision (relative to the optimal solution), and the speed with which it accomplishes its approximation. [155]

An optimal heuristic excels both in precision and speed; although both of those criteria are usually mutually exclusive factors, thus resulting in generally imprecise or slow algorithms.

To properly investigate heuristics, we have to consider some of its principal topics.

Parallelism versus communication delay

A heuristic cannot blindly consider parallelism as something that needs to be maximized. Indeed, there are situations when parallelism is not to our advantage. For example, consider Figure 3.

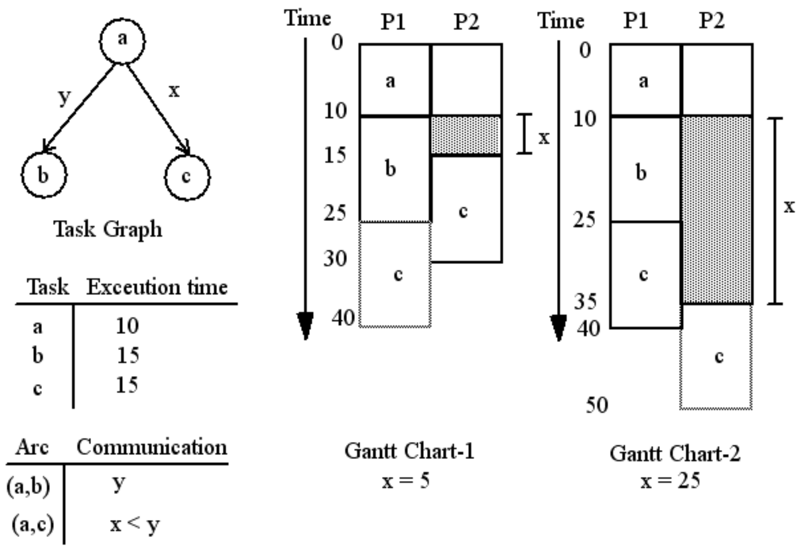


Figure 3: A communication diagram with Gantt Charts [155]

The first Gantt chart has been scheduled with communication delay x = 5, and the second has been scheduled with communication delay x = 25. If a heuristic were to always favour parallelism, we would lose efficiency if task c were scheduled on P2 when x = 25.

Instead, task c should have been scheduled immediately after task b on P1. The gains from parallelism did not outweigh the loss from communication delay in this case.

Thus, heuristics should make some level of effort in considering communication delay when deciding on the level of parallelism for a set of tasks.

Grain size and data locality

Grain size is the concept of determining how finely (or coarsely) we should partition our program into smaller tasks. Data locality relates to the efforts made in keeping the data from each grain "separate" from the data in other grains. It is a qualification of the level of independence between the various grains. A heuristic should carefully consider these issues when deciding on scheduling, as both grain size and locality can affect performance greatly.

Choosing a grain size which is too large (to save on context switching and data transfer costs) limits the benefits we can gain from parallelism. On the other hand, if the grains are too small, we will be spending more time switching between tasks than actually concentrating on executing these tasks.

Also, choosing grains based on data locality requires us to consider the nature of the data which is required for the grains' computations. It is important to choose grains which are as "data-independent" as possible (i.e. the data from one grain is neither related nor necessary for computations in the next grain). Minimizing the interdependence of data will reduce communication delay between tasks, as less data will need to be transferred to achieve parallelism.

Nondeterminism

There are times when we are unable to determine the exact path of execution of a program ahead of time. This type of situation is called nondeterminism. An example of this is looping and branch statements. We don't always have the luxury of knowing when loops will terminate, nor whether certain branch statements will be executed or not. The optimal approach, in this case, would be to schedule the program in an entirely dynamic way. Unfortunately, this leads to unacceptable performance losses.

The heuristic approach is to attempt to limit dynamic scheduling by performing as much static approximations as we can beforehand. This is possible on sections of code that are almost fully predictable. There will be a remaining possibility of error due to the prediction's imprecision. However, the gains in performance will outweigh the cost of this error.

Priority-based scheduling

Given scheduling algorithms that use task priorities to distribute work, it would make sense to assume that these priorities need to be generated in real time. Indeed, some heuristics exist which accomplish this task in a reasonable fashion.

One approach is to prioritize the node in a task graph which is farthest from the terminal node. This will guarantee that work which is being held up further down the graph (due to dependence) will be freed up as soon as possible. Similarly, another (more cost-effective) approach is to first schedule tasks which have the greatest amount of immediate successors. This does not consider the graph as a whole, and thus may be slightly more inefficient in attributing priorities. However, the algorithm is much faster than one which looks at the entire graph.

Clustering

Clustering is a heuristic process which consists in grouping sets of tasks together in clusters, with the ultimate goal being to group together tasks that have the least amount of interdependencies, and the highest communication delays.

An example of a clustering process is the following:

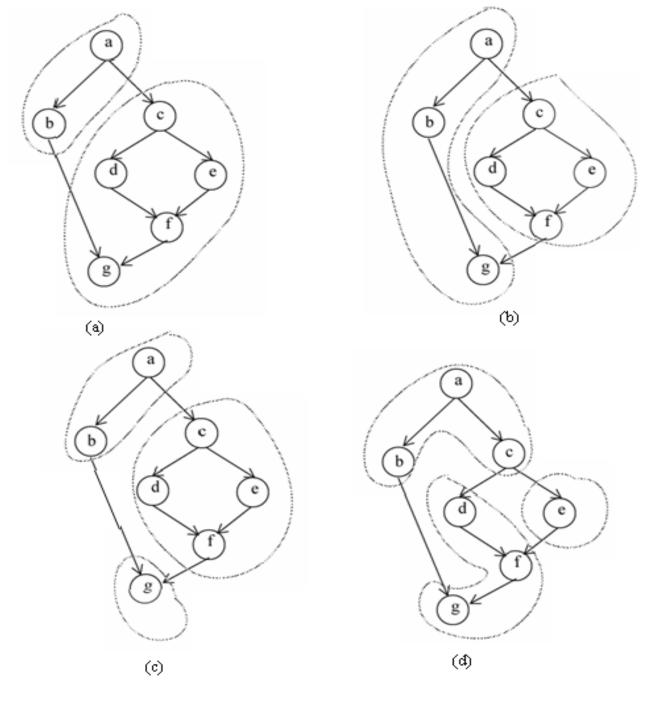


Figure 4: Clustering [155]

Individual clusters are executed exclusively on the same processing element, thus rendering the communication delay between two internals tasks equal to zero.

The main goal of the algorithm which determines the shape of the clusters is to maximize parallelism. Indeed, if we minimize the amount of "waiting" due to dependence issues (by separating children of a parent task), and concentrate on achieving minimal communication delay between clusters (by grouping the most data-intensive tasks together), we will have achieved an optimal cluster graph.

Task duplication

It is sometimes easier to do more in order to do things faster. This may sound counter-intuitive at first, but consider the following example:

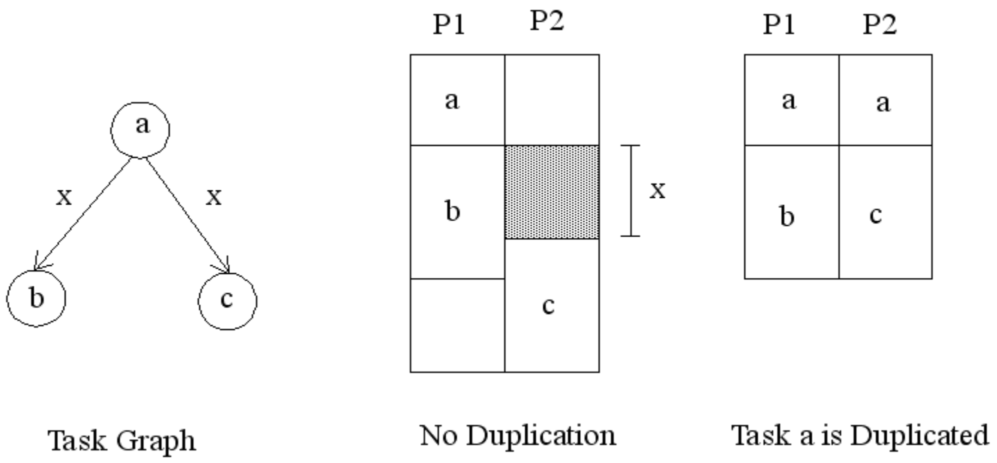


Figure 5: Duplication [155]

Without duplication, we notice that task C suffers communication delay ‘x’ between the completion of task A, and the start of its own execution. While this is still more efficient than scheduling C after B on the same processor, we have not gained all that we could have gained from parallelism. In order to increase our gains, we can schedule a copy of task A to be run on processor 2. Indeed, since A will always produce the same output from the same input, we can safely run a copy of A on processor 2 in order to have the data output from A available locally to C once it has completed. As we can see, a performance gain (equal to the communication time ‘x’) has been achieved.

Heuristics should thus consider duplication as a viable choice when contemplating optimal scheduling. However, it should not implement this blindly, as performance gains can easily (and quickly) be offset by the communication costs involved with duplicating tasks. [155]

Summary

As we have seen, there are numerous factors to consider when performing scheduling on multiprocessor systems. It is easily one of the most complex (and imprecise) topics in the field of computer engineering, and is inherently problematic when attempting to analyze in an academic setting.

It is generally accepted that the potential for research lies mostly in the area of heuristics, as the impracticality of optimal scheduling has been proven time and time again. Nonetheless, it is important to consider the fact that there is always the possibility for applying optimal scheduling in real-world applications (perhaps by using a dedicated scheduling processor).

Paper 1 Deterministic Processor Scheduling

Paper 2 Scheduling Problems for Parallel and Distributed Systems

Lecture #20a: Vector Processing and Vector Architectures

Course: Computer Architecture III
Lecturer: Miodrag Bolic
Scribes: Evan Leclair (293 2223), Sumeet Abrol (286 9998)
Date: November 17, 2006

Introduction

Previous lectures have covered the concepts relating to achieving parallelism using multiple processing elements. This involved general discussions about system level architectures. Specifically, solutions were investigated for connecting processing elements to their memories and for establishing an effective communications architecture. Programming techniques for these architectures were also examined.

Here, the focus changes towards realizing parallelism using a single processor. The conventional solutions to this problem include superscalar processors, very large instruction word processors (VLIW), and vector processors. This lecture explicitly covers the topic of vector processors.

Section 1 introduces the concept of vector processing, while Section 2 highlights typical characteristics and properties of a vector processing system. Section 3 investigates the general architecture of a vector processor. Section 4 covers programming for vector processors, and finally, Section 5 provides performance analysis.

Explanation

1 Vector Processing

There are different approaches taken in exploiting parallelism on a single processor. As stated, there are three main classes of such processors: superscalar processors, VLIW processors, and vector processors. Superscalar and VLIW processors make use of instruction level parallelism (ILP), which utilizes the potential of executing short instruction sequences in parallel [159]. ILP is achieved through the use of pipelining (overlapping instructions), superscaling (executing multiple instructions per clock cycle), and out of order execution [159].

In both the superscalar and VLIW approach, a single instruction is executed with single elements as operands, and the result (also a single element) is stored in a register. In contrast, vector processors exploit data level parallelism (SIMD) and perform operations on sets of data, or vectors. Figure 1 illustrates this difference. The scalar operation uses the values in two registers and stores the result in another, while the vector operation uses values in a register bank (a set of registers) and stores the result in another register bank.

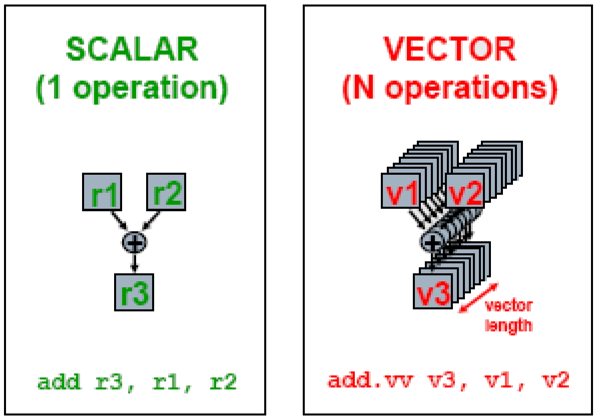


Figure 1: Scalar vs. vector operations [159].

The basic idea behind vector processing is that applications that must process large sets of data can execute a single instruction that takes entire vectors as its operands. Note that the entire vector instruction does not compute in the same number of clock cycles as the corresponding scalar instruction; the individual vector element operations are computed sequentially, not in parallel (although the element operations of the instruction may be pipelined). Despite this, a single vector instruction performing *n* operations and taking vectors with *n* elements as operands still computes in less clock cycles than *n* scalar instructions performing the same operation. This is due to the fact that the vector instruction requires a single decode for all *n* vector elements, whereas the *n* scalar instructions require *n* separate decodes. Moreover, a single instruction is fetched from main memory into the CPU prior to the computational instruction in the case of the vector instruction. In comparison, *n* scalar instructions must be fetched by the CPU from main memory or cache. A similar comparison can be made with the instruction operands themselves. There is a single load operation from main memory to registers (over a range of memory) for each vector prior to the computational instruction. In comparison, the *n* scalar computational instructions require *n* separate loads to occur (one load per operand). Finally, if the *n* scalar operations are computed in a loop, extra overhead is incurred due to the incrementing of indices and checking of boundaries. These facilities are inherently supported in vector processors.

Vector processors are particularly useful for large scientific and engineering applications that typically consist of very large data sets (for example, car crash simulations and weather forecasting) [160]. Multimedia applications, such as graphics processing, also greatly benefit from being executed on vector processors [161]. The common theme in most of the applications is a “number crunching” service that must be performed.

As a note of interest, the Cell processor used in the Sony PlayStation 3 and developed by Sony, Toshiba, and IBM consists of a general purpose RISC processor and streamlined vector coprocessors that are utilized to accelerate, among other things, graphics processing [162].

2 Characteristics of Vector Processors

There are a number of characteristics and properties that are typical of vector processors. These are highlighted below:

The result of any operation on a vector element is independent from the result of the previous operation [159]. In other words, no data dependencies exist between elements within a vector. This is essential for vector processing to take place. The compiler is assigned the task of ensuring that this condition holds [159].

- The clock rate of a vector processor is typically high, as are the number of pipeline stages for any given instruction [159]. This allows the elements of a vector to be efficiently pipelined for the given operation.

- Vector processors contain highly interleaved memories [159]. The data elements of a vector are arranged amongst memory banks in such a way that many elements can be accessed at once by addressing the same location in each memory bank. This, of course, leads to an increase in performance.

- There are no data caches implemented in vector processors [159]. The idea is that the system should not rely on constructs as unpredictable as caches. In any case, there is no need for data caching as temporal locality is usually very low for the kinds of applications running on vector processors. Take for example an imaging application in which input data to a graphics vector processor is used to produce a rendered image. The input data to the processor is always new, and the initial data will never be accessed again.

3 Vector Architecture

There are two main kinds of architectures used in vector processors. They are called vector-register processor and memory-memory vector processor architectures [163]. The vector-register architecture is similar to the RISC processor architecture in that all operations are performed using operands from CPU registers (other than load and store). In the memory-memory architecture, all operations are performed using direct access to main memory. The discussion that follows pertains mainly to the vector-register architecture.

The basic structure of the vector-register architecture is given in Figure 2. There are four main components of interest: the vector load-store, the vector registers, the scalar registers, and the vector functional units (operational blocks to the right in Figure 2).

1) <u>Vector load-store</u>
This is a vector memory unit that loads vectors into the vector registors from main memory or stores them to main memory from the vector registers [163].

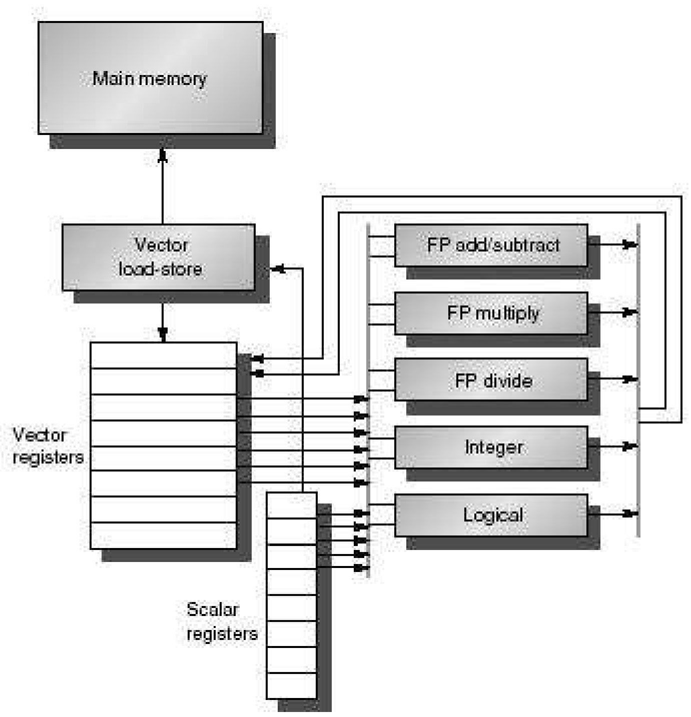


Figure 2: The basic structure of the vector-register architecture [163].

2) <u>Vector registers</u>

Taken together, the vector registers compose a vector register file. In this case, the vector register file consists of eight vector registers. Each vector register in turn contains 64 registers of size 64 bits. A RISC processor typically contains only 64 registers in total, each one with a size of 32 bits. The vector register file is depicted in Figure 3.

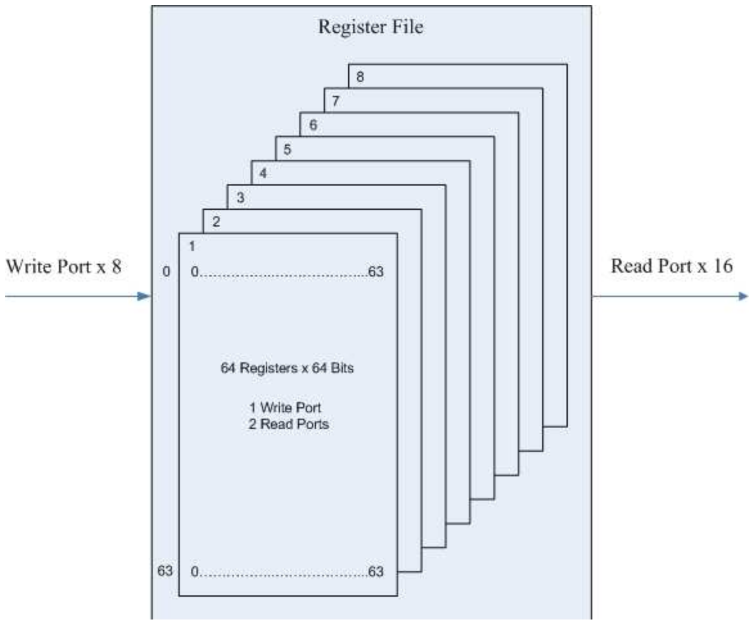


Figure 3: The vector register file and its eight vector registers.

Each vector register is designed so that it has one write port and two read ports [159]. This gives a total number of 16 read ports and 8 write ports for the vector register file as a whole. A vector (with 64 elements, for example) is stored in a vector register with each element of the vector being stored in a single register. The vector registers provide input to the vector functional units.

3) Scalar registers

The scalar registers are the normal 32 general-purpose registers found in RISC processors [163]. They can be used for providing input to the vector functional units and for computing addresses to be passed to the vector load-store unit.

4) Vector Functional Units

The vector functional units are the components that actually compute the vector operations. Each unit is fully pipelined and can begin a new operation (on vector elements) each and every clock cycle [163]. Floating point multiplication can reach up to 20 pipelined stages, while floating point addition is usually between 4 and 6 stages. It is important to note that the vector functional units are all in parallel.

The vector-register architecture can be viewed as a large extension of the general RISC architecture with facilities for performing instructions on sets of data. The architecture allows for data to be output to the vector functional units from the register file or from the scalar register. Furthermore, it allows pipelining to occur from one functional unit to another in a technique known as chaining. This will be discussed in greater detail in Section 5.

If the floating point functional units from Figure 2 are the only ones to be considered, there are seven points from which the register file can be read (two per floating point unit and one for the store unit). The problem then becomes connecting the 16 possible outputs (read ports) from the register file to these seven points. This is done using a 16 x 7 crossbar network. A similar occurrence is observed at the register file's input (write ports). There are eight possible inputs to the register file and only 4 points from where data can be written (one per floating point unit and one for the load unit). These can be connected using an 8 x 4 crossbar network.

Figure 2 gives the basic architecture for a vector-register processor and clearly leaves out much of the detail. There are, however, two important hardware additions that should be discussed: the Stride register and the Vector Length register (VLR) [163].

It might be the case that the positions in memory of successive elements of a vector are not sequential. In this case, it may be required that every i^{th} position in memory be considered a vector element. The distance separating elements to be gathered into a single register is called the stride, and the Stride register is used to store this value. Figures 4a, 4b, and 4c give examples of instances where the stride value would be used. Consider a vector stored sequentially in main memory representing an 8 x 8 matrix. Figure 4a shows memory being accessed with a stride equal to one in order to obtain a row vector. Figure 4b shows memory being accessed with a stride equal to 8 in order to

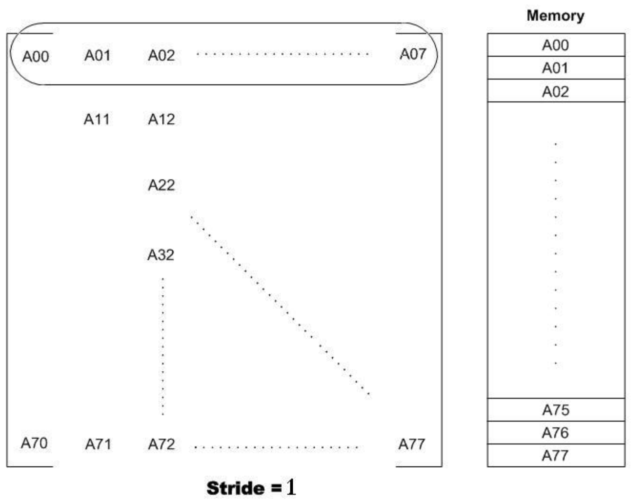


Figure 4a – Row accessed with stride = 1.

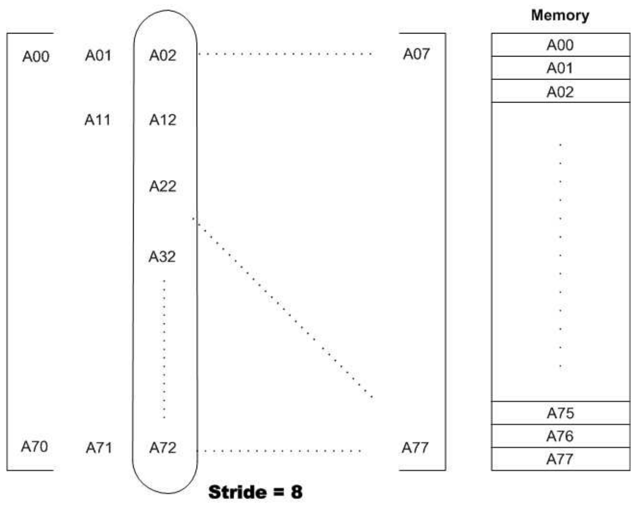


Figure 4b – Column accessed with stride = 8.

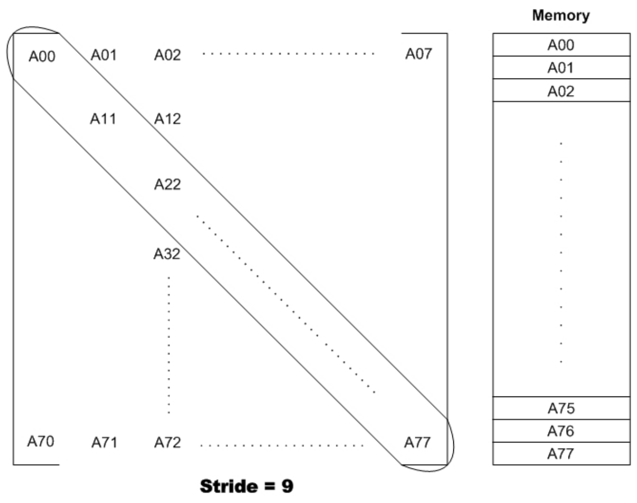


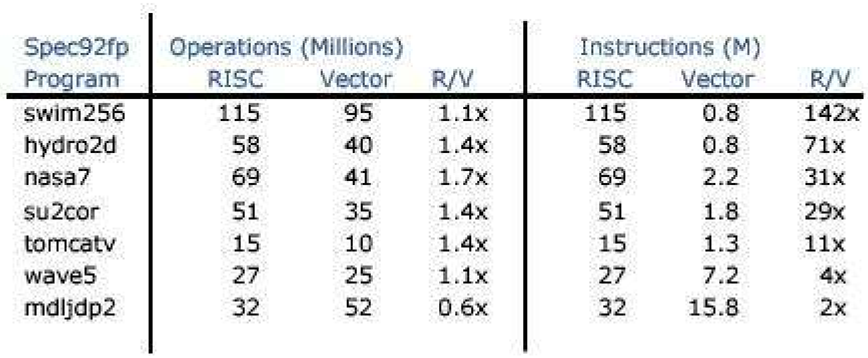
Figure 4c – Diagonal accessed with stride = 9.

obtain a column vector. Finally, Figure 4c shows memory being accessed with stride equal to 9 in order to obtain the diagonal through the matrix.

Finally, the Vector Length Register (VLR) is used to specify the size of the vector to be used (i.e. the number of elements in the vector). It is likely that the number of registers in the vector register will not exactly match the actual vector length in all applications. Therefore, the VLR is used to control the length of any vector operation [163].

4 Vector Processor Programming

Programs written for vector processors possess characteristics that are much different than those written for RISC based processors. A table is provided in Figure 5 comparing the number of operations and instructions on RISC based and vector processors under a variety of benchmarks. Looking at the *Instructions* column, one observes a striking decrease in the number of instructions for the vector processor's program in comparison to the RISC processor's program. This is due to the fact that a single vector processor instruction performs many more operations than that of the RISC processor's. As a result, the RISC processor must perform more instructions in order to execute the same number of operations.



Spec92fp	Operations (Millions)			Instructions (M)		
Program	RISC	Vector	R/V	RISC	Vector	R/V
swim256	115	95	1.1x	115	0.8	142x
hydro2d	58	40	1.4x	58	0.8	71x
nasa7	69	41	1.7x	69	2.2	31x
su2cor	51	35	1.4x	51	1.8	29x
tomcatv	15	10	1.4x	15	1.3	11x
wave5	27	25	1.1x	27	7.2	4x
mdljdp2	32	52	0.6x	32	15.8	2x

Figure 5: RISC vs. vector processors under Spec92fp benchmarks [159].

Looking at the *Operations* column of the table, one observes a small increase in the number of operations that the RISC processor performs when compared to that of the vector processor. This extra overhead is due to the fact that indices are incremented and boundary checks are performed in the RISC program (executed in a loop). These facilities are inherently supported in vector processors.

Before analyzing an actual sample program written for execution on a vector processor, some typical vector instructions will be introduced. These are presented in Table 1 (instructions taken from [163]). Note that the forms of the instructions are very similar to those found in RISC architectures. Keep in mind, however, that the operands for these instructions are vectors and not single data elements (with the exception of scalar addition and scalar multiplication which take a scalar as one of the operands).
Table 1 – Summary of Useful Vector Instructions

instruction	operands	description

addv	v1, v2, v3	Vector Addition: Add elements of v2 and v3, then put each result in v1.
addvs	v1, v2, f0	Scalar Addition (with elements of a vector): Add f0 to each element in v2, then put each result in v1.
mulv	v1, v2, v3	Vector Multiplication: Multiply elements of v2 and v3, then put each result in v1.
mulvs	v1, v2, f0	Scalar Multiplication (with elements of a vector): Multiply each element of v2 by f0, then put each result in v1.
lv	v1, r1	Load Vector: Load vector register v1 from memory starting at address R1.
sv	r1, v1	Store Vector: Store vector register v1 into memory starting at address R1.
lvws	v1, (r1, r2)	Load Vector With Stride: Load v1 from address at R1with stride in R2.
svws	(r1, r2), v1	Store Vector With Stride: Store v1 into address at R1with stride in R2.

Now, the code of an example program will be compared under a RISC processor and a vector processor. The program's task is to compute the following:

$Y = a*X + Y$, where *X* and *Y* are vectors of size 64 and *a* is a scalar value.

In the case of the RISC processor, the scalar value *a* is loaded and kept constant for the duration of the program. The values at *X(i)* and *Y(i)* are also be loaded into CPU registers. Next, the computation of *a*X(i)* is performed followed by the addition of *Y(i)* to the result. This result is then stored back into main memory at location *Y(i)*. Finally, the indices to *X* and *Y* are incremented, and a counter is updated and checked to determine if the operation has completed on all elements of *X* and *Y*. The process is repeated for all elements of *X* and *Y*. The code listing for the RISC processor is given below (obtained from [159]):

```
LD F0,a
ADDI R4,Rx,#512                 ;last address to load
loop:   LD F2, 0(Rx)            ;load X(i)
MULTD F2,F0,F2      ;a*X(i)
LD F4, 0(Ry)              ;load Y(i)
ADDD F4,F2, F4            ;a*X(i) + Y(i)
```

```
SD F4 ,0(Ry)            ;store into Y(i)
ADDI Rx,Rx,#8           ;increment index to X
ADDI Ry,Ry,#8           ;increment index to Y
SUB R20,R4,Rx           ;compute bound
BNZ R20,loop            ;check if done
```

In the case of the vector processor, the same sequence of steps are taken; however, there is no *loop* for all of the elements in the vector as the instructions use complete vectors are operands. As a result, there is no need for incrementing indices or performing boundary checks. The code listing for the vector processor is given below (obtained from [159]):

```
LD F0,a                         ;load scalar a
LV V1,Rx                        ;load vector X
MULTS V2,F0,V1       ;vector-scalar mult.
LV V3,Ry                        ;load vector Y
ADDV V4,V2,V3           ;add
SV Ry,V4                        ;store the result
```

Comparing these program implementations, a few things become evident. The RISC implementation requires 578 instructions (2 for the original ld and addi, and 1 x 64 for each instruction in the loop). On the other hand, the vector implementation requires only 6 instructions. Furthermore, the RISC implementation requires the same number of operations as instructions (i.e. 578). In comparison, the vector implementation requires 321 operations (1 for the initial load, and 1 x 64 for each of the following 5 instructions). The decrease in number of operations is due to the fact that the vector program does not incur the loop overhead.

5 Performance Analysis

In order to conduct performance analysis on vector processes, the notion of vector execution time will be used. For the purpose of our analysis, it will be assumed that the vector processor described herein has vector functional units with single parallel pipelines. It is also assumed that individual operations are performed at a rate of one vector element per clock cycle. In other words, each functional unit is pipelined with a number of stages, and each stage takes one clock cycle to complete. Note that entire functional units perform in parallel [163]. Two terms are introduced in order to simplify the discussion of vector execution time: convoy and chime.

The convoy is the set of vector instructions that can be executed together in one clock cycle. In basic terms, it is a set of instructions that must complete execution before any other instructions can execute [163]. Consider a sequence of instructions S1, S2, S3, S4, and S5 where:

S1 is independent
S2 is dependent on S1
S3 is independent

S4 is dependent on S2 and S3
S5 is dependent on S4

The situation is depicted in Figure 6. S2 cannot begin executing until S1 has completed since it is dependent on the result of S1. The first convoy consists of a single instruction, S1. For the second convoy, S2 is allowed to execute because S1 has completed. S3 is also allowed to execute (within another functional unit) since it is not dependent on any other instruction. S4, however, is dependent on S2 and S3 and must execute in the following stage. Therefore, the second convoy consists of 2 instructions; namely, S2 and S3. Note that S3 could have been a part of the first convoy with S1, in which case the second convoy would only have consisted of S2. S4 is the only instruction to execute in the third convoy (S5 is dependent on S4). The same is true for the final convoy with S5. The concept of convoys is used to speed up execution by determining which instructions can be executed in parallel.

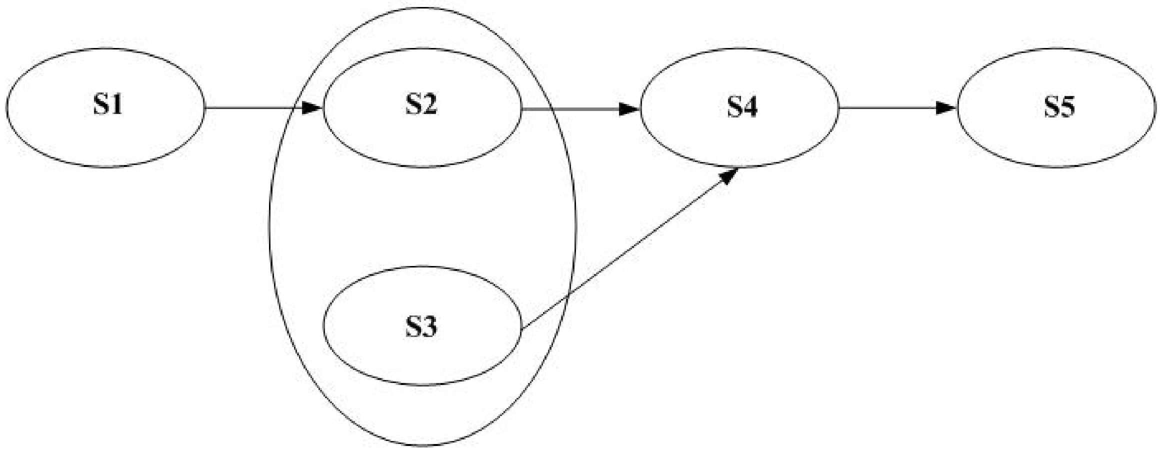


Figure 6 – The sequence of instructions for S1, S2, S3, S4, and S5.

The chime is used for estimating the performance of a vector sequence consisting of convoys, and it is defined as the unit of time taken to execute a convoy [163]. In other words, the chime can be used to determine the time needed to execute the vector operation under consideration. If a vector sequence for one element of a vector consists of *m* convoys, it will be executed in *m* chimes. If the length of the vector is *n*, and the vector sequence is performed on each element in the vector, then the time to execute is approximately *m* x *n* clock cycles. Although this method of estimating the time to execute is good when one does not wish to go into details, it is not entirely accurate. The chime model ignores the vector start-up time [163]. Initially, when a vector is passed to a functional unit, the first element must go through all of its pipeline stages before the result is available. This takes as many clock cycles as the number of stages in the pipeline (assuming one clock cycle per stage). Subsequent elements, however, will be output every clock cycle. This initial delay of output from the functional unit is called the start-up time.

Another way of improving execution time other than executing independent instructions in parallel (in a convoy) is a method called chaining. Chaining deals with making a sequence of dependent vector instructions run faster [163]. If two vector instructions must be performed sequentially on a vector (for example, multiply followed by addition), the instructions can be sped up if the result of a single vector element in the first functional unit (the multiply) is forwarded to the second functional unit (the addition). In this way, the second instruction can begin execution without having to wait for the first instruction to have completed. The situation is depicted in Figure 7. The first element of vector V1 must be processed through the stages of the multiplication functional unit pipeline. Once it has been processed through the last stage, the element is immediately forwarded to the addition functional unit for processing. After this point, a new result from the multiplication functional unit (a single element) will be forwarded to the addition functional unit on each subsequent clock cycle.

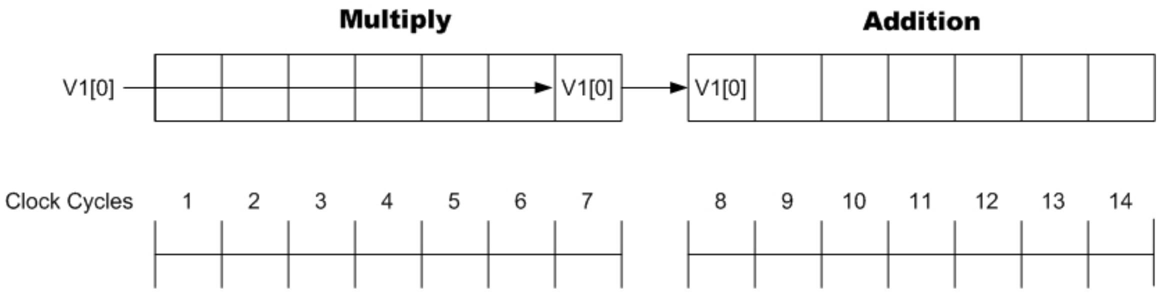


Figure 7 – Chaining a multiplication and addition instruction.

Another example is considered for a program in which a vector is loaded, multiplied with a static value, added to another vector (which has also been loaded), and the result is stored. If there existed a single time multiplexed load/store unit in hardware, chaining would be performed as follows:

```
Lv      mulvs
Lv      addv
Sv
```

However, if there existed separate load and store units, chaining would be performed as given below:

```
Lv      mulvs
Lv      addv sv
```

In the second case, the store instruction is chained to the output of the addition instruction since loading is being performed on a separate hardware unit. In the first case, the store instruction can not be chained to the addition instruction as the shared load/store functional unit is already in use for the load.

Summary

This lecture discussed parallelism using a single processor, specifically utilizing vector processors. Vector processors, unlike superscalar and VLIW processors, perform operations on sets of data, or vectors. Vector processors are particularly useful for applications that typically consist of very large data sets

The general concepts of vector processing were introduced as well as the typical characteristics and properties of a vector processing system.

The basic elements of vector architectures were investigated, highlighting the use of large register files and parallel functional units. It was determined that crossbar networks were most widely used in connecting the register file to the functional units and other system components. Also, the concept of stride was discussed to be the distance separating elements to be gathered into a single register.

Additionally, the common programming techniques used for vector processors were discussed. It was shown that the number of instructions of a program written for a vector processor was much lower than that written for a typical RISC processor. Furthermore, it was shown that the number of operations was also lower. It was determined that this was due to a smaller number of instruction loads, operand loads, decodes, and no loop overhead.

The lecture concluded with a brief overview of performance analysis of vector processing systems. Execution time was discussed using the concepts of convoy, chime, and start up time. Performance improvements were investigated through the use of parallel instruction execution (in a convoy) as well as through chaining.

Paper 1Scalable Vector Processors for Embedded Systems [164]
Paper 2 Simple Vector Processors for Multimedia Applications [165]

Lecture #20b: Vector Processors

Course: Computer Architecture III
Lecturer: Miodrag Bolic
Scribe: Mohammed Agha 2717439 – Khalid El Mously 3575624
Date: October 15, 2007

Introduction

This lecture covered the concept of Vector Processors (VPs). The topics included in the lecture consisted of:

- Vector Processor Architecture.
- Parallelism in VPs.
- VP Instructions.
- Chaining of Instructions.

The lecture discussed each of these in topics and presented the concept of parallelism using a single processor only; parallelism that is achieved by manipulating large arrays of numbers (i.e. vectors) simultaneously, using only a single instruction.
This paper discusses VPs in more detail. First, the history of VPs is discussed, and the factors leading to its development are explored. Next, VP architecture is presented, and the functionality of the basic VP hardware elements is explained. Vector programming and the instruction set is then presented; it is in this section that the optimization of instructions is also shown. Finally, the performance of VPs is shown by comparing it to other architectures running specific benchmarks.

Explanation

In the early 1960s, Westinghouse Electric, in their Solomon project, aimed to improve computer performance by creating a large number of ALU units that were to be controlled by a single CPU 0. What arose was the basis for future VP designs.
The first commercially available VPs, sold in the early 1970's, were Texas Instrument's TI-ASC and Control Data Corporation's STAR-100 [167]. These were not very successful, as the vector instructions required a significant amount of time to prepare for an instruction, and the system itself was slow because it was a memory-to-memory vector architecture 0. It was not until the introduction of the CRAY-1 in 1976 that VPs became popular [167]. The major difference between the CRAY and its predecessors was the vector registers and the interconnect paths to move data from the registers to the functional units [167]. This sped up performance considerably as memory latency was significantly reduced with the vector registers.

Since that time, until the early 1990's VPs were the leading, commercially available supercomputers [167]. This all radically changed however with the advances in CMOS VLSI technology, which allowed for a huge increase in the number of transistors that could be placed on a single die; moreover, the die could be clocked faster as feature sizes shrank [167]. This led to the eventual downfall of the VP as a supercomputer, as smaller faster VLSI chips were being made that could compete in speed with VPs, and that cost a lot less to manufacture.

Nevertheless, the concepts of VPs still live on to this day but not as complete functional systems. The vectorization of instructions does appear in some applications, particularly in the gaming industry, where the high degree of parallelism at low cost is a big attraction for systems such as the Nintendo 64 and Playstation 2 [169]. IBM's new Cell Processor also utilizes vector instructions for multimedia applications [172].

Why vector processors?

The rise of VPs came as a direct result of limitations in other architectures [169]. It has been shown, that in other architectures, scaling the design results in an exponential increase in complexity. Increasing a system's pipeline depth also requires increasing the number of independent instructions so as to keep the processor busy with useful work [169]. "For a dynamically-scheduled machine, hardware structures, such as instruction windows, reorder buffers, and rename register files, must grow to have sufficient capacity to hold all in-flight instructions, and worse, the number of ports on each element of these structures must grow with the issue width. The logic to track dependencies between all in-flight instructions grows quadratically in the number of instructions" [169]. In other words, the deeper the pipeline the more complex the hardware becomes. A simple solution was needed to solve this dilemma, and VP architecture was the answer. A number of factors contributed to making VPs suitable, these are [169]:

A single vector instruction specifies a great deal of work; it is equivalent to executing an entire loop. This reduces the instruction fetch and decode bandwidth needed to keep deeply pipelined functional units busy.

Elements in a vector can be computed using parallel functional units, or a single very deeply pipelined functional unit because there is no data dependency between elements in the same vector.

Data dependency checks need to be done only once per vector instruction, and not for every element in the vector. That means that data dependency logic for vector instruction sets require the same complexity as for scalar instruction sets, but now many more elemental operations can be in the pipeline with the same amount of logic.

Cost of the latency to main memory is seen only once per vector, and not once for every word as in scalar architectures. Thus, it is greatly reduced.

Control hazards that are inherent in loops will be eliminated, as a single vector instruction represents a loop in scalar architectures.

As can be seen though, VPs are only useful for data sets that are represented as large vectors. Data sets representing scientific simulations, meteorology, and multimedia applications are the most common. VPs are very useful for streamlining Data Level Parallelism (DLP). They are not that useful however, for an average user nowadays.

Vector Processor Architecture

Vector processors can be divided into two subcategories: memory-to-memory, and vector-to-register [168]. As discussed in the history section, memory-to-memory architectures were the first VP architectures designed.

Vector-register processors, the more common of the two, operate only on registers, and only access main memory during load-store operations [168]. Since this is the more common of the 2 architectures, we will be discussing it for the remainder of this document.

Components of a Vector Processor

All vector processors have the same basic components. They only differ in the way these components are structured, and in the number of each component there actually is in the design. These basic components are [168]:

Vector Register. Fixed length bank holding a single vector. It has at least 2 read and 1 write port. It usually consists of 8-32 vector registers, each vector containing 64-128 64-bit elements.

Vector Functional Units. Fully pipelined, and can start a new execution at every clock. Typically a system will have 4 to 8 Functional Units, consisting of: FP addition, FP multiplication, FP reciprocal, integer addition, logical operations, and shifting. A system could have multiple units of the same functionality.

Vector Load-Store Units. Fully pipelined unit used to load or store a vector. Again, a system may have more than one of these units.

Scalar Register. Scalar register for use in scalar operations, and addressing memory.

Stride Register. The value stored in this register tells the CPU the memory access order. For example, if the stride is 1 memory is accessed sequentially, but if it is 64 then every 64th element of memory is accessed.

High performance cross-bar network. Used to connect the functional units together with the registers, and load-store units.

These are the basic essential components that all VPs contain. The number of each, however, is architecture specific.

Performance Improvement

There are many performance enhancements inherent in vector architectures. It is these enhancements that

As mentioned earlier, all functional units are deeply pipelined and can begin executing an instruction at any clock cycle. When performing an addition, for example, the unit will produce a result every clock cycle[169], as the vector is fed into it. The operation would look something like that of Figure 1 below.

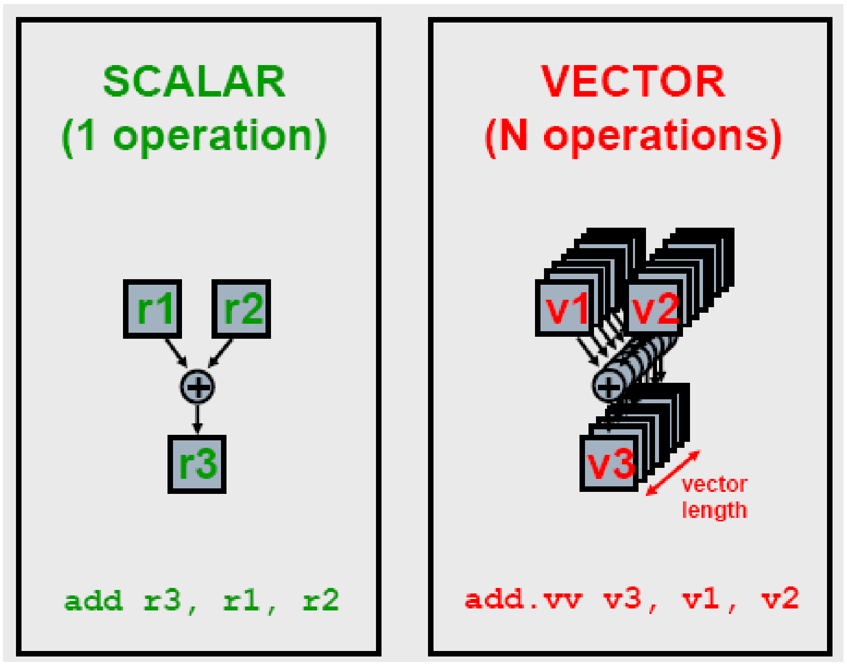


Figure 65 Scalar Instruction vs. Vector Instruction [168]

The vector instruction will produce the result v3 from the vectors v1 and v2. In addition to performing operations on whole vectors, the units have forwarding lines to forward the result directly to other functional units in the case of dependent instructions.
Consider, for example, the following (scalar) RISC algorithm that attempts to add two 50-element arrays, and store the result in a 50-element array.

```
Set i = 0
        add_loop
If (i – 50 == 0) branch to outside
Load the ith item of first array into reg1
Load the ith item of second array into reg2
Add reg1 and reg2, and place result in reg3
Store reg3 in ith element of result array
Increment i.
GOTO add_loop
        outside
```

By looking at the above pseudo code, it is seen that some instructions are repeated unnecessarily, always producing the same result (e.g. the fourth instruction in the add_loop is always add reg1, reg2 and place in reg3). One can also see the loop *overhead* (setting i, incrementing i and checking the value of i against 50). In fact, in the above example, the loop overhead is more than the actual loop *content* – causing a significant performance hit.
On the other hand, in a vector processor-version of the algorithm, memory access for instruction fetch only takes place once (as opposed to 50 times), while loop overhead is

virtually eliminated. Consider the following vector processor algorithm to achieve the same result:

```
Fetch next instruction
decode instruction
get 50 numbers of first array
get 50 numbers of second array
add them
put result in result array
```

The above pseudo code is executed only once – i.e. there are no loops, no branches, and no repeated instructions. It's easy to see the reduction in number of program instructions.

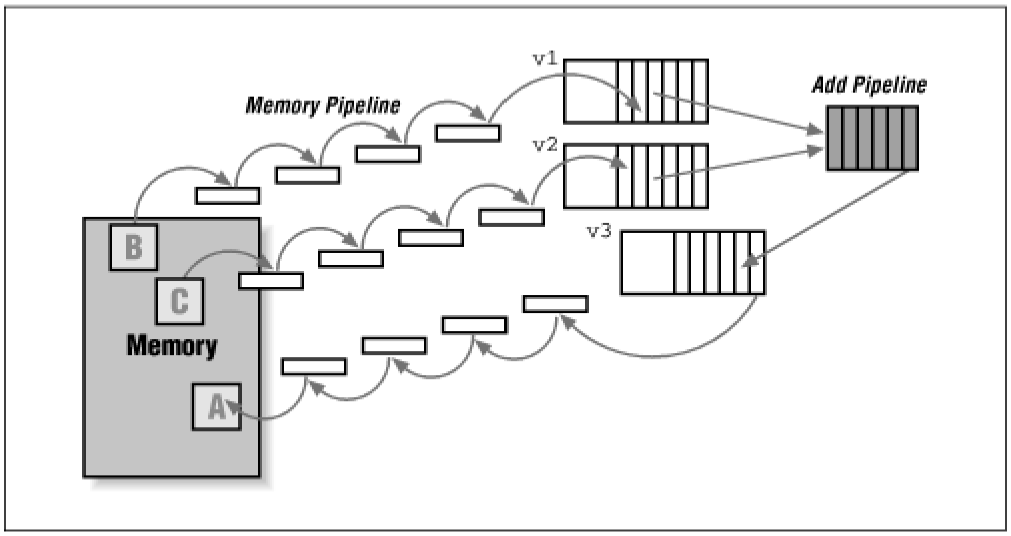


Figure 66 Abstract figure showing how memory access and arithmetic operations can be pipelined

Memory access is typically in the order of 20 CPU clocks[169]. In a vector processor, the stride register indicates where the next element of the vector is located in memory as an offset from the last accessed memory location, while the VLR register (Vector Length Register) indicates how many elements need to be brought from main memory. Using those two pieces of information, memory access can be pipelined to reduce overall memory latency.

Vector processors have a register file bank – an array of register files, each one similar to that of normal RISC processors. In the VMIPS processor, for example, there are 8 vector registers, each 64 words, and every word is 64 bits.

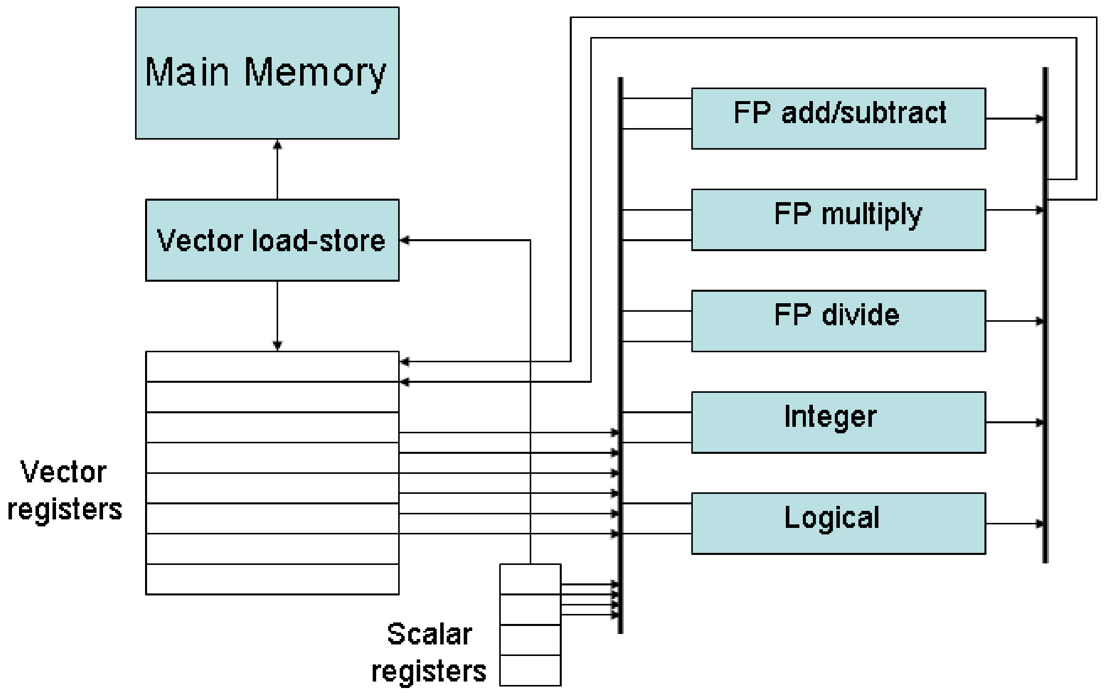


Figure 67 High level diagram of a typical vector processor. Note that there are usually multiple vector registers (only one is shown). The bold vertical line is typically implemented as a crossbar

The VMIPS vector processor has multiple functional units (add/subtract unit, logical unit, etc.). All functional units are deeply pipelined and have two input ports and one output port and all vector registers have two output lines and one input line. The output of the vector registers are connected to the input of the functional units using a crossbar (shown in the above figure as a bold vertical line). Thus, if there are 8 register files and 5 functional units, the crossbar will need to be a 16x11 crossbar (5 functional units x 2 inputs each + 1). The output of the functional units is connected back to the vector register bank input, also using a crossbar. Therefore, with the above configuration, the required crossbar for the input *to* the register file must be a 5x8 (5 outputs from the functional units connected to 8 inputs of the 8 register files).
Vector processors also have two special registers not found in ordinary RISC processors – *VL* register and *stride* register. VLR, or Vector Length Register, indicates the length of the vector register. This is necessary since it’s extremely rare that all the vectors that need to be manipulated are exactly 64 elements long. Using the VLR, we can know how many of the elements are to be processed.
The stride register is used when accessing main memory to fill up the register file. When stride is set to 1, elements are brought from consecutive memory locations, that is, the first element is brought from main memory address N, and element number VLR is brought from location N+VLR-1. If stride is set to 2, the first element of the vector register file is brought from memory address N, the second from address N+2, and the last from address N+2*VLR-1, and so on. All vector processors support two types of stride: *constant* (unit and non-unit, discussed previously) and *indexed* (also known as

gather-scatter). Indexed stride is the vector equivalent of the RISC *register-indirect* memory access, and allows the CPU to gather elements from sparse memory locations. As a rule, all vector processors support scalar operations (scalar-scalar operations, and scalar-vector operations). Scalar-scalar operations are the ordinary operations performed on scalar RISC/CISC machines, while scalar-vector operations are operations where one of the operands is a vector, while the other is a scalar (think scalar-matrix multiplication). To support scalar-vector operations, vector processors are also equipped with scalar registers (like that found in MIPS). The *read* (output) port of the scalar register file is connected to the input of multiplexers connected to the inputs of each functional unit. These multiplexers pass either the output of the crossbar or the scalar value. They can also allow the output of functional units to be passed as input to other functional units – a concept known as *chaining* (see Optimization).

Optimization: Chaining, Convoys and Out-of-Order Execution
Vector processors can be further optimized using a variety of techniques. Of these, we discuss the three most prevalent ones: chaining, convoys, and out-of-order execution.

Starting with the Cray-1, most vector processors support *chaining*.. With chaining, the vector result of one operation can be used directly as the input to another vector operation – bypassing the write-back stage to the register file bank. This is the vector processor equivalent of *forwarding* found in scalar RISC CPUs. After the startup overhead of the first functional unit (due to the depth of the pipeline), the first output can be fed directly into another functional unit. The control logic checks for dependencies in the program code (e.g. the result of operation N is one of the operands in operation N+1), and when found, sends control signals to the multiplexers at the input of the second functional unit to pass the output of the first functional unit. Chaining increases the efficiency and utilization of the vector processor by overlapping instruction clock cycles, and therefore further reducing CPI[167].

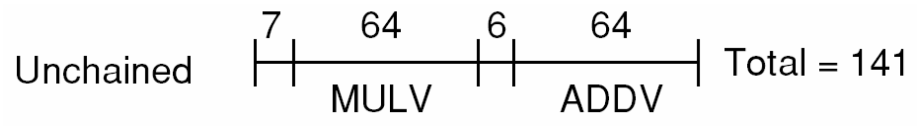


Figure 68 Unchained multiply then add operations. 7 clock cycles are the multiplication overhead, and 6 clock cycles are the addition overhead needed to fill the pipeline

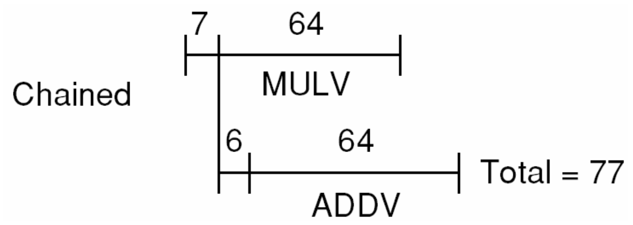


Figure 69 Chained multiply and add operations. After the inital overhead of 7 clock cycles found in multiplication operations, the first result can be immediately fed into the addition FU, bypassing the write back stage

In the above example, chaining the 2 operations (multiply vectors and add vectors) reduced the total number of clocks required from 141 to 77 - nearly a 50% reduction.

As was mentioned previously, vector processors, being superscalar processors, have multiple *independent* functional units. This independency allows the functional units to function in parallel, and the execution paths are referred to as *lanes*. Independent operations can then be dispatched to different lanes and begin execution simultaneously, despite having different instruction orders. When different independent operations begin execution simultaneously on different lanes, they are said to be in the same *convoy*. The instructions in one convoy must not contain any hazards – data or structural. If data hazards do exist, the instructions leave in different convoys, but are then chained together.

Allowing consecutive instructions to be executed in the same convoy can be thought of as a very specific form of out-of-order execution. That is, instructions are not executed in their *instruction order* (the order they had in the code), but rather, are assigned a new order called their *data order*. In the case of convoys, operations with different instruction orders are assigned the same data order (since they are being executed simultaneously), however, some vector processors (and indeed, scalar ones too) allow operations with lower instruction orders to precede those with a higher one. After execution, the instruction results are reordered back to their instruction order, making the entire out-of-order process invisible to the end user. Out-of-order execution improves the performance of CPUs by increasing their utilization. Consider the following example:

```
1      lv       v1, r2   //load VLR elements from location r2 in mem
2      addv.d v2, v1, v3      //v2 = v1+v3
3      addv.d v4, v3, v5      //v4 = v3+v5
```

Assuming that v3 and v5 are ready (vector registers are loaded), and that loading v1 causes a cache read-miss, then using in-order execution will leave the CPU idle while v1 loads (~20 CPU clocks typically). By using out-of-order execution, instruction 3 can be executed before instruction 1 and 2 (while v1 loads), thus making better use of the memory-latency time.

Performance & Benchmarks

One can expect a significant increase in performance of vector processors compared to scalar architectures, especially in areas of computing demanding vector manipulation. This is due to the inherent parallel architecture of vector processors, and their reduction in number of memory accesses, as well as the optimization techniques discussed previously.

1976-1991 was known as the “golden era” of supercomputers. During that time, the clock frequency of supercomputers was between 7-10 times that of contemporary microprocessors [167]. Not only did supercomputers have higher clock rates, but due to their multiple lanes and chaining techniques, were able to perform many more operations per clock.

Table 6

Machine	Year intro	Cycle time	LD/ST paths	Vector Registers	Elements/ Register	Functional Units	Pipes	Flops/ cycle	Words/ cycle
TI-ASC	1972	60.0	LS	–	–	A/M	4	4	4 (32bit)
STAR-100	1973	40.0	L,L,S	–	–	A/D/L, A/M	1	2	3
CRAY-1	1976	12.5	LS	8	64	A,M,R,I,L,S	1	2	1
Fujitsu VP200	1982	7.0	LS,LS	256–8	1024–32	A/L,M,D	2	4	4
Cray X-MP	1983	9.5	L,L,S	8	64	A,M,R,I,L,L,S,P	1	2	2+1
Hitachi S810/20	1983	19.0	L,L,L,LS	32	256	A/L,A/L,M/D+A,M+A	2	12	8 or 2
NEC SX-2	1984	6.0	L,LS	8+8K	256/64–256	A,M/D,L,S	4	16	8 or 4
CRAY-2	1985	4.1	LS	8	64	A,M/R/Q,I,L	1	2	1
Hitachi S820/80	1987	4.0	L,LS	32	512	A/L,M+A,D	4	12	8 or 4
Cray Y-MP	1988	6.3	L,L,S	8	64	A,M/L,R,I,L,S,P	1	2	2+1
Fujitsu VP2600	1989	3.2	LS,LS	2048–64	64–2048	M+A/L,M+A/L,D	4	16	8
NEC SX-3	1990	2.9	L,L,S	8+16K	256/64–256	A/S,A/S,M/L,M/L	4	16	8+4
Cray C90	1992	4.0	L,L,S	8	128	A,M/L,R,I,L,S,P	2	4	4+2
NEC SX-4	1996	8.0	LS,LS	8+16K	256/64–256	A/S,M,D,L	8	16	16

Table 1 shows the functional unit and register file characteristics of several vector supercomputers. Legend: A = FP-Add, D = FP-Divide, I=Integer-Add, L=Logical, M=FP-Multiply, P=Population Count/parity, R=Reciprocal Approximation, Q=Reciprocal Square Root, S=Shift, Independent functional units are separated by commas. Functional units that perform several operations are indicated using a slash mark (A/S, for example). Cascaded units, that is units that hang off other functional units, are indicated using a "+" sign. [167]

The combination of higher clock rates and more operations per clock created vector machines that were 16-70 times faster than contemporary microprocessors [167].

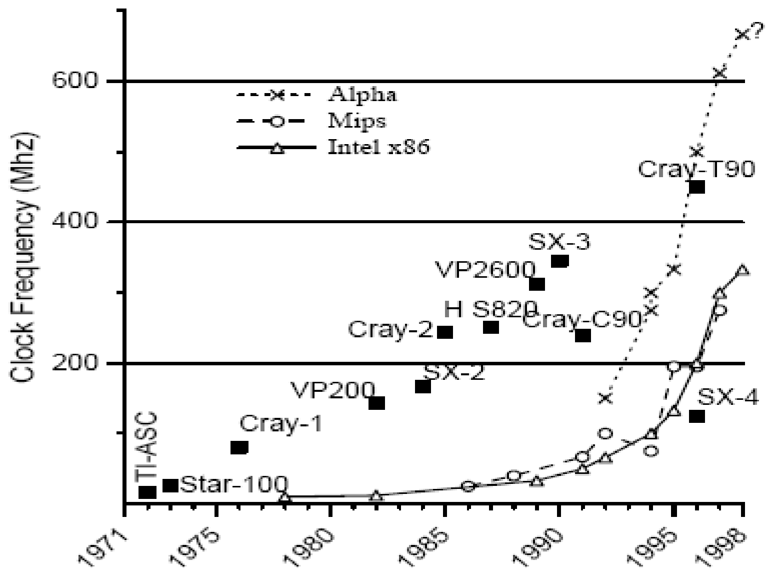


Figure 70 Evolution of clock frequency over the years for vector supercomputers and for several microprocessors. [167]

Besides clock rates and operations per clock, vector processors compared very favorably to microprocessors in terms of memory bandwidth.

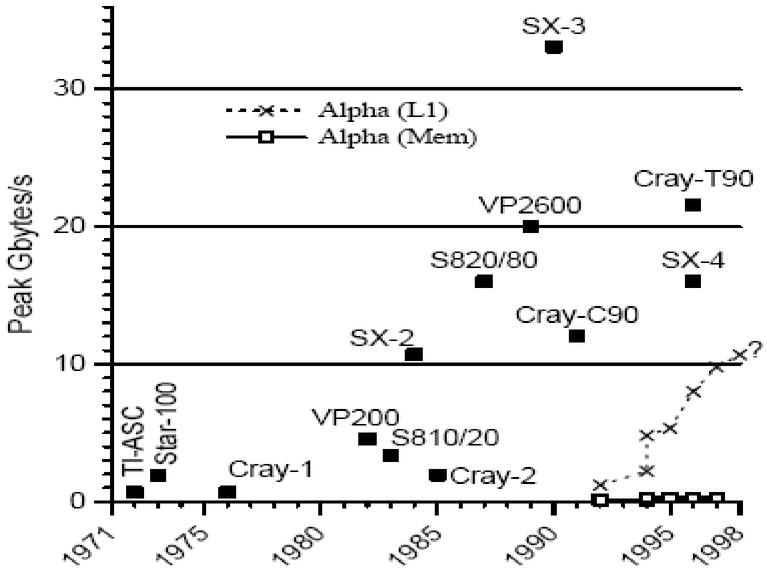


Figure 71 Evolution of peak main memory bandwidth for vector supercomputers and for an Alpha based workstation. The values for the Alpha main memory bandwidths are derived using the 33Mhz 21072 and 21172 chipsets. [167]

Since the early nineties, vector machines have lost their performance lead in the supercomputer market. The leading factor for the decline in popularity of vector machines is cost. This is because the vector processor market is a very specialized market, as opposed the microprocessor one. Design costs are spread over a (relatively) few end users raising the price of the hardware significantly, whereas in the microprocessor market, the (one time) design costs are spread over a much bigger market, making the dominant factor of the end user price the *manufacturing costs*. The second reason is the fact that relatively few innovations have taken place since the Cray-1, introduced in 1976. In fact, all the optimization techniques discussed above and found in most vector machines were present in the Cray-1. Compared to the much more dynamic research taking place in microprocessors, the vector processor research is very stagnant indeed. Finally, the emergence of parallel computing and cluster based computing saw the decline in vector processor popularity even more – making it difficult to envision vector processors being in the performance lead ever again.

Summary

This lecture started with the need for vector processors, and why their idea was conceived and what they're used for. It then moved on to discuss the architecture of vector processors, focusing on the design of the VMIPS vector processor.

It talked about special hardware usually found in vector processors (VLR, stride, vector register banks, etc), what they're used for, and how they improve performance, as well as the reason behind certain design choices (for example, why crossbar networks are used to connect vector register files to functional units).

Next, the lecture discussed the optimization techniques most commonly found in vector processors (namely, chaining and convoys) as well as sample code detailing the programming methods of vector processors, and the key differences between it and orthodox RISC programming (for example, the lack of loop overhead).

The lecture concluded with performance analysis of vector processors compared to RISC processors. Benchmark results of notable vector processors were reviewed and discussed, and finally, the lecture touched upon the main reasons in the decline in popularity of vector processors.

Paper 1 Scalable Vector Processors For Embedded Systems [170]
Paper 2 Vector Architectures: Past, Present and Future [167]

Lecture #21a: VLIW Processors

Course: Computer Architecture III
Lecturer: Dr. Miodrag Bolic
Scribes: Jefferson Bather A. Moraes(3524984) and Luai Alrantisi(3561997)
Date: 22 November, 2006

Introduction

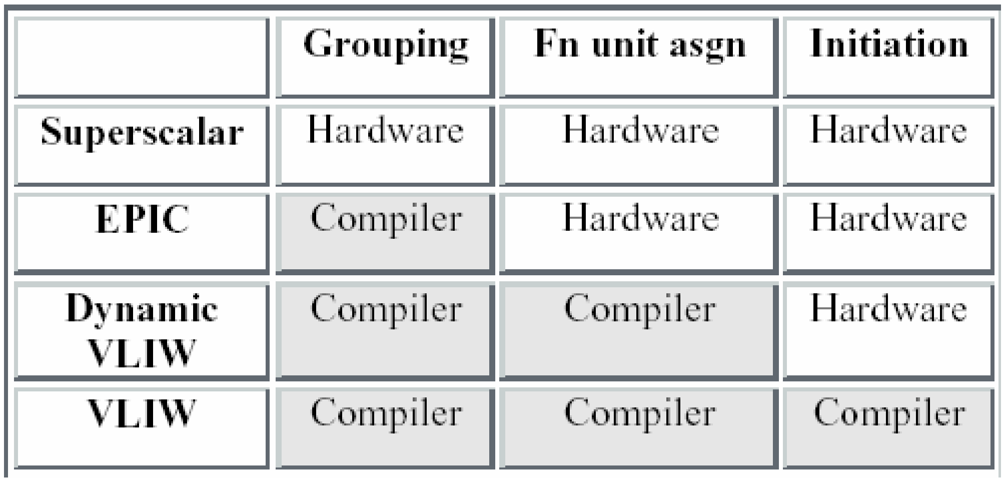
1. This lecture provided basic concepts on implementation and design of VLIW (Very Long Instruction Word) processors. Among the topics covered in the lecture, they were:

- Categories of ILP(Instruction Level Parallelism) Architectures: Superscalar, EPIC (Explicitly Parallel Instruction Computing), Dynamic VLIW and VLIW;
- Brief description of Superscalar architecture;
- Description of VLIW Architecture;
- Datapath details of a TMS processor based on VLIW;
- Unrolling and Trace Scheduling;

Explanation

2. Major Categories of ILP Architectures:
ILP architectures were developed with the purpose of improving computer performance by executing many simultaneous operations in a computer program. These architectures are characterised for having several parallel processing units.

In this context, computers must define the dependencies such that they can be grouped, assigned to a functional unit and initiated accordingly by the compiler or hardware. The following figure shows how grouping, functional unit assignment and scheduling is done in different architectures:



	Grouping	Fn unit asgn	Initiation
Superscalar	Hardware	Hardware	Hardware
EPIC	Compiler	Hardware	Hardware
Dynamic VLIW	Compiler	Compiler	Hardware
VLIW	Compiler	Compiler	Compiler

Table 1: Four Major Categories of ILP Architectures. 0

Table 1 clearly defines how each of the ILP Architectures is organised regarding how the tasks are done.

In Superscalar, all tasks are performed in hardware. It requires the implantation of complex and heavy hardware such that it can handle all steps for processing data (fetching, decoding, issuing, executing and completing the instruction cycle. It is an implementation with broad compatibility. 0

In EPIC, the compiler checks the dependencies and all the other tasks are done in hardware. Compatibility with other implementations is assured but does not suffer from hardware grouping of Superscalar.

In Dynamic VLIW, instructions are initiated by hardware.

In VLIW, opposite to Superscalar, all tasks are done by the compiler.

In addition to table 1, the following figure shows how the tasks as performed by the compiler and the hardware. The horizontal lines represent the information sent by the compiler to the hardware in accordance to the specified task (code generation, grouping, function unit assignment and initiation) as shown in table 1. The dashed lines within the compiler block represent the information that is going to be processed by each implementation in software in accordance to each task. For instance, with respect to the tasks that are performed in a Dynamic VLIW implementation, it should be observed that the code is generated and the compiler is responsible for Instruction grouping and Functional Unit Assignment; whereas, the initiation timing task is performed in hardware.

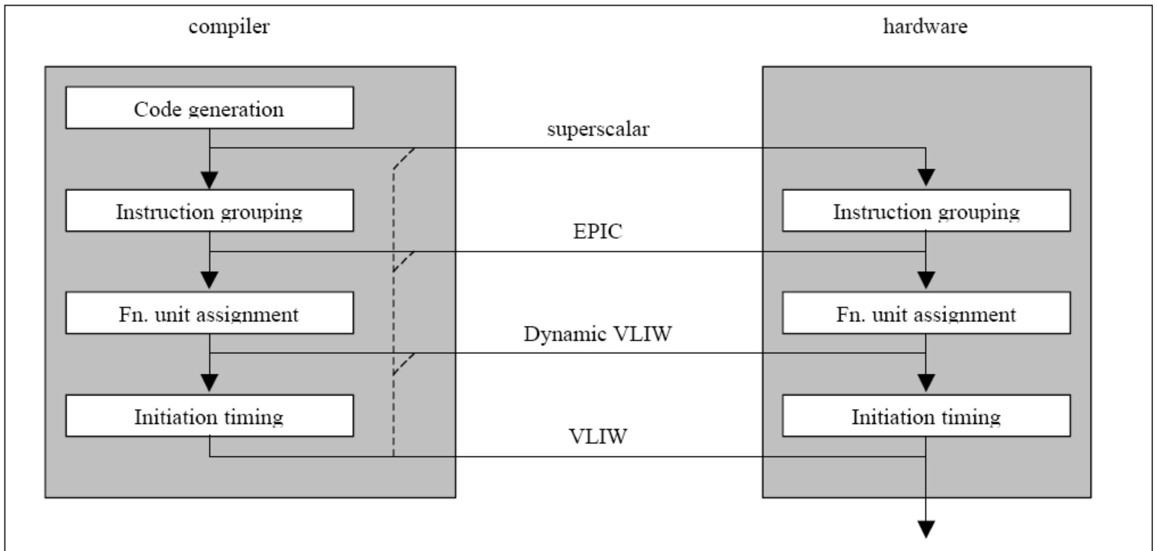


Figure 1. Graphical Depiction of the Three Major Tasks.

Figure 1: Graphical Depiction of the Three Major Tasks 0

3. Superscalar Processors:

3.1 Introduction to Superscalar Processor

A Superscalar processor is one of the implementations of ILP. It is capable of executing multiple instructions simultaneously in a clock cycle using a single pipeline. It fetches multiple instructions at once and dispatches them to the functional units where they are executed concurrently.

In a Superscalar processor, the processor manages more than one instruction at each stage of the execution cycle. The term ILP in the context of Superscalar Processors refers to the degree to which, on average, the instructions of a computer program are executed in parallel[186]. Figure 2 demonstrates a superscalar processor of degree 3 executing in a pipeline implementation. Degree 3 in this context means 3 parallel instructions in one clock cycle.

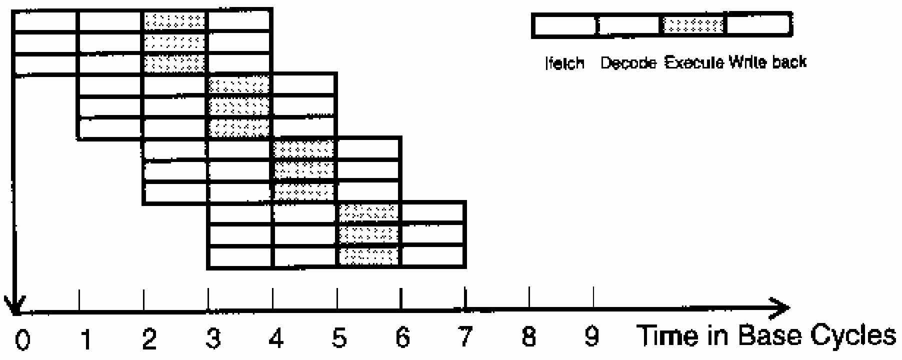


Figure 2: pipelined superscalar implementation of degree 3 [175]

The processor in figure 2 initiates the execution cycle fetching three instructions in the first clock cycle. In the second, it decodes the instructions issued previously and fetches three new instructions. In the third, it executes, decodes and fetches instructions. And this continues until all steps for execution are completed for all instructions (fetching, decoding, executing and write back). In the end, the superscalar processor executed 12 instructions in 7 clock cycles.

Although the figure shows a very efficient pipelining execution, efficiency in superscalar is relative to the type of instructions being executed. Some of them may require the processor to stall some stages for other instructions in the pipeline to be executed properly and this reduces efficiency considerably. One example is the branch instruction which is complicated enough in pipeline implementation since it may cause stalling.

3.2 Execution of Instructions in Superscalar

Instructions in Superscalar are executed as the following:

The static program is the one generated by the compiler.

Next it is fetched along a branch prediction to form a dynamic stream of instructions (this is needed for checking dependencies).

Instructions are dispatched in the window of execution where order of execution is determined by the data dependencies and hardware resources available.
After this stage instructions are reordered accordingly. This step is very important because instruction execution in parallel and pipeline implementation may cause them to be out of order in accordance to the static program. There is also the problem with branch instructions which may ignore some instructions from the static program and this may cause instructions to be completed out of order. Therefore, a temporary storage unit is required for holding the results and used at the end of the instructions execution to put the results in order. [186]

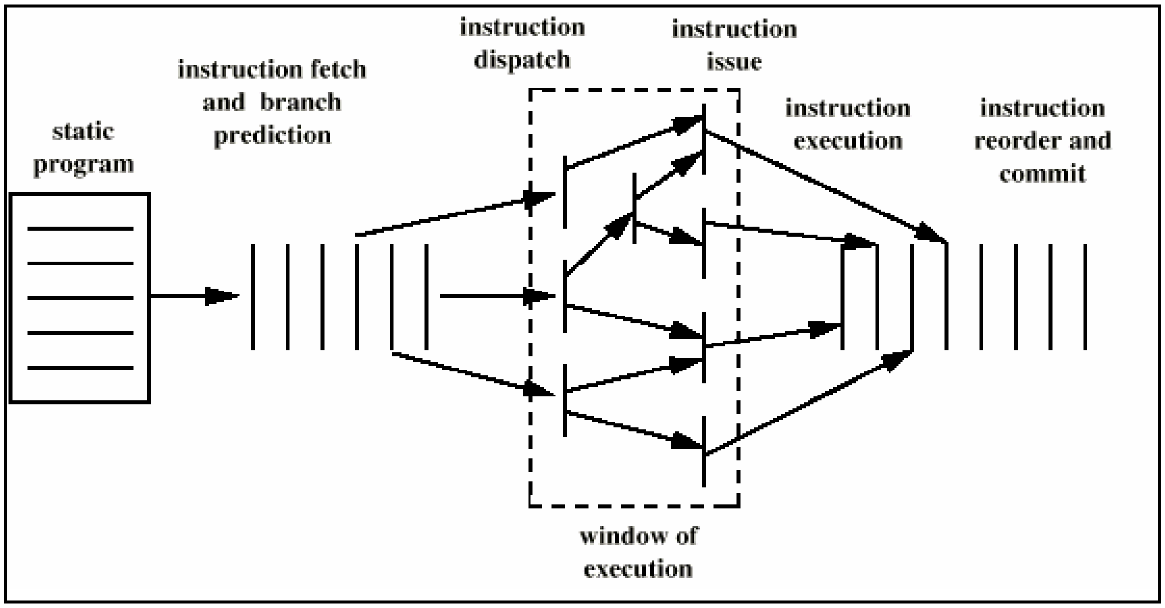


Figure 3: Superscalar Processing Unit [Stallings].

3.3 Advantages and Disadvantages

Superscalar processors make use of complex hardware structure for execution of instructions making it inappropriate to be implemented in embedded systems. Another issue is the number of instructions executed in parallel. It is imperative for the processor to check dependencies at run time and this is costly with respect to logical elements of the processor. This certainly is a problem when the number of parallel instructions to be executed comes in great number. Therefore this sets a limit on the number of parallel instruction Superscalar processors can execute. For this specific case, compiler-based architectures have better performance. Another issue is high power consumption by the processor. Despite that, superscalar is still a good alternative if compatibility to multiple systems is needed.

Examples of Superscalar Processors [175]

PowerPC 604[181] features six independent execution units (Branch, Load/Store, 3 Integer and floating point units), in-order issue and register renaming. PowerPC 620[182] provides in addition to the 604, out-of-order issue instead. Pentium[183] features three independent execution units(2 Integer and floating point units) and in-order issue.

For more information related to these architectures, please use the references.

4. VLIW Processors:

4.1 Introduction to VLIW Processors

VLIW stands for Very-Long Instruction Word. This architecture is another approach to ILP implementation. In contrast to Superscalar, VLIW relies completely on the compiler for all tasks pre-execution. In this case, the compiler determines the how the instructions are grouped, assigned to available functional units and scheduled.

VLIW is characterised for having a very simple instruction set with respect to the number of different instructions, but complex and large with respect to the size of each instruction. This last part applies because it is specified in each instruction the state of each instruction and all functional units of the system. The main purpose is to plan the computer program such that the compiler implements all the necessary tasks for execution, opposed to superscalar processors, which has the hardware as the mean to execute all tasks for execution.

The main triumph of VLIW is that it can provide higher parallelism since its implementation is much simpler and cheaper to build than equivalent Superscalar CISC or RISC architectures. The long instruction word already encodes the concurrent operations and this certainly reduces hardware complexity as opposed to Superscalar [179]. This is very useful when dealing with complicated branch instructions. In, branch is predicted by the compiler itself which profiles information to indicate the direction of the branch. This allows it to move and preschedule operations before the branch is taken.

4.2 Comparison: CISC, RISC and VLIW [177] [179]

In this context, one should consider CISC and RISC some of the possible implementations of Superscalar.

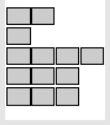
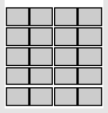
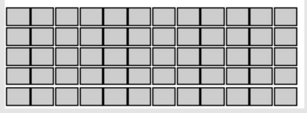
ARCHITECTURE CHARACTERISTIC	CISC	RISC	VLIW
INSTRUCTION SIZE	Varies	One size, usually 32 bits	One size
INSTRUCTION FO RMAT	Field placement varies	Regular, consistent placement of fields	Regular, consistent placement of fields
INSTRUCTION SEMANTICS	Varies from simple to complex; possibly many dependent operations per instruction	Almost always one simple operation	Many simple, independent operations
REGISTERS	Few, sometimes special	Many, general-purpose	Many, general-purpose
MEMORY REFE RENCES	Bundled with operations in many different types of instructions	Not bundled with operations, i.e., load/store architecture	Not bundled with operations, i.e., load/store architecture
HARDWARE DESIGN FOCUS	Exploit microcoded implementations	Exploit implementations with one pipeline and & no microcode	Exploit implementations with multiple pipelines, no microcode & no complex dispatch logic
PICTURE OF FIVE TYPICAL INSTRU CTIONS ▭ = 1 BYTE			

Table 2: CISC, RISC and VLIW features. [177]

CISC (Complex Instruction Set Computer) instructions vary in size depending on type. The more complex the instruction set, the greater the overhead in decoding these instructions is. This may affect performance. This is true for processors that require microcode to decode macroinstructions. Memory reference instructions are bundled with operations. This means that there could have one instruction that does sum access memory directly and perform operations such as addition and subtraction. (E.g. add memory to register).

RISC (Reduced Instruction Set Computer) instructions usually have the fixed size and take the same amount of time to be executed. So decoding for complex instructions takes the same time as a simple one. Memory reference instructions are not bundled with operations. For example, in order use a particular element from memory in any instructions, it is required to load a register with a value from memory and apply operations only using registers.

VLIW instructions are very similar to RISC architecture except for the fact of having larger size and may apply multiple independent operations. In this context, VLIW instructions can be thought of as a combination of several RISC instructions joined together. So, this implies that VLIW instruction length can be increased or decreased with respect to the number of operations to be performed by the processor.

4.3 VLIW Architecture and Execution of Instructions

VLIW processor uses multiple independent functional units. Basically VLIW packets instruction into a single very long instructions and dispatches them. The Figure 4 shows a basic VLIW processor and a pipeline implementation.

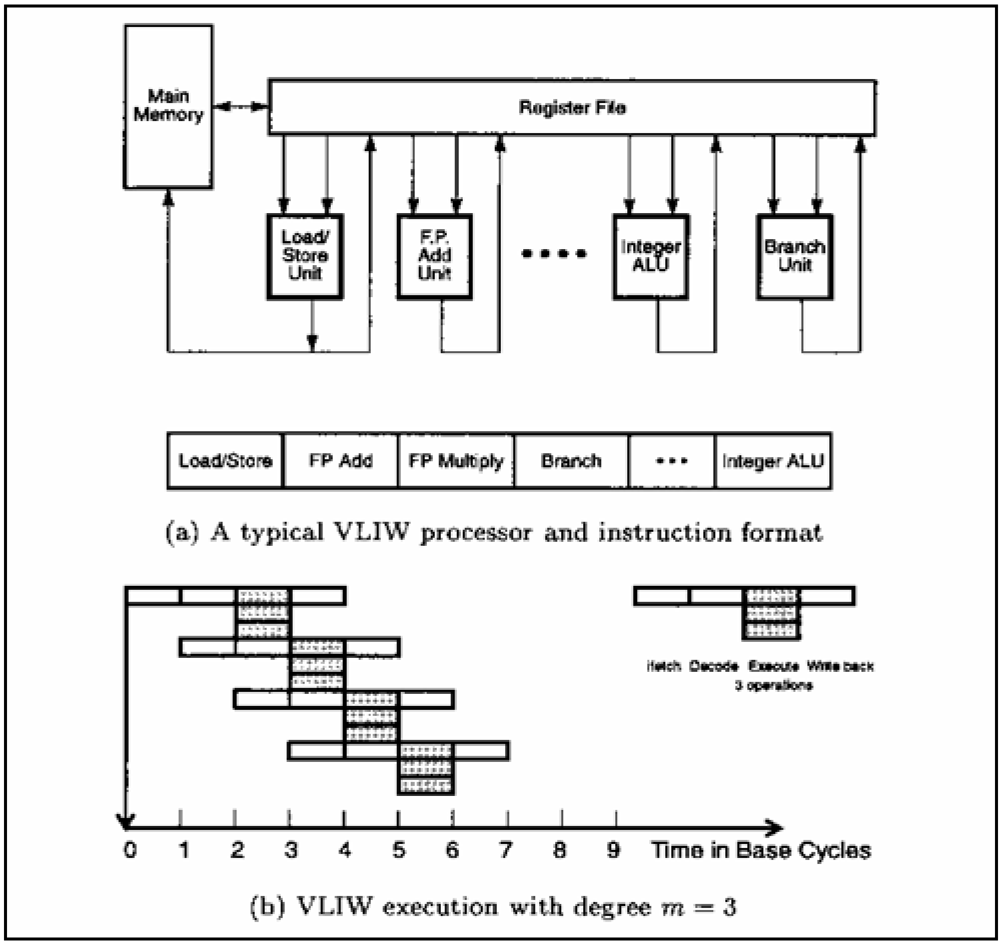

Main Memory
Register File
Load/ Store Unit
F.P. Add Unit
Integer ALU
Branch Unit
Load/Store
FP Add
FP Multiply
Branch
Integer ALU
(a) A typical VLIW processor and instruction format
Fetch Decode Execute Write back
3 operations
0 1 2 3 4 5 6 7 8 9 Time in Base Cycles
(b) VLIW execution with degree $m = 3$

Figure 4: VLIW Processor and execution in a pipeline [175]

Figure 4a shows the organisation of the processor with all its components. In (b), a pipeline implementation of VLIW with degree m = 3 is shown. In this case the processor can execute three parallel instructions. One interesting issue to notice is that instructions are assembled together in the fetching, decoding, and write back stages and executed separately in execute stage.

In an ideal parallel execution in VLIW, the instruction generated should be able to utilise all functional units, in other words, from a single instruction in VLIW, there will be separated instructions to be executed in all functional units. Thus the number of functional units should be the same as the number of instructions for each instruction fetch. See figure 4.

It is up to the compiler to determine the dependencies between instructions, in order to preserve independency of program codes to be executed by the functional units. In this case, the number of instructions to be executed is a function of number of functional units and of their type as well. Each operation carries an independent opcode, which means that in a clock cycle, the functional units should be executing different instructions from one another.

One interesting feature in VLIW processors is the simplified control of functional units as opposed to Superscalar. This is true because the complexity of analysing instructions is transferred from hardware to the compiler.

In VLIW, the program code to be executed is first analysed by the compiler in order to detect any portion that can be executed in parallel. And these are packed into a single instruction. See figure 5.

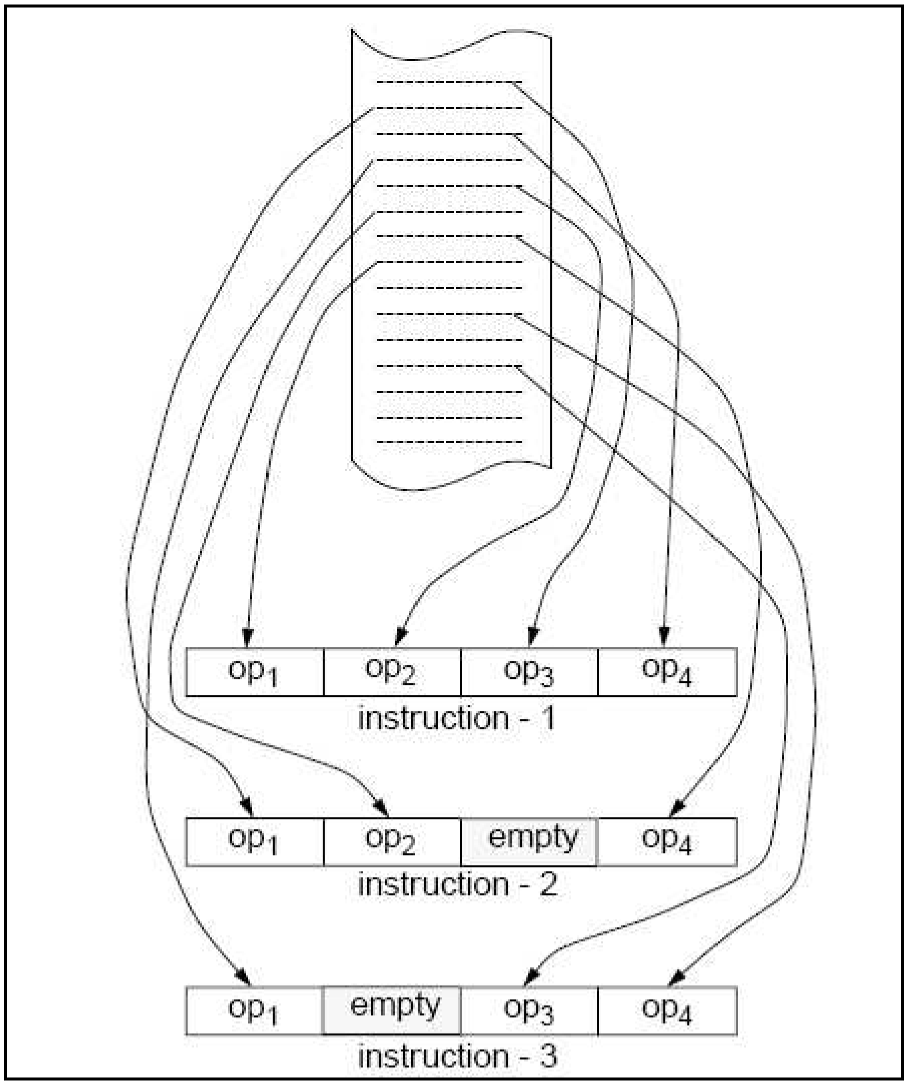
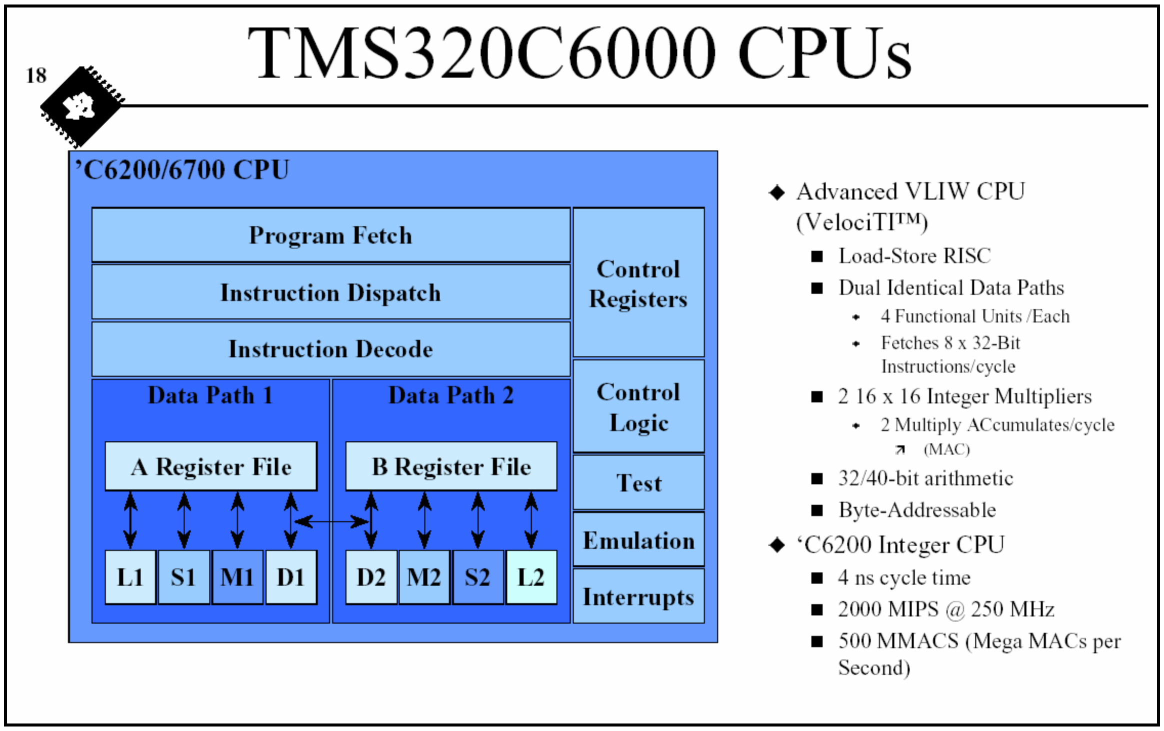


Figure 5: detection of parallelism and packaging of operations. [176]

Example of VLIW Processor: TMS

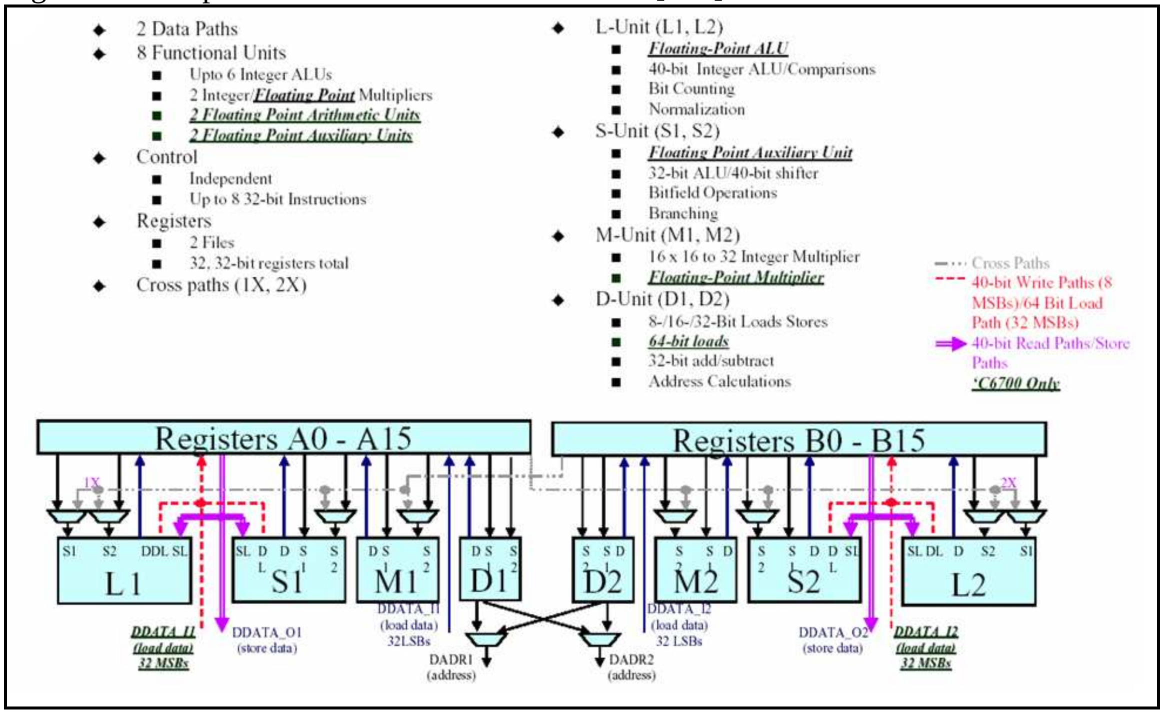


TMS320C6000 CPUs

18

- Advanced VLIW CPU (VelociTI™)
  - Load-Store RISC
  - Dual Identical Data Paths
    - 4 Functional Units /Each
    - Fetches 8 x 32-Bit Instructions/cycle
  - 2 16 x 16 Integer Multipliers
    - 2 Multiply ACcumulates/cycle
      - (MAC)
  - 32/40-bit arithmetic
  - Byte-Addressable
- 'C6200 Integer CPU
  - 4 ns cycle time
  - 2000 MIPS @ 250 MHz
  - 500 MMACS (Mega MACs per Second)

Figure 6: TMS processor from Texas Instruments. [175]



- 2 Data Paths
- 8 Functional Units
  - Upto 6 Integer ALUs
  - 2 Integer/*Floating Point* Multipliers
  - *2 Floating Point Arithmetic Units*
  - *2 Floating Point Auxiliary Units*
- Control
  - Independent
  - Up to 8 32-bit Instructions
- Registers
  - 2 Files
  - 32, 32-bit registers total
- Cross paths (1X, 2X)
- L-Unit (L1, L2)
  - *Floating-Point ALU*
  - 40-bit Integer ALU/Comparisons
  - Bit Counting
  - Normalization
- S-Unit (S1, S2)
  - *Floating Point Auxiliary Unit*
  - 32-bit ALU/40-bit shifter
  - Bitfield Operations
  - Branching
- M-Unit (M1, M2)
  - 16 x 16 to 32 Integer Multiplier
  - *Floating-Point Multiplier*
- D-Unit (D1, D2)
  - 8-/16-/32-Bit Loads Stores
  - *64-bit loads*
  - 32-bit add/subtract
  - Address Calculations

Cross Paths
40-bit Write Paths (8 MSBs)/64 Bit Load Path (32 MSBs)
40-bit Read Paths/Store Paths
'C6700 Only
Registers A0 - A15
Registers B0 - B15
1X
2X
L1
S1
M1
D1
D2
M2
S2
L2
DDATA_I1 (load data) 32 MSBs
DDATA_O1 (store data)
DDATA_I1 (load data) 32LSBs
DADR1 (address)
DADR2 (address)
DDATA_I2 (load data) 32 LSBs
DDATA_O2 (store data)
DDATA_I2 (load data) 32 MSBs

Figure 7: Datapath Detail of TMS [175]

Figures 6 and 7 show one implementation of VLIW processor designed by Texas Instruments. The datapath of the model C62x presents two sets of Register Files (A and B) which can be used as pointers or, registers of condition or simply for storage of data. Each of these sets contains 32 bit registers (from A0-A15, B0-B15). Each register A or B possesses a number of functional units available for operations. One interesting feature is that A can be used for operations performed in B’s functional units as shown by the grey dashed lines in figure 7.

4.5 Advantages and Disadvantages of VLIW

The main advantage of VLIW is definitely practicability in embedded systems. Since is it entirely based in software, it does not require complicated or heavy hardware to execute instructions. It is a processor capable of performing more instructions in parallel than Superscalar; this implies that more function units can be added without any complications to performance or power consumption [176]. Compilers can detect parallelism by analysing the code prior to its execution and thus ensuring that there is no dependency check during execution

However, there are serious problems with number of instructions to be executed. In VLIW, it is necessary to keep all functional units busy in order to have a better improvement over RISC or CISC architectures. Otherwise some portions of the slots will be wasted and a NOP (no operation) will be applied. Another problem is incompatibility with different architectures.

One issue to consider is that the compiler applied in VLIW assumes all operations to be predicted. Therefore most VLIW implementations do not make use of cache memories which would make some operations if not all unpredictable. If there is unpredictable code then stalling will be required decreasing performance.

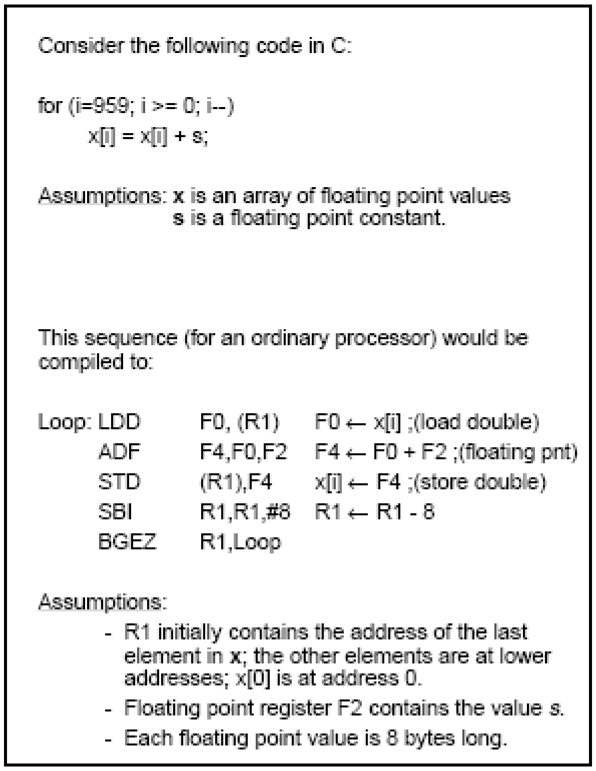
5. Loop Unrolling and Truce Scheduling

5.1 Example of a C Code Execution in VLIW

In this example, a C code with a for loop, shown in figure 8, is executed by a VLIW processor with instruction set and the number of clock cycles for each instruction, shown in table 3, using Loop Unrolling technique.

VLIW Processor of degree n = 5	
Instruction Type	**Number of clock cycles assigned**
Load	2 cycles
Floating Point	3 cycles
Branch/Integer	1 cycle

Table 3: Instructions and number of clock cycles of a VLIW Processor.



Consider the following code in C:

```
for (i=959; i >= 0; i--)
        x[i] = x[i] + s;
```

Assumptions: **x** is an array of floating point values
s is a floating point constant.

This sequence (for an ordinary processor) would be compiled to:

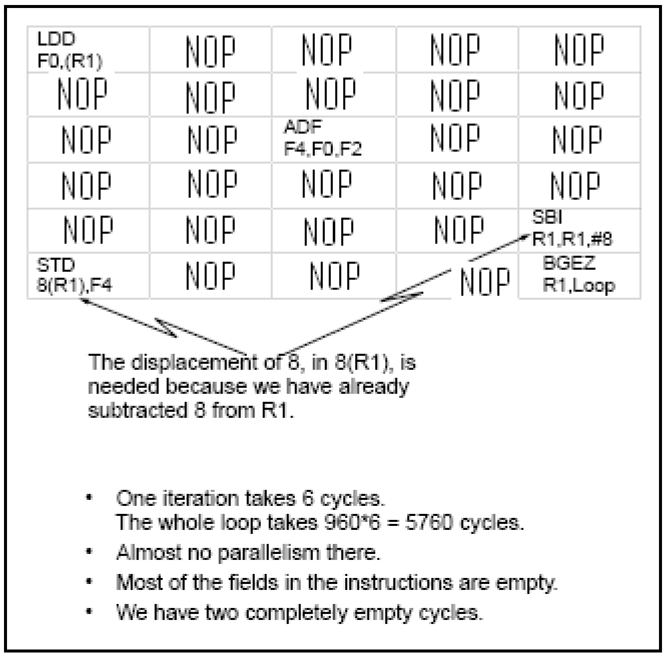
```
Loop: LDD    F0, (R1)    F0 ← x[i] ;(load double)
      ADF    F4,F0,F2    F4 ← F0 + F2 ;(floating pnt)
      STD    (R1),F4     x[i] ← F4 ;(store double)
      SBI    R1,R1,#8    R1 ← R1 - 8
      BGEZ   R1,Loop
```

Assumptions:

- R1 initially contains the address of the last element in **x**; the other elements are at lower addresses; x[0] is at address 0.
- Floating point register F2 contains the value *s*.
- Each floating point value is 8 bytes long.

Figure 8: Example 1[176]

So, the following results are obtained when executing the code:



LDD F0,(R1)	NOP	NOP	NOP	NOP
NOP	NOP	NOP	NOP	NOP
NOP	NOP	ADF F4,F0,F2	NOP	NOP
NOP	NOP	NOP	NOP	NOP
NOP	NOP	NOP	NOP	SBI R1,R1,#8
STD 8(R1),F4	NOP	NOP	NOP	BGEZ R1,Loop

The displacement of 8, in 8(R1), is needed because we have already subtracted 8 from R1.

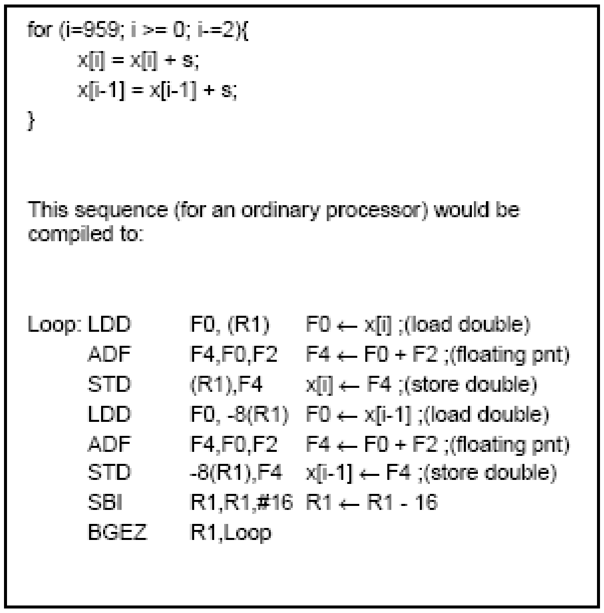
- One iteration takes 6 cycles.
  The whole loop takes 960*6 = 5760 cycles.
- Almost no parallelism there.
- Most of the fields in the instructions are empty.
- We have two completely empty cycles.

Figure 9: Results of execution of C code example. [176]

In this example, it is clearly seen that the number of instructions to be executed and the type of instructions how parallel this processor can be. Unfortunately, Load, Branch Store and Floating Point operations cannot be executed in parallel because they share the same data and thus it is required for them to wait for the other to complete. This really makes this processor implementation no less efficient than a RISC processor since there is almost no parallelism. It is also possible to see that many slots were wasted with NOP's. This shows one of the flaws encountered in VLIW processor. However this can be improved increasing the number of instructions to be executed or by applying techniques such as Loop Unrolling or Truce Scheduling.

5.2 Loop Unrolling

Loop Unrolling is a very simple technique which increases the number of instructions executed in a loop and decreases the number of iterations. This is done by calling the instructions that are used in multiple iterations of the loop and combine them in a single iteration. By using the example in figure 8 with a few modifications in the code, the implementation has a better performance.



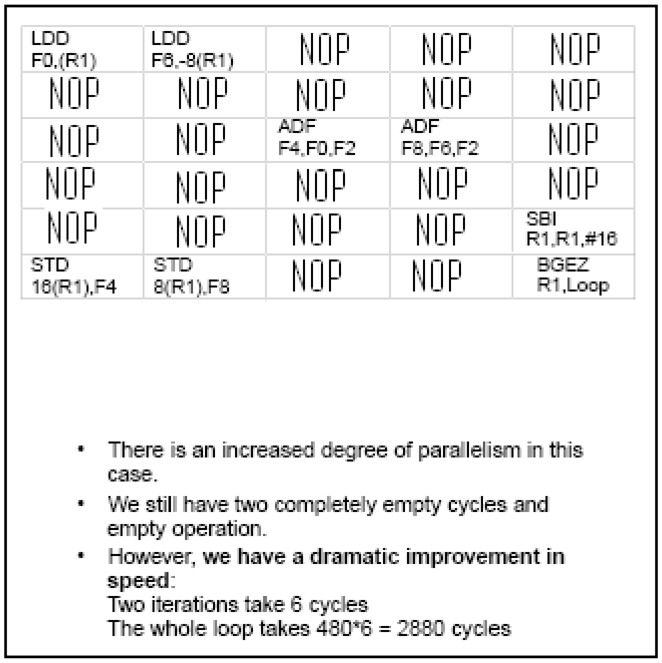
```
for (i=959; i >= 0; i-=2){
        x[i] = x[i] + s;
        x[i-1] = x[i-1] + s;
}
```

This sequence (for an ordinary processor) would be compiled to:

```
Loop: LDD      F0, (R1)     F0 ← x[i] ;(load double)
      ADF      F4,F0,F2     F4 ← F0 + F2 ;(floating pnt)
      STD      (R1),F4      x[i] ← F4 ;(store double)
      LDD      F0, -8(R1)   F0 ← x[i-1] ;(load double)
      ADF      F4,F0,F2     F4 ← F0 + F2 ;(floating pnt)
      STD      -8(R1),F4    x[i-1] ← F4 ;(store double)
      SBI      R1,R1,#16    R1 ← R1 - 16
      BGEZ     R1,Loop
```

Figure 10: Loop Unrolling of example in figure 8. [176]

Considering the rewritten code in figure 10, the “for” loop part of the code contains an additional instruction which deals with an index lower. Another difference is that the counter i inside the declaration of the loop increased by 2 at each iteration. From this point, it is possible to see that this loop will be executed twice as faster than the example in Figure 8. Another feature noticed from the from the assembly code is that the number of instructions has increased and this favours parallelism. The results are shown next.



LDD F0,(R1)	LDD F6,-8(R1)	NOP	NOP	NOP
NOP	NOP	NOP	NOP	NOP
NOP	NOP	ADF F4,F0,F2	ADF F8,F6,F2	NOP
NOP	NOP	NOP	NOP	NOP
NOP	NOP	NOP	NOP	SBI R1,R1,#16
STD 16(R1),F4	STD 8(R1),F8	NOP	NOP	BGEZ R1,Loop

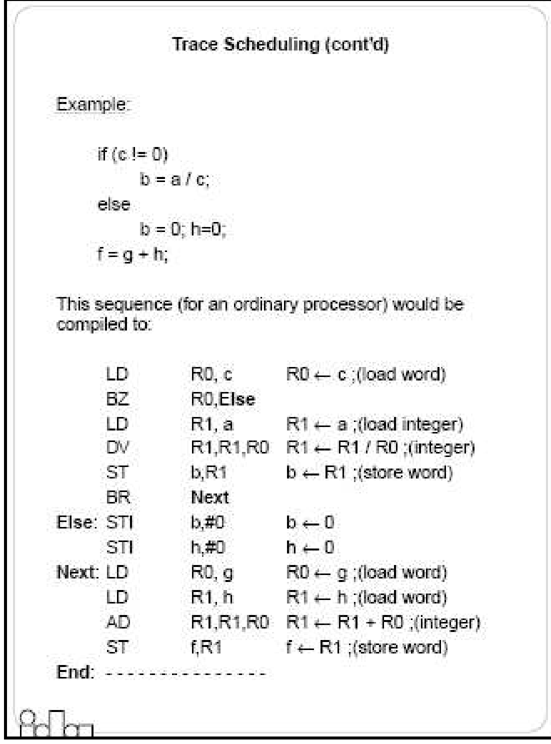
- There is an increased degree of parallelism in this case.
- We still have two completely empty cycles and empty operation.
- However, we have a dramatic improvement in speed:
  Two iterations take 6 cycles
  The whole loop takes 480*6 = 2880 cycles

Figure 11: Results from figure 10. [176]

At the end the number of NOP's has reduced, more instructions are executed in parallel, and the execution time dropped by 50% which is a great improvement from the example in figure 8. The performance can be increased further by applying the same technique with more instructions executed per cycle and reducing the number of iterations accordingly as shown in some examples in [176]. It is up to the compiler or the programmer to define optimal level of unrolling for best performance.

Trace Scheduling

Trace scheduling is another technique used for parallel execution but it is used in conditional branches instead [176]. This separates parts of the codes that are unlikely to be executed by predicting the outcome during compilation time. Prediction does not always work and may even reduce performance if the guess was incorrect because the processor will need to stall parts of the execution and recover the code to be executed. Trace scheduling makes use of three steps: trace selection, instructions scheduling, and replacement and compensation. The following figure shows the example to be implemented.



Trace Scheduling (cont'd)

Example:

```
if (c != 0)
      b = a / c;
else
      b = 0; h=0;
f = g + h;
```

This sequence (for an ordinary processor) would be compiled to:

```
       LD    R0, c       R0 ← c ;(load word)
       BZ    R0,Else
       LD    R1, a       R1 ← a ;(load integer)
       DV    R1,R1,R0    R1 ← R1 / R0 ;(integer)
       ST    b,R1        b ← R1 ;(store word)
       BR    Next
Else:  STI   b,#0        b ← 0
       STI   h,#0        h ← 0
Next:  LD    R0, g       R0 ← g ;(load word)
       LD    R1, h       R1 ← h ;(load word)
       AD    R1,R1,R0    R1 ← R1 + R0 ;(integer)
       ST    f,R1        f ← R1 ;(store word)
End:   - - - - - - - - - - - - - - -
```

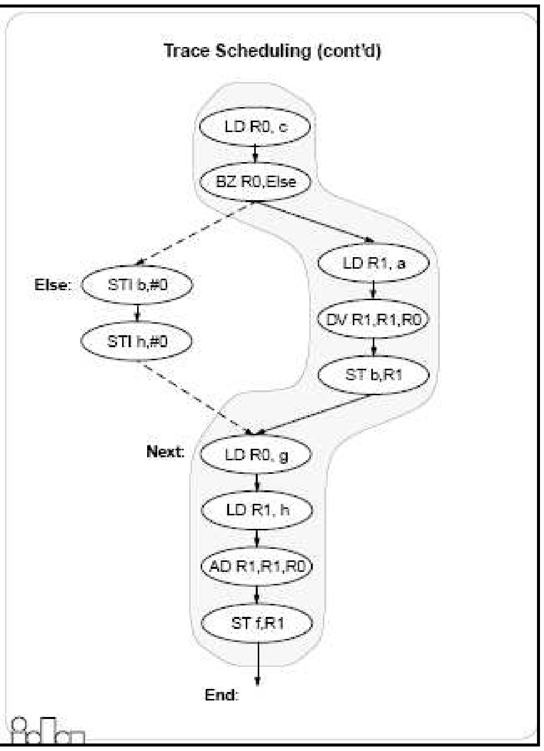
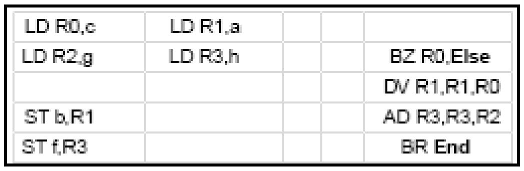


Figure 12: Trace scheduling example and code. [176]

Trace selection is based on choosing the code sequence that is likely to be executed. In this example, the piece of code chosen was b = a/c, since the value c = 0 would certainly cause an exception during execution time.

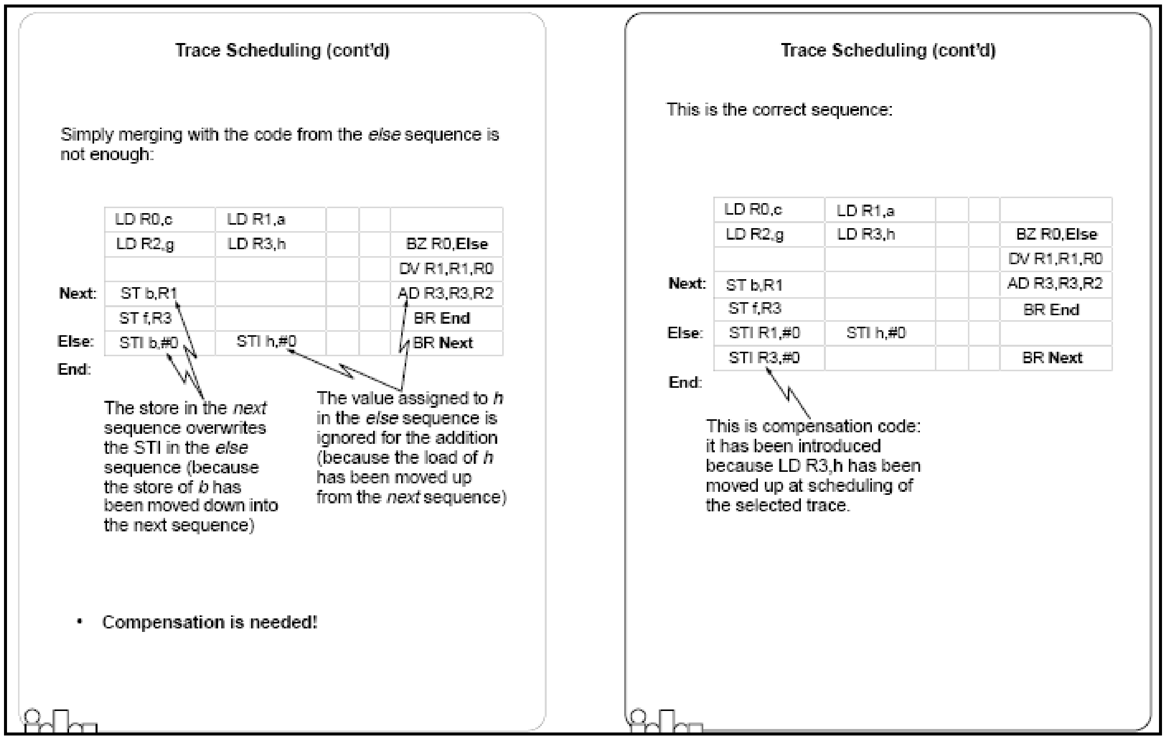
Instruction scheduling, as the name implies, schedules instructions such that they can be executed in parallel.



LD R0,c	LD R1,a			
LD R2,g	LD R3,h			BZ R0,**Else**
				DV R1,R1,R0
ST b,R1				AD R3,R3,R2
ST f,R3				BR **End**

Figure 13: Scheduling of instructions.

Replacement and compensation refers to the stage of execution whenever the **less likely path** is taken to optimise execution. If for some reason the less likely path taken, it is necessary to use the compensation code in order to keep the program data correct. The less likely path scenario adds some overhead to the execution. See figure 13.



Trace Scheduling (cont'd)

Simply merging with the code from the *else* sequence is not enough:

	LD R0,c	LD R1,a			
	LD R2,g	LD R3,h			BZ R0,**Else**
					DV R1,R1,R0
Next:	ST b,R1				AD R3,R3,R2
	ST f,R3				BR **End**
Else:	STI b,#0	STI h,#0			BR **Next**
End:					

The store in the *next* sequence overwrites the STI in the *else* sequence (because the store of *b* has been moved down into the next sequence)

The value assigned to *h* in the *else* sequence is ignored for the addition (because the load of *h* has been moved up from the *next* sequence)

- **Compensation is needed!**

Trace Scheduling (cont'd)

This is the correct sequence:

	LD R0,c	LD R1,a			
	LD R2,g	LD R3,h			BZ R0,Else
					DV R1,R1,R0
Next:	ST b,R1				AD R3,R3,R2
	ST f,R3				BR End
Else:	STI R1,#0	STI h,#0			
	STI R3,#0				BR Next
End:					

This is compensation code: it has been introduced because LD R3,h has been moved up at scheduling of the selected trace.

Figure 14: Compensation code. [176]

Summary:

VLIW processors are one of the most promising implementations of ILP due to its better performance when executing many instructions in parallel. The main problem in VLIW implementation is its inefficiency performance when there are few instructions to be executed in parallel, leading to a performance equivalent of CISC and RISC in this case. Another problem is compatibility with other architectures. VLIW is mainly applied in embedded systems.

Loop unrolling and trace scheduling are techniques that improves degree of parallelism of VLIW processors when executing instructions.

Paper 1: Very Long Instruction Word Architectures and the ELI - 512 [187]
Paper 2: “Dynamically Scheduled VLIW Processors” [187]

Lecture #21b: Very Large Instruction Word (VLIW) Processors

Course: Computer Architecture III
Lecturer: Miodrag Bolic
Scribe: Mustafa Nur and Ali Al-Munayer
Date: October 15, 2007

Introduction

1 This lecture introduced several architectures that are able to execute more than 1 instruction per clock cycle, mainly Superscalar and VLIW architectures. The pros and cons of both Superscalar and VLIW architectures are also covered in this lecture. Two other architectures that try to minimize the shortcomings of Superscalar and VLIW architectures are also introduced (EPIC and Dynamic VLIW). A three way comparison is made between scalar CISC and RISC architectures and VLIW architectures.

Explanation

2 Parallel Processing

Instruction level parallelism is the initiation and execution within a single processor of multiple instructions in parallel [190]. As stated on the introduction, VLIW and Superscalar processors are designed to exploit instruction level parallelism. The code snippet (Figure 1) below has instructions that can be executed in parallel. Notice that line 1 and 2 are independent to each other. They don’t share any variable, so they can be run in parallel. But line 3 depends on the results from line 1 and 2 before it can be executed, so it can’t be run in parallel with line 1 and 2. Both VLIW and Superscalar processor would exploit this situation by executing lines 1 and 2 simultaneously.

```
1 → c = a + b
2 → e = f - g
3 → h = c + e
```

Figure 1: Code Snippet with Independent and dependent Instructions

Processing instructions in parallel requires three major tasks[106]:
Checking dependencies between instructions to determine which instructions can be grouped together for parallel execution;
Assigning instructions to the functional units on the hardware;

Determining when instructions are initiated placed together into a single word.

2.1 Functional Unit

Functional units, aka execution units, are the modules that operate on the operands that are passed with an instruction. Some examples of functional units are integer or floating-point ALUs, load/store units and branch units. A processor would have several functional units of different types. The number of instructions a process can execute in parallel would depend on the number of functional units. If a processor has 2 integer ALUs, it would be able to perform 2 integer arithmetic instructions in parallel.

2.2 Data Dependency

To ensure that errors are not introduced from executing instructions in parallel, dependencies between instructions have to be checked. The most common type of dependency is data dependency (true dependency). A data dependency occurs when one instruction needs to read the output register of a previously executed instruction [191]. Line 3 from Figure 1 has a data dependency.

Another dependency that needs to be checked is a naming dependency. A naming dependency occurs when a variable is written to after the variable is used by another instruction [191]. Figure 2 show 2 instructions that have a naming dependency. These two lines cannot be executed in parallel; r6 is read in line 1 and written to line 2. Figure 3 has the same code without the naming dependency. In the example below, the naming dependency was fixed by introducing a new register (r3) and copying the value of r6 into it.

```
1 → AND r4, r5, r6
2 → NOT r6, r7
```

Figure 2: Code Showing Naming Dependency [191]

```
1 → LD r3, r6
2 → AND r4, r5, r3
3 → NOT r6, r7
```

Figure 3: Code Without Naming Dependency

The third type of dependency is output dependency. Output dependence occurs when a variable is written to multiple times. The order in which the variable is written to would result in different values. Instructions 1 and 3 in figure 4 are output dependent. If instruction 3 is executed before instruction 1 then instruction 2 would have the wrong r3 value. Just like the naming dependency a change in variable name or register can get rid of the output dependency [191]. Figure 5 shows that same code as Figure 4 but without the output dependency.

```
1 → LD r3, r6
2 → AND r4, r5, r3
3 → LD r3, r2
```

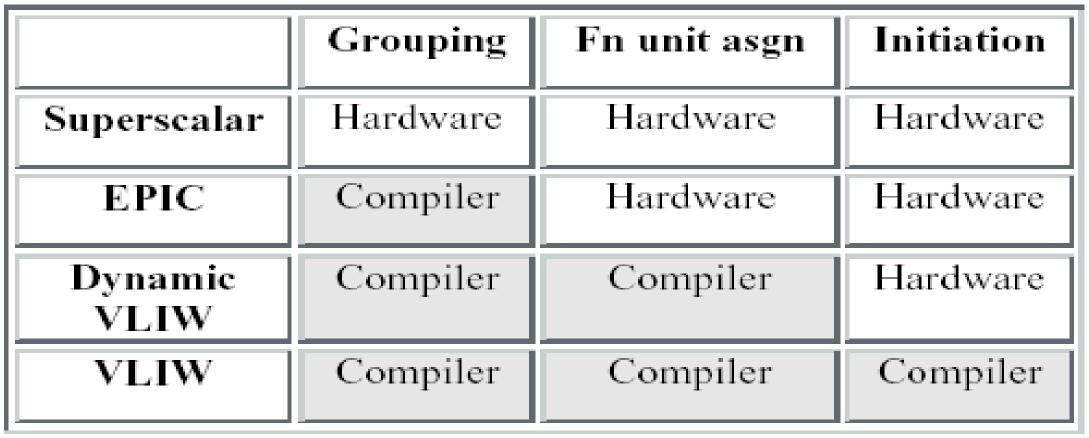
Figure 4: Code with Output Dependency

```
1 → LD r7, r6
2 → AND r4, r5, r7
3 → LD r3, r2
```

Figure 5: Code with Output Dependency

3 Major Categories of ILP Architectures

Processors can use a combination of software compiler and hardware to achieve parallel instruction processing. Table 1 lists several types of processors and the method they use to group independent instructions, assign instructions to a functional unit and initiate instruction execution [190].



	Grouping	Fn unit asgn	Initiation
Superscalar	Hardware	Hardware	Hardware
EPIC	Compiler	Hardware	Hardware
Dynamic VLIW	Compiler	Compiler	Hardware
VLIW	Compiler	Compiler	Compiler

Table 1: Comparison Table of Several Processor Architectures [190]

The grouping stage is where dependency checks are done on the instructions. Instructions that are independent to each other are then grouped together. Each instruction in the group is then assigned to a functional unit to be executed. When the instruction is going to be executed, the start time, is determine in the initiation stage [190]. In VLIW all instruction that are grouped together are executed in parallel [190].

From Figure 4 we can see that Superscalar and VLIW are at opposite extremes. VLIW uses a compiler to group instructions, assign instructions to functional unit and initiate execution of instructions. Superscalar architectures use hardware to satisfy the same requirements as VLIW.

For Superscalar architectures, the compiler produces instructions that are to be executed in sequential order. It is the responsibility of the processor to execute the instructions in parallel. Grouping, functional unit assignment and initiation have to be done on the fly. So, Superscalar architectures are fully dynamic. Dynamic systems are always more complex than their static counterparts. For example, the dependency check hardware for a Superscalar architecture does not scale well [190]. The complexity is $O(n^2)$ with respect to the number instructions the processor can load at once. The main advantage of Superscalar architectures is compatibility. The same code can be run on two Superscalar processors that share the same instructions set but have different types and quantities of functional units. This cannot be said about VLIW architectures.

VLIW architecture is considered static because the grouping, functional unit assignment and initiation are done at compile time. All the VLIW processor has to do is execute the grouped instructions in parallel. One big advantage of VLIW architectures is that they are less complex than their Superscalar counterparts. A major issue with VLIW architectures is that they are not compatible with each other. Code compiled for one implementation with a certain set of functional units with certain latencies will not execute correctly on a different implementation with a different set of function units and/or different latencies [190].

Another flavor of VLIW is Dynamic VLIW. Dynamic VLIW uses hardware for the initiation stage instead of the compiler. Dynamic VLIW has the ability to deal with events that appear at run time [190]. The compiler cannot predict certain events that might occur, for example a cache miss [190]. VLIW architectures usually do not include a cache because a cache miss would invalidate the compilers assumption of latency for long instruction words [190].

The fourth ILP architecture is EPIC (Explicitly Parallel Instruction Computing). The EPIC architecture is similar to the Superscalar architecture, but EPIC uses a compiler for the grouping stage. Like Superscalar architectures, different implementations of EPIC architectures are compatible [190]. EPIC architectures do not require the complex dependency checking hardware because the compiler handles the grouping of the instructions.

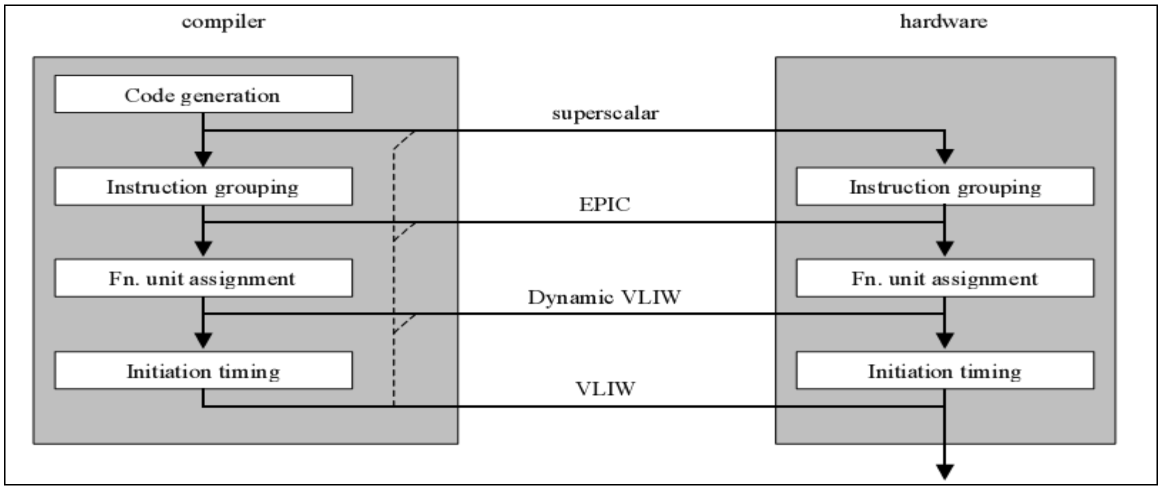


Figure 6: Graphical Depiction of the Three Major Tasks [190]

Figure 6 depicts graphically the same information as Table 1. The dash lines in Figure 6 indicate that for best performance the compiler should complete the three tasks for a specific implementation and then pass that information to the hardware. Even though the Superscalar processor will have to rediscover the groups and so forth, it will still experience performance gains. For example, the HP PA-8000 ran the SPECint95 benchmarks 38% faster and the SPECfp95 benchmarks 53% faster when the code was compiled specifically for the HP PA-8000 [190].

4 Superscalar Processors

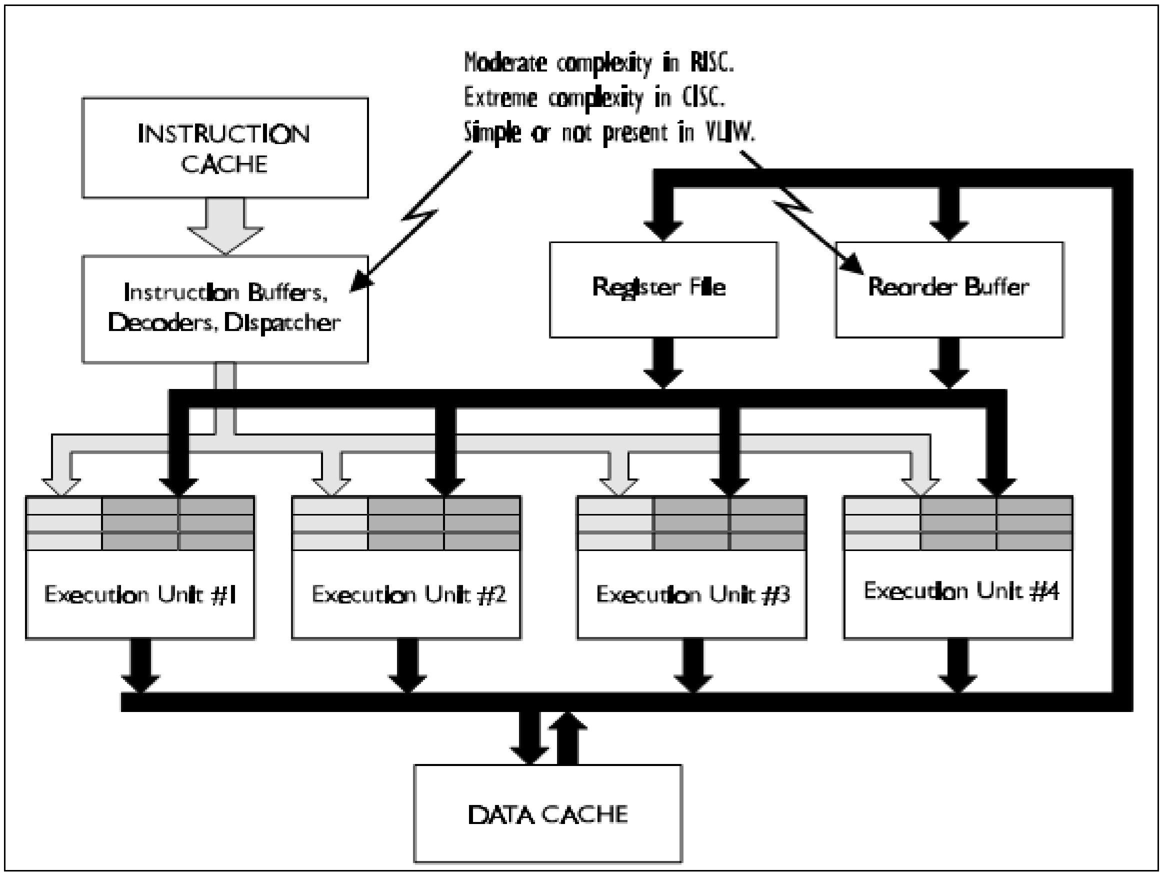
A Superscalar processor can employ a CISC or RISC instruction set. Superscalar processors are designed to exploit instruction-level parallelism in user programs [106]. A scalar CISC or RISC processor can only execute one CISC or RISC instruction at a time while a Superscalar CISC or RISC processor can execute several CISC or RISC instructions at a time [192]. Only independent instructions can be executed in parallel without causing a wait state [106]. The amount of instruction-level parallelism varies widely depending on the type of code being executed [106].

4.1 Architecture

The Superscalar implementation illustrated in Figure 7 has 4 execution units. At most 4 instructions can be executing at a time, depending on the instructions that need to be executed. Each execution unit has a buffer to store instructions that have not been executed [192]. The instructions in the buffer might be waiting for operands to be fetched from the register file [192]. The functional units are almost always pipelined to increase throughput.

The instruction buffer is filled with instructions from the instruction cache. The dispatcher has a window of execution. The window of execution is the number of instructions the dispatcher will compare for dependencies. Independent instructions are grouped together and dispatched to the proper execution unit. The size of the window of execution depends on the number of execution units [192]. As the window of execution gets larger the complexity of the dispatcher grows with it.

CISC and RISC compilers produce code which minimize code size and run time [4]. To minimize code size and runtime the compiler introduces a lot of conditional branches, one in every six instructions . A superscalar processor would have to wait until the branch is resolved, unless it uses branch prediction. The dispatcher takes a guess at the outcome of the branch and starts looking for parallelism . The act of dispatching and executing instructions from the predicted branch path is called speculative execution. Since branch predictions can be wrong there has to be a way to reverse the changes speculative executions might have made. The reorder buffer is used to undo speculative executions when the branch prediction is incorrect . Changes made by speculative execution are stored in the reorder buffer. If a branch is mispredicted the reorder buffer removes the changes made by the speculative execution. If the branch was predicted correctly the reorder buffer writes the changes to the register file. [192]


Moderate complexity in RISC.
Extreme complexity in CISC.
Simple or not present in VLIW.
INSTRUCTION CACHE
Instruction Buffers, Decoders, Dispatcher
Register File
Reorder Buffer
Execution Unit #1
Execution Unit #2
Execution Unit #3
Execution Unit #4
DATA CACHE
Figure 7: Block Diagram of RISC/CISC Superscalar Architecture Implementation [192]

The Pros and Cons of Superscalar Architectures

Pros for Superscalar architecture [193]:
The hardware solves everything, Fully dynamic
Hardware detects potential parallelism between instructions (Grouping).
Hardware tries to issue as many instructions as possible in parallel.
Hardware solves register renaming (naming dependency).
Binary compatibility
If functional units are added in a new version of the architecture or some other improvements have been made to the architecture (without changing the instruction sets), old programs can benefit from the additional potential of parallelism. Why?
Because the new hardware will issue the old instruction sequence in a more efficient way.
More functional units which leads to higher degrees of parallelism.

Cons against Superscalar architectures [193]:
Very complex
More hardware is needed for run-time detection. There is a limit in how far we can go with this technique.

Scaling the dependency check hardware increases the complexity of the hardware by O(n^2) [190].
The window of execution is limited
This limits the capacity to detect potentially parallel instructions

Superscalar Execution

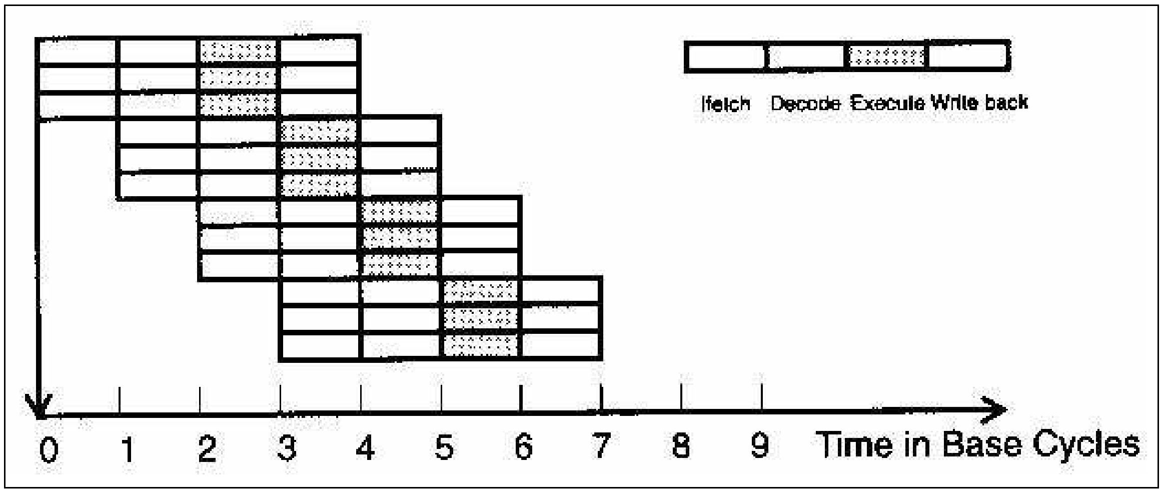


Figure 8: A Superscalar Processor of Degree m=3 [106]

In Figure 8, the degree value m is the number of functional units the processor has, m=3. So at most the processor can execute 3 executions simultaneously. Since each functional unit is pipelined, ideally each functional unit can start a new instruction every clock cycle.

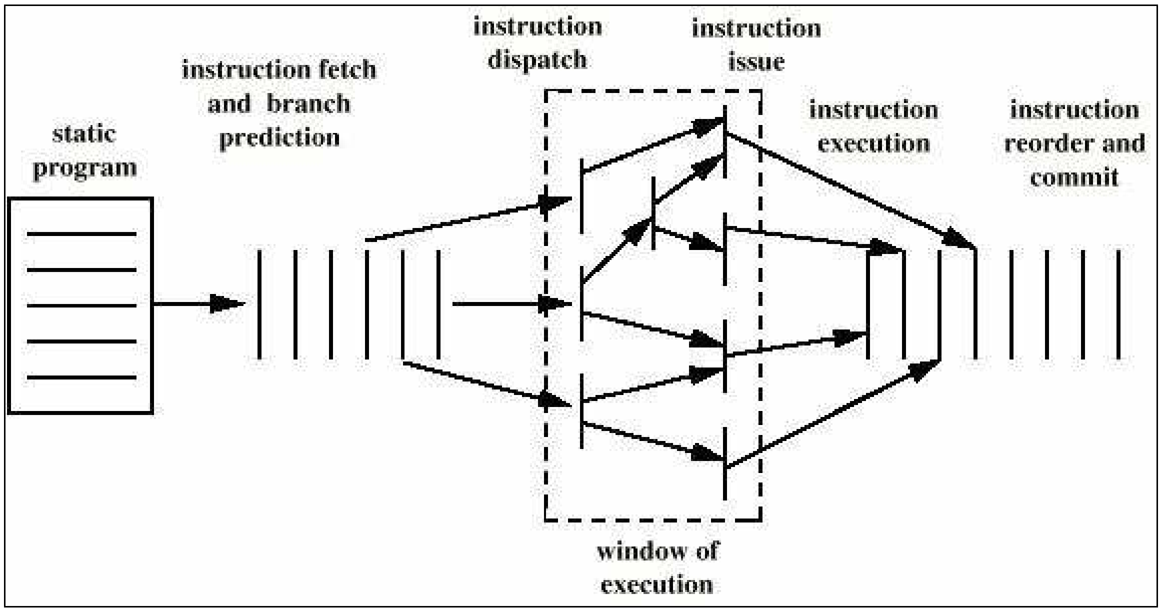
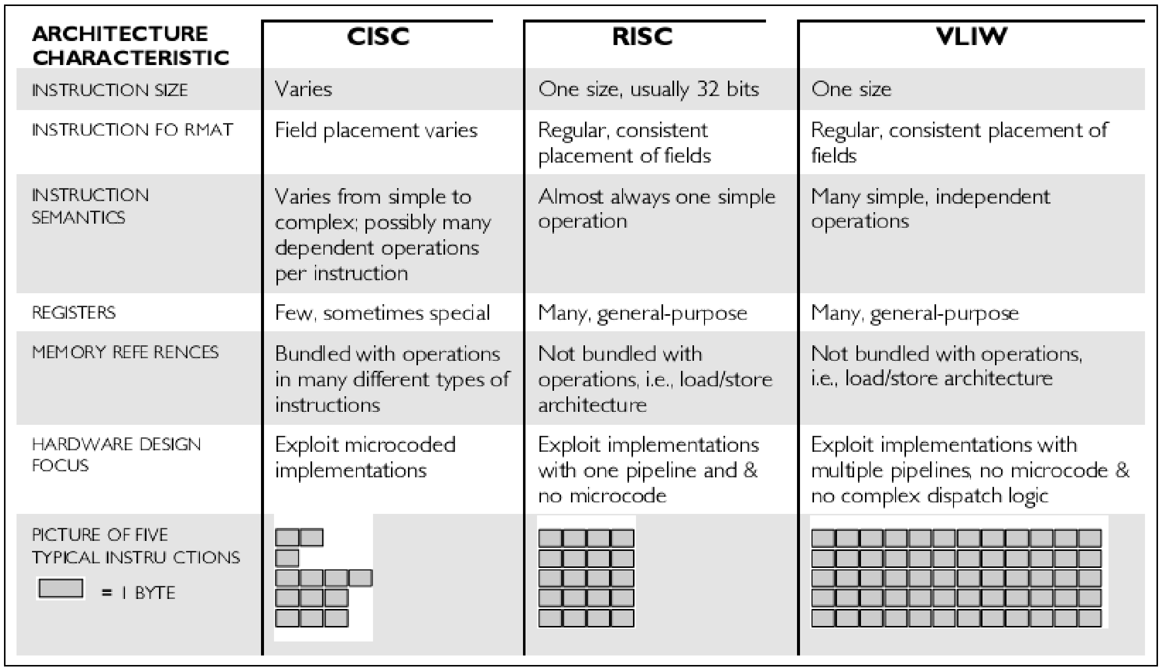


Figure 9: Superscalar Architecture execution [106]

Figure 9 shows all three task that need to be completed to process instructions in parallel. The instruction fetch stage fetches several instructions at a time. The instructions that were fetched are then checked for dependencies and dispatched to a functional unit. The instruction is executed.

5 Comparison: CISC, RISC, VLIW



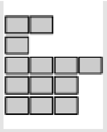
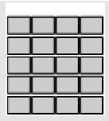
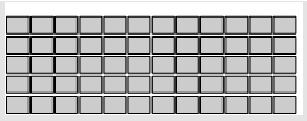
ARCHITECTURE CHARACTERISTIC	CISC	RISC	VLIW
INSTRUCTION SIZE	Varies	One size, usually 32 bits	One size
INSTRUCTION FO RMAT	Field placement varies	Regular, consistent placement of fields	Regular, consistent placement of fields
INSTRUCTION SEMANTICS	Varies from simple to complex; possibly many dependent operations per instruction	Almost always one simple operation	Many simple, independent operations
REGISTERS	Few, sometimes special	Many, general-purpose	Many, general-purpose
MEMORY REFE RENCES	Bundled with operations in many different types of instructions	Not bundled with operations, i.e., load/store architecture	Not bundled with operations, i.e., load/store architecture
HARDWARE DESIGN FOCUS	Exploit microcoded implementations	Exploit implementations with one pipeline and & no microcode	Exploit implementations with multiple pipelines, no microcode & no complex dispatch logic
PICTURE OF FIVE TYPICAL INSTRU CTIONS ▭ = 1 BYTE			

Table 2: Characteristics Comparison of CISC, RISC and VLIW

VLIW is similar to RISC. A VLIW instruction is pretty much several RISC instructions. The size of the VLIW instruction depends on the number of functional units.

6 VLIW Processors

VLIW stands for Very Long Instruction Word. From Table 2 we can see that the instruction size is very large compared to RISC and CISC architectures. VLIW instructions have a set size like RISC instructions. One VLIW instruction would incorporate several RISC like instructions. Figure 10 show a general VLIW architecture and its instructions format.

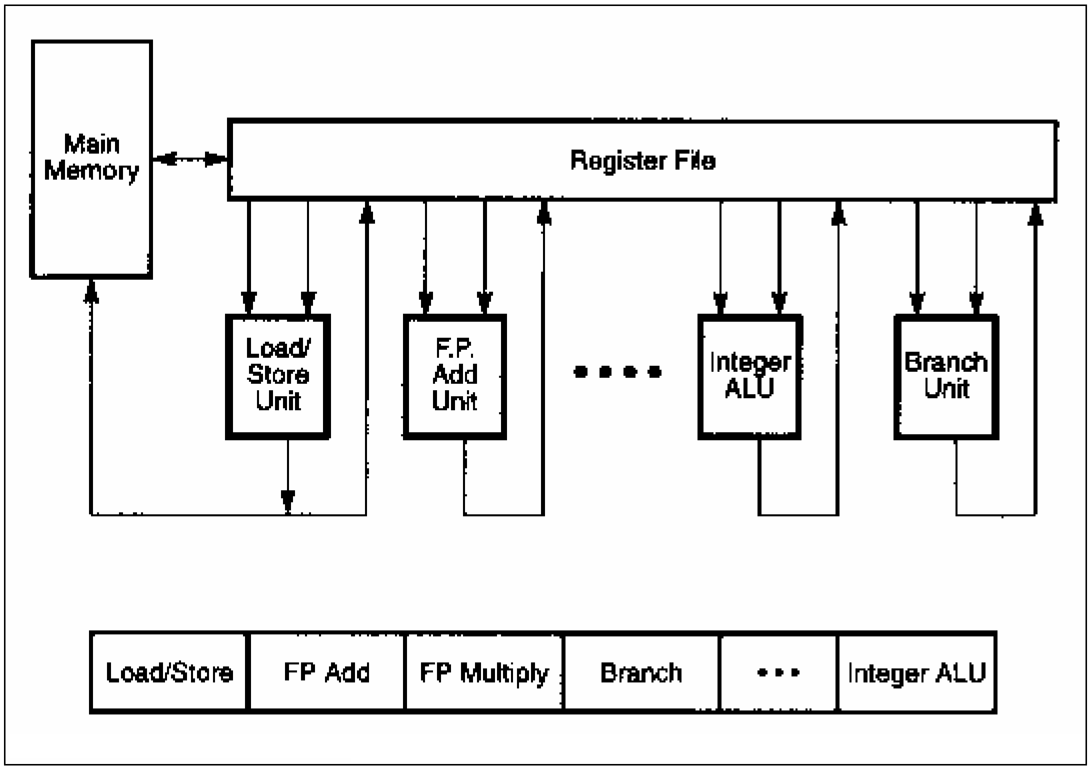


Figure 10: General VLIW Architecture and its Instructions Format [106]

The size of the VLIW instruction depends on the number of functional units the processor has. In figure 10, the instruction format has a slot for every functional unit. If the compiler wants to do an ADD instruction it would put the instruction in the integer ALU slot.

6.1 Architecture

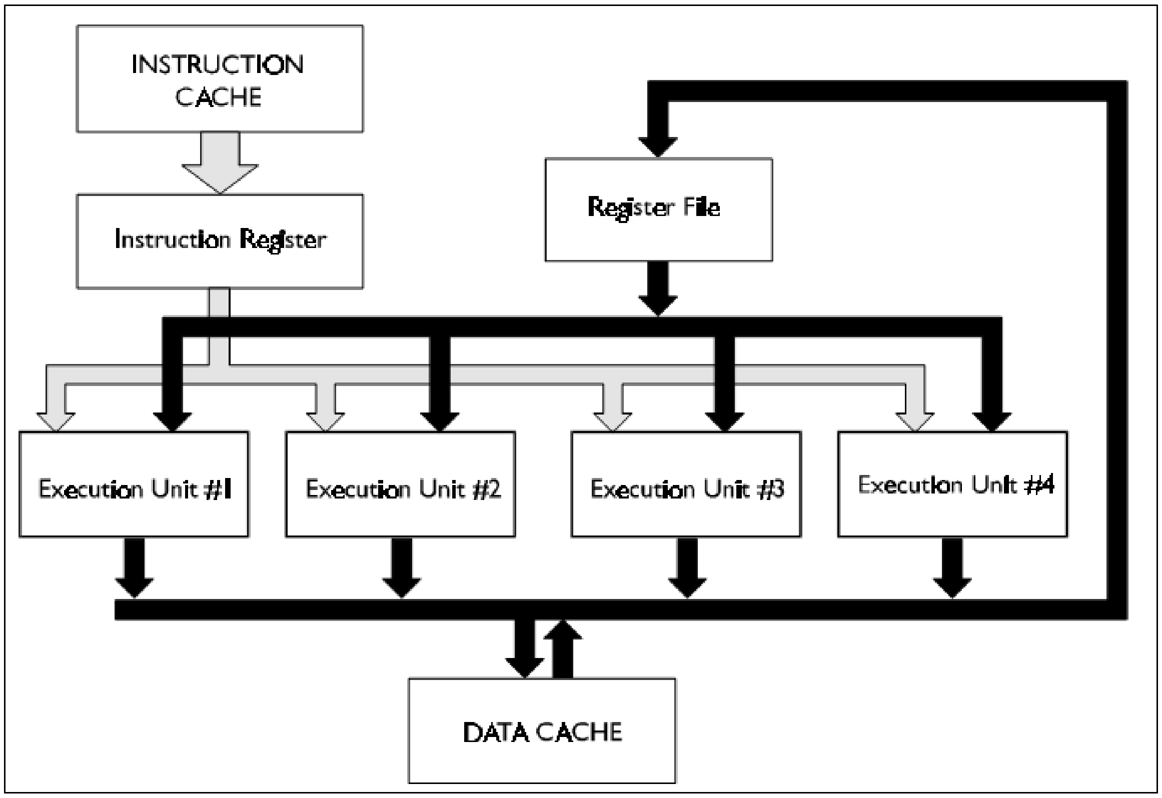


Figure 11: Generic VLIW Implementation

If you compare Figure 11 to Figure 7, you will notice that Figure 11 is a lot simpler. In the VLIW architecture the compiler handles all three tasks that are needed to process instructions in parallel. The VLIW processor just executes the instruction it receives in the order it receives it.

Figure 12 shows a VLIW compiler creating VLIW instructions. The compiler does the grouping of independent instructions. The compiler assigns the instruction to a functional unit by putting the instruction in the proper slot. The compiler handles the initiation task by ordering the VLIW instructions in the order the processor has to execute them. So, a VLIW processor would execute instruction-1 then instruction-2 and so forth. The VLIW executes the instructions sequentially.

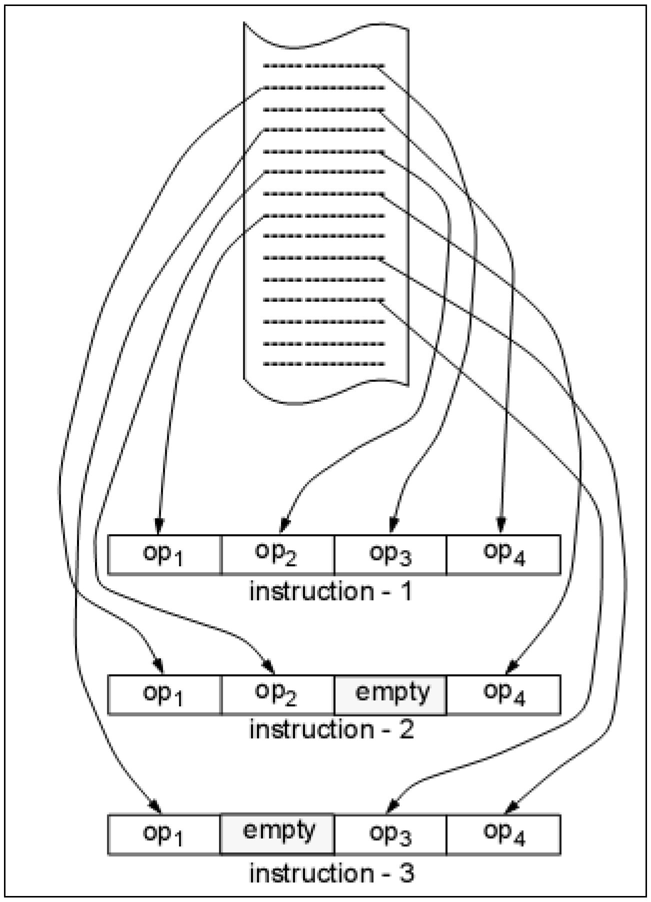


Figure 12: VLIW Compiler Creating VLIW Instructions [106]

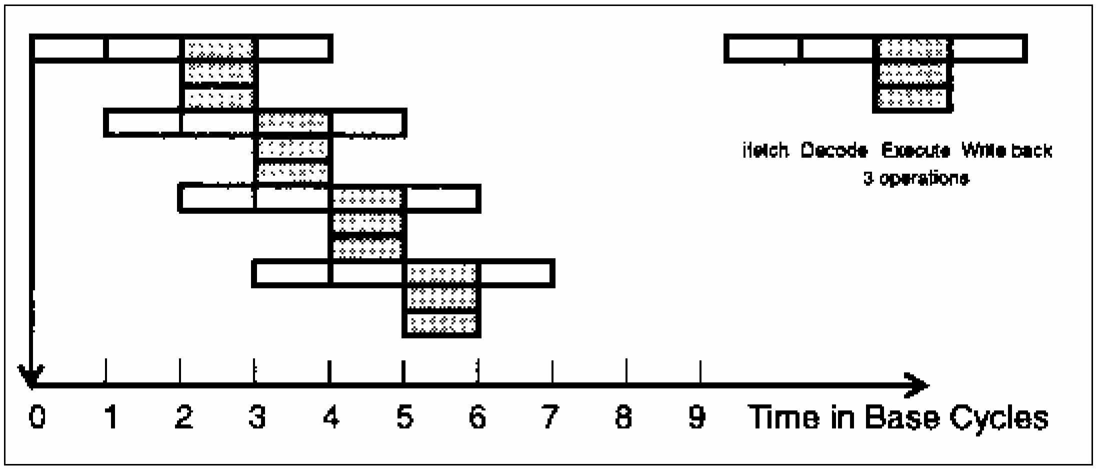
The Pros and Cons of VLIW Architectures

Pros for VLIW architecture:
Compiler prepares fixed packets of multiple operations that give the full "plan of execution" [106]
dependencies are determined by compiler and used to schedule according to function unit latencies
function units are assigned by compiler and correspond to the position within the instruction packet ("slotting")
compiler produces fully-scheduled, hazard-free code => hardware doesn't have to "rediscover" dependencies or schedule
One VLIW instruction contains all the instructions that need to be processed in parallel [193].
The process has to do only one fetch.
Simple hardware [193]
No hardware is needed for run-time detection of parallelism.
The number of functional units can be increased without needing additional sophisticated hardware to detect parallelism, like in Superscalar processors.
Window of execution problem that exist in Superscalar processor is solved [193].

The compiler can potentially analyse the whole program in order to detect parallel operations.

Cons against VLIW processors:
Compatibility across implementations is a major problem [106].
VLIW code won't run properly with different number of function units or different latencies
unscheduled events (e.g., cache miss) stall entire processor
Code density is another problem [106]
low slot utilization (mostly nops)
reduce nops by compression ("flexible VLIW", "variable-length VLIW")
Incompatibility of binary code [193].
If a new version of the processor with additional functional units is introduced, the number of operations possible to execute in parallel is increased. But the VLIW instruction size increases because of the additional functional unit. The old binary code cannot be run on this processor because of the different instruction sizes.
High bandwidth between instruction cache and fetch unit is needed [193].
Caused by the large size of the VLIW instruction.
One instruction with 7 operations (slots) ,each 24 bits needs 168 bits/instruction

VLIW Execution


Fetch Decode Execute Write back
3 operations
0 1 2 3 4 5 6 7 8 9 Time in Base Cycles
Figure 13: A VLIW Processor of Degree m=3 [106]

In Figure 13, the degree value m is the number of functional units the processor has, m=3. So at most the processor can execute 3 executions simultaneously.. If you compare Figure 13 to Figure 8 you will notice that they both take 7 clock cycles to finish executing. Unlike Superscalar processors, the instructions that need to be executed in parallel are fetched and decoded in one operation.

Loop Unrolling to Increase Parallelism for VLIW Compilers

Loop unrolling is a technique to increase the potential parallelism in a program [193].

```
for (i=959; i >= 0; i--)
x[i] = x[i] + s;
```

Figure 14: A Simple For-Loop

```
for (i=959; i >= 0; i-=3){
x[i] = x[i] + s;
x[i-1] = x[i-1] + s;
x[i-2] = x[i-2] + s;
}
```

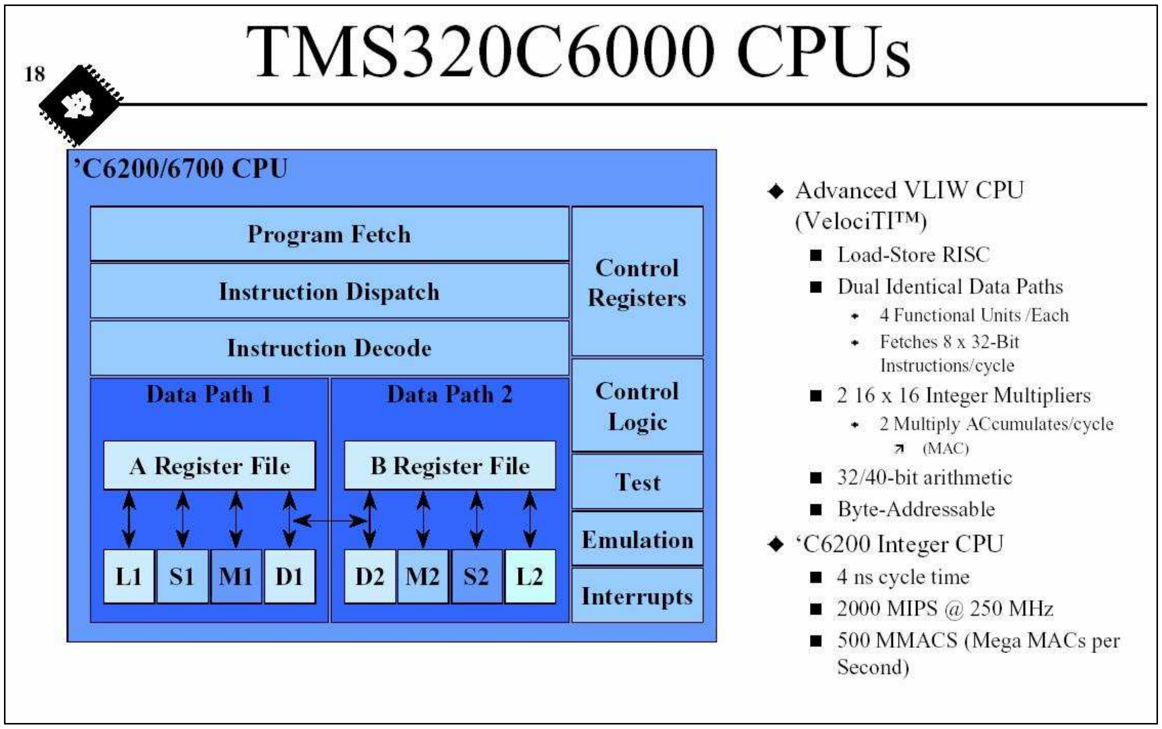
Figure 15: A For-Loop with 3 Iterations Unrolled

The For-Loop in Figure 14 has no parallelism at all. Each iteration of the For-Loop takes 6 cycles. Ignoring loop overhead, the For-Loop in Figure 14 takes 960* 6= 5760 cycles. Figure 15 uses the loop unrolling technique and unrolls 3 iterations. Each Iteration of the For-Loop in Figure 15 takes 7 cycles [193]. Figure 15 takes 960/3*7=2240 cycles to execute. There is a limit to the number of iterations that can be unrolled for a given VLIW processor.

Trace Scheduling

Trace scheduling exploits parallelism across conditional branches by using compile time branch prediction [193]. Three steps are needed to implement trace scheduling :
Trace Selection
Instruction scheduling
Replacement and Compensation

A trace is a sequence of basic blocks, likely to be executed most of the time [193]. Traces are defined during compile time by the compiler. After the traces are defined, the instructions in a trace are executed in parallel. In the instruction execution stage instructions are moved across branches. This will lead to errors unless compensation code is added.
TMS320C6x VLIW DSP



TMS320C6000 CPUs

- Advanced VLIW CPU (VelociTI™)
  - Load-Store RISC
  - Dual Identical Data Paths
    - 4 Functional Units /Each
    - Fetches 8 x 32-Bit Instructions/cycle
  - 2 16 x 16 Integer Multipliers
    - 2 Multiply ACcumulates/cycle
      - (MAC)
  - 32/40-bit arithmetic
  - Byte-Addressable
- 'C6200 Integer CPU
  - 4 ns cycle time
  - 2000 MIPS @ 250 MHz
  - 500 MMACS (Mega MACs per Second)

Figure 16: TMS320C6000 VLIW DSP Processor

Summary

7.1 Dot Product

The dot product can be calculated by the equation below.

$$Y = \sum_{n=1}^{N} a(n)\ \ x(n)$$

We need to Store a(n) and x(n) into an array of N elements

```
; clear A4 and initialize pointers A5, A6, and A7
        MVK  .S1  40,A2       ; A2 = 40 (loop counter)
loop    LDH  .D1  *A5++,A0; A0 = a(n)
        LDH  .D1  *A6++,A1; A1 = x(n)
        MPY  .M1  A0,A1,A3           ; A3 = a(n) * x(n)
        ADD  .L1  A3,A4,A4 ; Y = Y + A3
        SUB  .L1  A2,1,A2     ; decrement loop counter
[A2]    B    .S1  loop   ; if A2 != 0, then branch
        STH  .D1  A4,*A7      ; *A7 = Y
```

Figure 17: TMS320C6000 Code for calculating the Dot Product

In Figure 17, we are only using half the functional units, the ones in Data Path 1. We can speed up the calculation of the dot product by calculating the dot products for the odd indexed terms and even indexed terms separately, in different data paths. This way we can use all 8 functional units to calculate the dot product, see Figure 18. The code in Figure 18 has a higher degree a parallelism than the code in Figure 17.

```
; clear A4 and initialize pointers A5, A6, and A7
        MVK  .S1  40,A2      ; A2 = 40 (loop counter)
loop    LDW  .D1  *A5++,A0           ; load a(n) and a(n+1)
        LDW  .D2  *B6++,B1           ; load x(n) and x(n+1)
        MPY  .M1X A0,B1,A3           ; A3 = a(n) * x(n)
        MPYH .M2X A0,B1,B3           ; B3 = a(n+1) * x(n+1)
        ADD  .L1  A3,A4,A4 ; Yeven = Yeven + A3
        ADD  .L2  B3,B4,B4 ; Yodd = Yodd + A3
        SUB  .S1  A2,1,A2    ; decrement loop counter
[A2]    B    .S2  loop  ; if A2 != 0, then branch
        ADD  .L1  A4,B4,A4 ; Y = Yodd + Yeven
        STH  .D1  A4.*A7     : *A7 = Y
```
Figure 18: Optimized TMS320C6000 Code for calculating the Dot Product

Paper 1 Optimizing Loop Performance for Clustered VLIW Architectures [195]
Paper 2 A Code Generation Framework for VLIW Architectures with Partitioned Register Files [196]

Lecture #22 Superscalar Processors

Course: Computer Architecture III
Lecturer: Miodrag Bolic
Scribe: Francisco José García Rodrigo, Ana Lallena Arquillo
Date: October 15, 2007

Introduction

In Part III of this course, we are studying different architectures that implement Single Processor Parallelism. Together with superscalar processors, this part of the course has covered, in previous lectures Vector Architectures and Very Long Instruction Word Processors.

In this lecture, we learn different concepts about the superscalar architecture. Beginning with some general ideas about superscalar processors, we will continue studying the differences and similarities between superscalar and superpipelined processors. After, we will study different types of parallelism and design issues, and we will end up going deeper in the superscalar architecture.

Explanation

Superscalar processors

Superscalar processors implement a form of parallelism called instruction-level parallelism. This kind of parallelism allows multiple instructions of the same program to be executed in parallel (later on this document, we will study the differences between instruction-level parallelism and machine parallelism), reducing the time a certain program needs to be executed.

Superscalar processors try to find different independent instructions that can be executed at the same time, fetching and executing them at the same time (depending of the degree of parallelism. A superscalar architecture executes more than one instruction during a single pipeline stage by fetching multiples instructions and dispatching them simultaneously to different functional units in the processor. However, not all the functional units are operating at a given time, and some of the pipelines may be stalling in a waiting state.

The general superscalar organization is shown in figure 1, where we can see that in a superscalar processor there are different pipelined functional units. This units share an Integer Register File and a Floating Point Register File, so they can initiate and execute different instructions at the same time, whenever the instructions are non-dependent.

The **advantages** of the superscalar architectures are the fact that the hardware solves everything (detects parallelism between instructions, tries to issue as many instructions as possible in parallel and solves register renaming – explained in *Register renaming* section) and the binary compatibility. Binary compatibility allows to add new functional units to the architecture, without being necessary to change the instruction sets, so old programs can benefit from the additional parallelism.

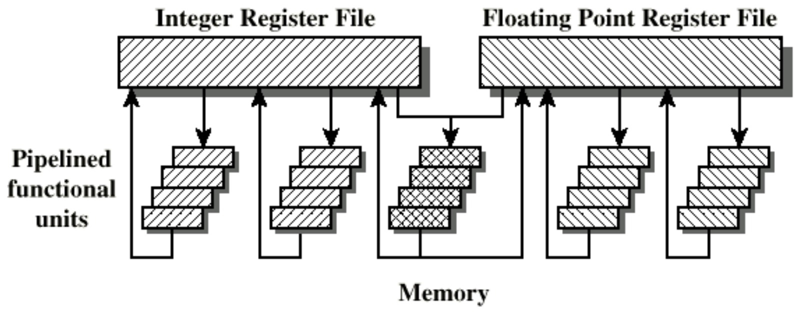


Figure 1. General superscalar organization

The **drawbacks** are the complexity since much hardware is needed for runtime detection (especially when detecting dependencies among instructions) and the limited capacity to detect potentially parallel instructions.

Although superscalar processors are theoretically able to be used in RISC or CISC architectures, in practice, superscalar architecture is almost only applied to RISC processors.

Superscalar vs. Superpipelined

Superpipelined processors "exploit the fact that many pipelined stages perform tasks that require less than half a clock cycle" [198]. Therefore, if the speed of the internal clock is doubled, two instructions can be executed in just one external clock cycle.

"A pipeline has four stages: instruction fetch, operation decode, operation execution, and result write back" [198]. In a given processor, "although several instructions are executing concurrently, only one instruction is in its execution stage at any one time".

In figure 2, we can see how the execution of several instructions differ when executing them in a normal, a superpipelined or a superscalar machine.

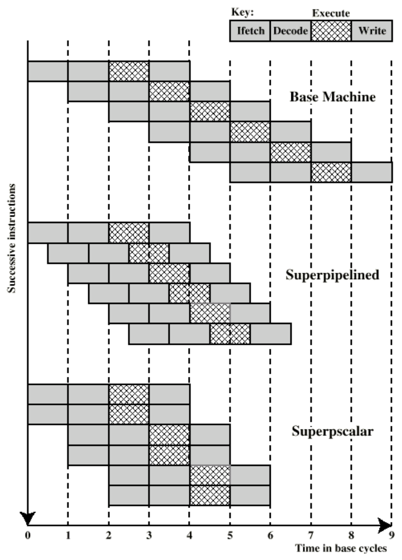


Figure 2. Comparison of superscalar and superpipelined approaches [2]

As we can see, in the superscalar scheme two instructions are fetched at the same time in each clock cycle. In the superpipelined approach, to instruction are fetched in the same clock cycle, also, but in this case, it occurs because the double speed of the internal clock allows fetching a new instruction each half a clock cycle. In the base machine case, two instructions cannot be on the same stage at any one time.

Therefore, the superscalar architecture can have more than one instruction in the same stage at a given time, depending on the degree of parallelism of the processor, while superpipelining does not permit that to happen.

Limitations of superscalar scheme

“The superscalar approach depends on the ability to execute multiple instructions in parallel” [198]. Therefore, the degree of efficiency of this scheme has a lot to do with the capacity of the system for “maximize the instruction-level parallelism” [198]. This parallelism can be improved using “a combination of compiler-based optimization and hardware techniques” [198].

The maximum obstacle to obtain a high degree of parallelism is the dependencies that can take place among the different instructions. We are going to study the five different types of dependencies:

1. True data dependency

This type of dependency occurs when one instruction depends on the result of a previous one to be executed. For example:

ADD r1, r2 (r1 := r1+r2)
MOVE r3,r1 (r3 := r1)

In this example, both instructions can be fetched and decoded in parallel, but to execute the second instruction, it is necessary that the first one has finished, since the "second instruction needs data produced by the first instruction" [198].
This kind of dependency would have no effect if the execution took place in a superpipelined machine.

2. Procedural dependency

"The instructions following a branch have a procedural dependency on the branch and cannot be executed until the branch is executed" [198]. Therefore, it is not possible to execute in parallel the instructions before and after a branch, since the result of the branch is not known *a priori*.
This type of dependency affects both superscalar and superpipelined processors, although the consequences are more severe for superscalar architectures.

Also, when the length of the instructions is not fixed, "it is necessary to partially decode each instruction before the following one can be fetched" [198] Therefore, it is not possible to fetch more than one instruction at the same time, as it is necessary in the superscalar pipeline to be efficient.

3. Resource conflicts

When two or more instructions require access to the same resource at the same time, there is a competition for that resource. "Examples of resources are: memories, caches, buses, register-file ports and functional units" [198].

Resource conflicts are considered to be similar to data dependencies. However, resource conflicts can be eliminated by duplicating the resources or pipelining the appropriate functional unit, while data dependencies cannot be overcome [198].

4. Output dependency

Also known as write-write dependency, it occurs when two instructions or more modify the same variable but one of them has to sent/store after the other. To illustrate this type of dependency, we are going to study the following example provided in [198].

We have the following instructions:

I1: R3 $\leftarrow$ R3 cp R5
I2: R4 $\leftarrow$ R3 + R1

I3: R3 ← R5 + R1
I4: R7 ← R3 op R4

Output dependency is the one existing between I1 and I3 since, if I3 finishes its execution before I1, when the value of R3 is fetched for the execution of I4, the stored value in R3 will be wrong, since will be the corresponding to instruction I1 instead of being the one corresponding to instruction I3.

5. Antidependency

It is also called read-write dependency, and it occurs when one instruction reads a location another instruction has changed. We can observe this in the following example, obtained from [198].

The following instructions are executed:

I1: R3 ← R3 op R5
I2: R4 ← R3 - 1
I3: R3 ← R5 - 1
I4: R7 ← R3 op R4

In this example, if I3 finished its execution before I2 started being executed, I2 would be fetching an incorrect value of R3, since R3 would have been updated by I3.

In figure 3. we can see how the different dependencies change the number of cycles needed to execute a certain program, in comparison with the same program without dependencies.

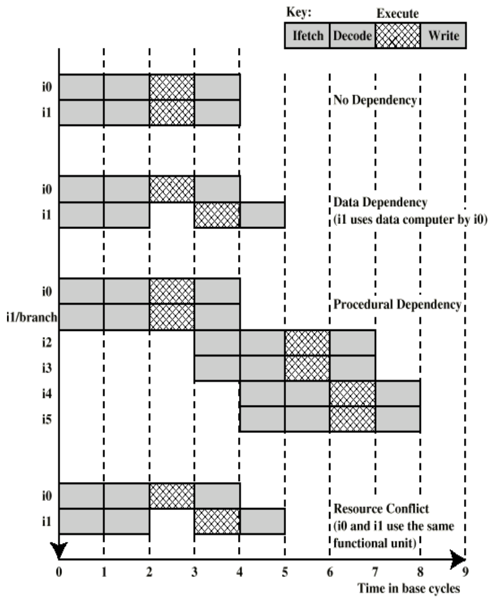


Figure 3. Effect of dependencies

Design Issues

"The term *instruction issue* is used to refer to the process of initiating instruction execution in the processor's functional units and the term *instruction issue policy* to refer to the protocol used to issue instructions" [198].
"In essence, the processor tries to look ahead of the current point of execution to locate instructions that can be brought into the pipeline and executed" [198].
Taking into account the different kinds of dependencies, it is important the order in which instructions are fetched and executed. It will also be necessary to order the instructions in such a way that they update the locations of memory in the sequence given by the program.

To see the difference among the different designing issues, we are going to see the difference when processing the following example.

<u>Example:</u>

We have instructions I_1 to I_6. We have a superscalar processor (figure 4) with 3 functional units and the capacity of decode 2 instructions at the same time and write 2 results at the same time. We have to execute them with the following restrictions:

- I_2, …, I_6 require 1 cycle to execute.
- I_1 requires 2 cycles
- I_3 and I_4 have a conflict for the same FU
- I_5 depends on the result of I_4
- I_5 and I_6 have a conflict for the same FU

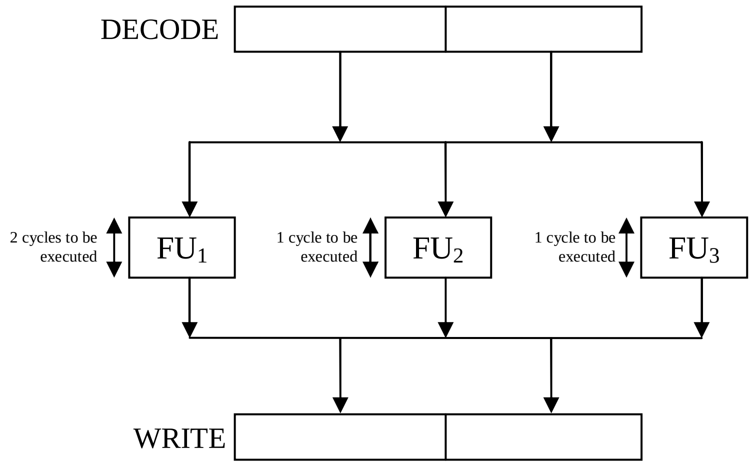


Figure 4. Superscalar processor corresponding to the design issue example

In-order issue in-order completion

Instructions are issued and fetched in the order they occur, which is not very efficient because everything has to be done in order.
Since we can decode and write 2 instructions at the same time, we can decode two instructions and execute them. Every FU is different and is specialised in a particular task. I_1 takes two clock cycles to be executed while I_2 takes one cycle. During the execution of these two instructions, we can decode I_3 and I_4, but we cannot execute them in the next cycle because we need I_1 to finish before we can execute any of the others (everything has to be in order). Since I_5 and I_6 and I_3 and I_4 are in competition to use the same FU, it is necessary to assign them one after the other. This causes a great lost of clock cycles (figure 5).

DECODE		EXECUTE			WRITE		Cycles
I_1	I_2						1
I_3	I_4	I_1	I_2				2
I_3	I_4	I_1					3
	I_4			I_3	I_1	I_2	4
I_5	I_6			I_4			5
	I_6		I_5		I_3	I_4	6
			I_6				7
					I_5	I_6	8

Figure 5. In-order issue in-order completion

In-order issue out-of-order completion

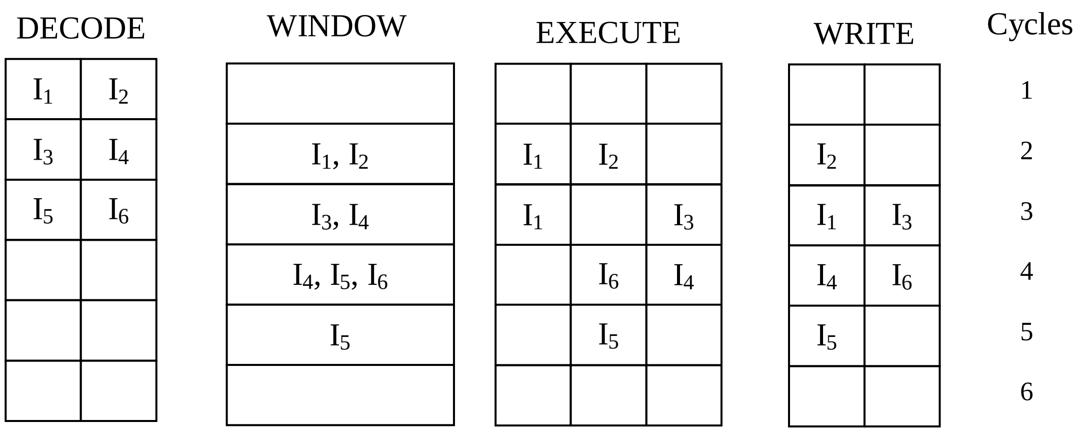
In Out-Of-Order completion we take the instructions in the order they come but we do not need to execute them in that order. Therefore, we can start executing I_3 before I_1 has finished, since they are independent, and we do not need either to write the products of the execution of each instruction in the order the instructions were fetched (figure 6).

DECODE		EXECUTE			WRITE		Cycles
I_1	I_2						1
I_3	I_4	I_1	I_2				2
	I_4	I_1		I_3	I_2		3
I_5	I_6			I_4	I_1	I_3	4
	I_6		I_5		I_4		5
			I_6		I_5		6
					I_6		7

Figure 6. In-order issue out-of-order completion

Out-of-order issue out-of-order completion

“With in-order issue, the processor will only decode instructions up to the point of a dependency or conflict. No additional instructions are decoded until the code is resolved” [198]. Thus, out-of-order issue requires the use of a buffer known as Decode Instruction Window. After one processor fetches and decodes an instruction, it stores it in the instruction window, and continues fetching and decoding instructions while the buffer is not full. “When a functional unit becomes available in the execute stage, an instruction from the instruction window may be issued to the execute stage. Any instructions may be issued, provided that (1) it needs the particular functional unit that is available and (2) no conflicts or dependencies block that instruction.



DECODE		WINDOW	EXECUTE			WRITE		Cycles
I_1	I_2							1
I_3	I_4	I_1, I_2	I_1	I_2		I_2		2
I_5	I_6	I_3, I_4	I_1		I_3	I_1	I_3	3
		I_4, I_5, I_6		I_6	I_4	I_4	I_6	4
		I_5		I_5		I_5		5
								6

Figure 7. Out-of-order issue out-of-order completion

Register renaming

Out-of-order issue and out-of-order completion increase the possibilities of violating antidependencies and output dependencies. This occurs because register contents may not reflect the correct ordering from the program. “Multiple instructions compete for the use of the same register locations generating pipeline constraints that retard performance of the processor” [198].

Register renaming is a technique based on the duplication of resources. “In essence, register are allocated dynamically by the processor hardware; and they are associated with the values needed by instructions at various points in time. When a new register value is created, a new register is allocated for that value. Subsequent instructions that access that value as a source operand in that register must go through renaming process: the register referenced in those instructions must be revised to refer to the register containing the needed value. Thus, the same original register reference in several different instructions may refer to different actual registers, if different values are intended” [198].

We can see the following as a n example of register renaming:

I1: R3b $\leftarrow$ R3a + R5a

```
I2:  R4b  ← R3b + 1
I3   R3c  ← R5a + 1
I4:  R7b  ← R3c + R4b
```

Note that now R3 has a letter attached that simplifies the control of the dependencies. Without subscript refers to logical register in instruction. With subscript is hardware register allocated

Branch Prediction

(all the information in this section has been obtained from [198].

When we are using pipelined machines, we have to deal with the different branches that can occur. The prediction of branches is useful since it allows the execution of instructions that are after the branch in parallel with instructions that are before the branch. There are different ways of predicting branches, and we are going to see some of them now:

- The INTEL 80846 solved the problem of branch prediction fetching the sequential instructions after a branch and fetching the branch target instruction [198].
- RISC machines calculate the result of a conditional operation before fetching any instructions that would follow that operation.
- Superscalar machines have gone back to "pre-RISC techniques of branch prediction" [198].

Machine parallelism vs. Instruction level parallelism

As we have seen in previous sections, instruction-level parallelism corresponds to the degree of overlapping that can be reached when executing sequential and independent instructions. "Instruction level parallelism is determined by the frequency of true data dependencies and procedural dependencies in the code […] and also by the time until the result of an instruction is available for use as an operand in a subsequent instruction (operation latency). The latency determines how much of a delay a data or procedural dependency will cause." [198].

Machine parallelism "is a measure of the ability of the processor to take advantage of instruction-level parallelism". Machine parallelism is obtained via duplication of resources, register renaming and our of order issuing; and "is determined by the number of instructions that can be fetched and executed at the same time (the number of parallel pipelines) and by the speed and sophistication of the mechanisms that the processor uses to find independent instructions" [198].

It is important to distinguish between instruction-level parallelism and machine parallelism. Instruction level parallelism can be implemented in the machine level.

Architecture of Superscalar Processors

Dynamic Pipelines

A dynamic pipeline is “a parallel pipeline that supports out-of-order execution of instructions” [199]. By means of the use of different buffers between the different stages, it “allows the instructions to enter and leave the buffers in different orders”. We can see a dynamic pipeline in figure 8.

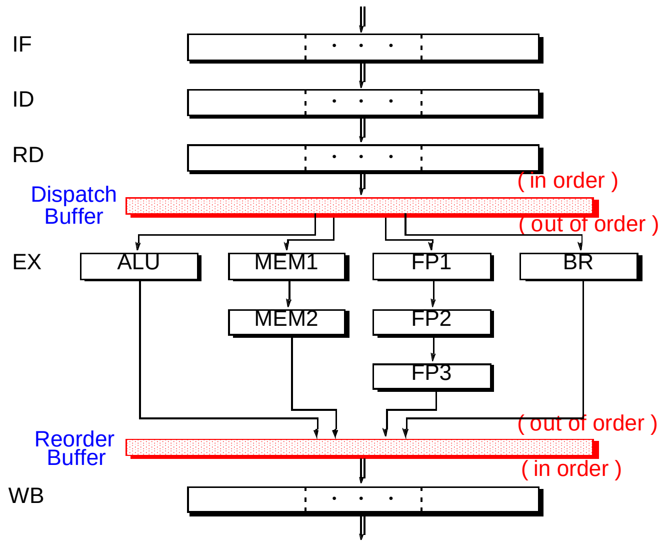


Figure 8. Dynamic pipeline

Summary

Superscalar Pipeline

To execute an instruction in a superscalar pipeline, it is necessary to pass through the following stages: Instruction Fetch, Instruction Decoding, Instruction Dispatch, Execution, Completion and Retirement.

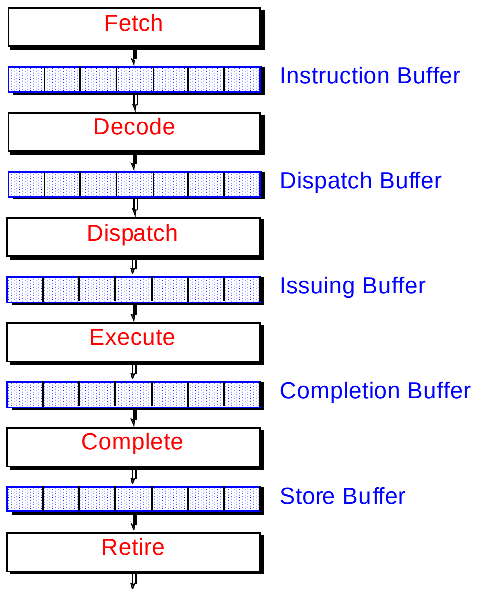


Figure 9. The 6-stage TEMPLATE (TEM) superscalar pipeline [199]

These six stages are not the same as in RISC processors. In RISC the execution of an instruction can take 1 cycle or 3 clock cycles.
In fact, it can take different time to decode different instructions because the instructions are different.

The first step is fetching the instruction and then decoding it to analyse dependencies with other instructions. After that, the instructions are dispatched, that is, assigned to different functional units, depending on the type of the instruction, and trying to get an optimal or a good total time of execution.
Since out-of-order issue is used, after the execution of the instruction, in the completion step, the results are written when they are done, but we can not store then in memory before being sure that the right branch is closed.

Between stages, there are buffers in order to store the instructions/data necessary depending on the policy used (in order or out-of-order).

PAPER 1 SUPERSCALAR VS. SUPERPIPELINED MACHINES.

PAPER 2 THE EFFECT OF EMPLOYING ADVANCED BRANCHING MECHANISMS IN SUPERSCALAR PROCESSORS

Lecture #23: Superscalar Processors – Review processors

Course: Computer Architecture III
Lecturer: Miodrag Bolic
Scribes: Senthooran Rajalingam (1007831) and Chenghai Dai (2142158)
Date: November 29, 2006

Introduction

1. This lecture first introduces the dependence graph to represent the dependence existing in the instruction which includes flow dependence, output dependence and anti dependence, and then further investigates the superscalar processor architecture, pipeline and data dependence by several examples.

Explanation

2. Lecture Theory

2.1 Superscalar processor

Superscalar processor can initiate multiple instructions in the same clock cycle. Nearly all modern microprocessors, including the Pentium, PowerPC, Alpha, and SPARC are superscalar [200]. Superscalar processor can initiate multiple instructions in the same clock cycle. A typical superscalar processor fetches and decodes several instructions at the same time. Instructions are executed in parallel on the basis of the availability of operand data instead of their original program sequence. Upon completion, instructions are re-sequenced in order to update the process state in the correct program order.

2.2 Superscalar architecture

Superscalar architecture is used in many superscalar processors for parallel computing. Superscalar architecture is a way of parallel computing used in many superscalar processors. In a superscalar computer, CPU manages multiple instruction pipelines to execute several instructions in the same clock cycle. This happens by feeding the different pipelines through a number of execution units within the processor [201]. To successfully implement a superscalar architecture, the CPU's instruction fetching mechanism has to intelligently retrieve and delegate instructions. If not, pipeline stalls may occur, resulting in execution units that are often idle.

2.3 Dependencies

All instructions in the window of execution to be executed are subject to dependence and resource constraints. Three types of dependencies can be identified in superscalar processor: (1) true data dependency, (2) output dependency, and (3) anti-dependency.

2.3.1 True Dependency

True data dependency exists when the output of one instruction is required as an input to a subsequent instruction. For example [202]:

```
MUL    R4, R3, R1    R4 ← R3 * R1
ADD    R2, R4, R5    R2 ← R4 + R5
```

True data dependencies are intrinsic features of the user's program. They cannot be eliminated by compiler or hardware techniques. True data dependencies have to be detected and treated. The addition above cannot be executed before the result of the multiplication is available. The simplest solution is to stall the adder until the multiplier has finished. In order to avoid the adder to be stalled, the compiler or hardware can find other instructions which can be executed by adder until result of the multiplication is available.

2.3.2 Output Dependency

An output dependency exists if two instructions are writing into the same location; if the second instruction writes before the first one, an error occurs. For example [202]:

```
MUL    R4, R3, R1    R4 ← R3 * R1
ADD    R4, R2, R5    R4 ← R2 + R5
```

2.3.3 Anti-dependency

Anti-dependency exists if an instruction uses a location as an operand while a following one is writing into that location; if the first one is still using the location when the second one writes into it, an error occurs. For example [202]:

```
MUL    R4, R3, R1    R4 ← R3 * R1
ADD    R3, R2, R5    R3← R2 + R5
```

2.3.4 The nature of output dependency and anti-dependency

Output dependencies and anti-dependencies are not real data dependencies since they are not intrinsic features of the execution, but storage conflicts. Output dependencies and anti-dependencies are only the consequence of the manner in which the programmer or the compiler is using registers (or memory locations). They are produced by the competition of several instructions for the same register. In the previous examples, the conflicts are produced only since the output dependency: R4 is used by both instructions to store the result; and the anti-dependency: R3 is used by the second instruction to store the result.

The examples [202] could be written as follows without dependencies by using additional registers:

```
MUL     R4, R3, R1     R4 ← R3 * R1
ADD     R7, R2, R5     R7← R2 + R5
MUL     R4, R3, R1     R4 ← R3 * R1
ADD     R6, R2, R5     R6← R2 + R5
```

2.3.5 Dependence graph

Dependence can be represented in the graph: (1) nodes are instructions; (2) edges are ordered relations among the instructions. In the dependence graph, any ordering-based transformation that does not change the dependencies of the program will be guarantied not to change the result of the program.

- Flow dependence (true dependence) can be represented as **S1** → **S2** [202]**.**
- Output dependence can be represented as **S1**   **S2** [202]**.**
- Anti-dependence can be represented as S1   S2 [202].

Figure 1 shows an example of all three dependence. For Example [202]:

```
S1: Load R1, A      / R1 ← Memory (A) /
S2: Add R2, R1      / R2 ← R2+R1 /
S3: Move R1, R3   / R1 ← R3 /
S4: Store B, R1      / Memory (B) ← R3 /
```

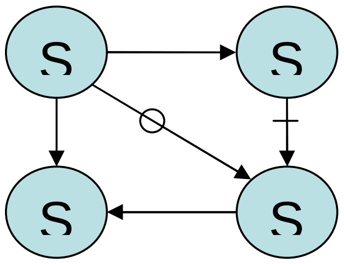


Figure 72 Example of dependence graph [200]

2.3.6 Window of Execution

The instructions in the window of execution are considered by superscalar processor for execution at a certain movement. Window of execution is that the set of instructions are considered for execution at a certain movement. The number of instructions in the window should be as large as possible because data dependencies and only some parts of the instructions are potential subjects for parallel execution. In order to find instructions to be issued in parallel, the processor has to select from a sufficiently large instruction sequence.

2.4 Parallel Instruction Execution

The policies for parallel instruction execution includes: (1) In-order issue with in-order completion; (2) in-order issue with out-of-order completion; (3) Out-of-order issue with out-of-order completion.

2.4.1 In-order issue with in-order completion

Instructions are issued in the exact order that would correspond to sequential execution. Results are written in the same order when they are completed. An instruction cannot be issued before the previous one has been issued. An instruction completes only after the previous one has completed. For example [202], in the following instruction sequence, I1 requires two cycles to execute; I3 and I4 are in conflict for the same functional unit; I5 depends on the value produced by I4. There is a true data dependency between I4 and I5 as well. I2, 15 and I6 are in conflict for the same functional unit.

I1:	ADDF	R12, R13, R14	R12 ← R13 * R14 (float point)
I2:	ADD	R1, R8, R9	R1← R8 + R9
I3:	MUL	R4, R2, R3	R4 ← R2 * R3
I4:	MUL	R5,R6,R7	R5← R6 * R7
I5:	ADD	R10, R5, R7	R10 ← R5 * R7
I6:	ADD	R11,R2,R3	R11← R2 * R3

Decode Issue		Execute			Write back/ Complete		cycle
I1	I2						1
I3	I4	I1	I2				2
I5	I6	I1					3
				I3	I1	I2	4
				I4	I3		5
			I5		I4		6
			I6		I5		7
					I6		8

To guarantee in-order completion, instruction issuing stalls when there is a conflict and when the unit requires more than one cycle to execute. The processor detects and handles (by stalling) true data dependencies and resource conflicts. As instructions are issued and completed in their strict order, the resulting parallelism is very much dependent on the way the program is written/compiled. If I3 and I6 switch position, the pairs I6-I4 and I5-I3 can be executed in parallel.

2.4.2 In-of-Order Issue with Out-Order Completion

Instructions are issued in the exact order that would correspond to sequential execution. Results can be written in out of order when they are completed. An instruction cannot be issued before the previous one has been issued, but an instruction can complete before the previous one. For the same set instruction in 2.1, superscalar processor will use 7 clock cycle instead of 8 clock cycle in section 2.1. The detailed procedures [202] are in the following table.

Decode Issue		Execute			Write back/ Complete		cycle
I1	I2						1
I3	I4	I1	I2				2
I5	I6	I1		I3	I2		3
				I4	I1	I3	4
			I5		I4		5
			I6		I5		6
					I6		7

2.4.3 Out-of-Order Issue with Out-Order Completion

With in-order issue, no new instruction can be issued when the processor has detected a conflict and is stalled, until after the conflict has been resolved. The processor is not allowed to look ahead for further instructions, which could be executed in parallel with the current ones. Out-of-order issue tries to resolve the above problem. Taking the set of decoded instructions the processor looks ahead and issues any instruction, in any order, as long as the program execution is correct. For example [202]:

I1:	ADDF	R12, R13, R14	R12 ← R13 * R14 (float point)
I2:	ADD	R1, R8, R9	R1← R8 + R9
I3:	MUL	R4, R2, R3	R4 ← R2 * R3
I4:	MUL	R5,R6,R7	R5← R6 * R7
I5:	ADD	R10, R5, R7	R10 ← R5 * R7
I6:	ADD	R11,R2,R3	R11← R2 * R3

Decode Issue		Execute			Write back/ Complete		cycle
I1	I2						1
I6	I4	I1	I2				2
I5	I3	I1		I3	I2		3
			I6	I4	I1	I3	4
			I5		I4	I6	5
					I5		6
							7

		☐				☐			☐	8

I6 can be now issued before I5 and in parallel with I4. Comparing to 8 clock cycles if we have in-order issue and 7 clock cycles with in-order issue & out-of-order completion, the sequence takes only 6 cycles.

With out-of-order issue and out-of-order completion, the processor has to consider the true data dependency, output-dependency and anti-dependency. Output dependency can be violated when addition completes before the multiplication. Anti-dependency can be violated when the operand in R3 is used after it has been over-written. Output dependency can give rise to a Write After Write (WAW) hazard, and anti-dependency a Write After Read (WAR) hazard.

2.5 Register renaming

Output dependencies and anti-dependencies can be treated similarly to true data dependencies as normal conflicts. Such conflicts are solved by delaying the execution of a certain instruction until it can be executed. Output dependencies and anti-dependencies can be eliminated by automatically allocating new registers to values, when such a dependency has been detected using register renaming. Adding register renaming to a processor generally gives less of an improvement than changing the instruction set architecture to make the new registers part of the architectural registers, because the compiler cannot use the new registers to store temporary values. A devious strategy is register renaming. The hardware has a large set of registers - often several times as many as the actual architecture claims to have. Register renaming allows the new processor to remain compatible with programs compiled for older versions of the processor since it does not require changing ISA. Consider the following code [202]:

```
I1: MUL R2, R2, R3   ; R2 = R2 * R3
I2: ADD R4, R2, 1    ; R4 = R2 + 1
I3: ADD R2, R3, 1    ; R2 = R3 + 1
I4: DIV R5, R2, R4   ; R5 = R4 * R4
```

Consider one problem: instruction 3 cannot go ahead until instruction 1 has finished, and instruction 2 has started. This is because there is an output dependency between instruction's 1 and 3 - both write to R2 - and an anti-dependency between instruction’s 2 and 3 - instruction 3 overwrites instruction 2's argument. Now consider the same program, but with the registers labelled.

```
I1: MUL R2_b, R2_a, R3_a
I2: ADD R4_b, R2_b, 1
I3: ADD R2_c, R3_a, 1
I4: DIV R5_b, R2_c, R4_b
```

Now instruction 3 can start immediately, because it is using a 'different' R2 from instructions 1 and 2. We are effectively using a history of the contents of each register -

for example, R2_c is the newest version of R2, then R2_b, followed by R2_a (the oldest version).

3. Problem sets:

3.1 How long would the following sequence of instructions take to execute on a superscalar processor with two execution units, each of which can execute any instruction? Load operations have a latency of two cycles, and all other operations have a latency of one cycle. Assume that the pipeline depth is 5 stages. [202]

I1: LD r1, (r2)
I2: ADD r3, r1, r4
I3: SUB r5, r6, r7
I4: MUL r8, r9, r10

For this set instruction, if they are in-order execution and in-order completion with five pipeline stages (Fetch, Decode, Execution, Memory access and Write back), and load has latency of 2 clock cycles. Total number execution of cycles is 8. The detailed procedures are shown in the following table.

F			D			E	E			M		W		clock
I1	***I2***													***1***
I3	I4		I1	I2										2
			I3	I4		I1								3
						I1								4
						I2	I3		I1					5
						I4			I2	I3		I1		6
									I4			I2	I3	7
												I4		8

If above instruction are out-order execution and out-order completion with five pipeline stages (Fetch, Decode, Execution, Memory access and Write back), and load has latency of 2 clock cycles. Total number execution of cycles is 7. The detailed procedures are shown in the following table.

F			D			E			M			W		clock
I1	***I2***													***1***
I3	I4		I1	I2										2

I1	***I2***													***1***
			I3	I4		I1								3
						I1	I3							4
						I2	I4		I1	I3				5
									I2	I4		I1	I3	6
												I2	I4	7

3.2 On an out-of-order superscalar processor with 8 execution units, what is the execution time of the following sequence with and without register renaming it any execution unit can execute any instruction and the latency of all instructions is one cycle? Assume that the hardware register file contains enough registers to remap each destination register to a different hardware register and that the pipeline depth is 5 stages. [202]

I1: LD r7, (r8)
I2: MUL r1, r7, r2
I3: SUB r7, r4, r5
I4: ADD r9, r7, r8
I5: LD r8, (r12)
I6: DIV r10, r8, r10.

Step 1: Figure 2 shows dependence graph without register renaming

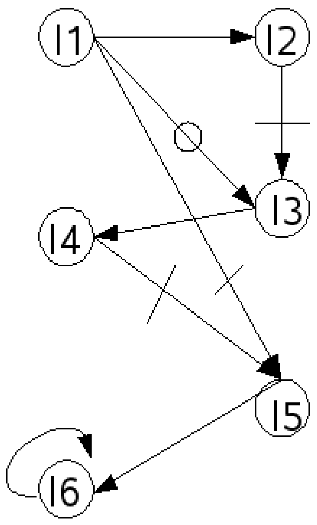


Figure 73 Dependence graph without register renaming

Step 2: For this set instruction without register renaming, total number execution of cycles is 8. The detailed procedures are shown in the following table.

F			D			E			M			W		clock
I1	***I2***													***1***
I3	I4		I1	I2										2
I5	I6		I3	I4		I1								3
			I5	I6		I2	I3		I1					4
						I4	I5		I2	I3		I1		5
						I6			I4	I5		I2	I3	6
									I6			I4	I5	7
												I6		8

In this example, WAR dependencies are a significant limitation on parallelism, forcing the DIV to issue 3 cycles after the first LD, for a total execution time of 8 cycles. In fact, the MUL and the SUB can execute in parallel with extra resources, as can the ADD and the second LD. Using register renaming, the program will be as follows:

```
I1: LD hw7, (hw8)
I2: MUL hw1, hw7, hw2
I3: SUB hw17, hw4, hw5
I4: ADD hw9, hw17, hw8
I5: LD hw18, (hw 12)
I6: DIV hw10, hw18, hw10
```

With register renaming, the program has been broken into three sets of two dependent instructions (LD and MUL, SUB and ADD, LD and DIV). The SUB and the second LD instruction can now issue in the same cycle as the first LD. The MUL, ADD, and DIV instructions all issue in the next cycle, for a total execution time of 6 cycles.

Step 1: Dependence Graph with register renaming as follows

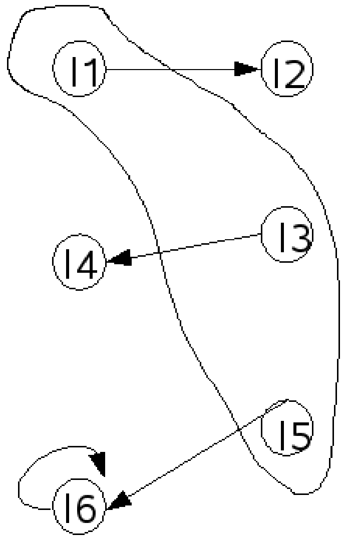
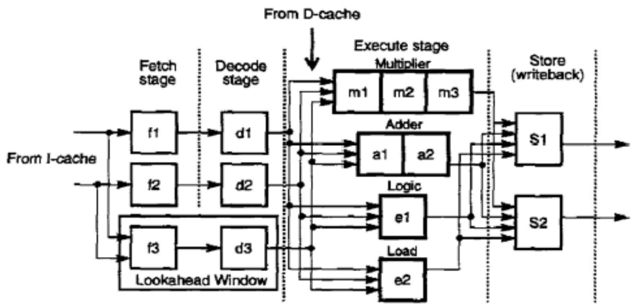


Figure 74 Dependence graph with register renaming

Step 2: For this set instruction with register renaming, total number execution of cycles is 6. The detailed procedures are shown in the following table.

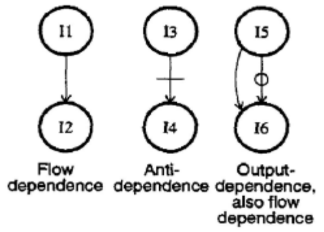
F				D				E					M				W			clock
I1	***I3***	***I5***																		***1***
I2	I4	***I6***		I1	I3	***I5***														2
				I2	I4	***I6***		I1	I3	I5										3
								I2	I4	I6			I1	I3	I5					4
													I2	I4	I6		***I1***	I3	I5	5
																	I2	I4	I6	6

3.4 Figure 4 shows an example of a superscalar processor organization. The processor can issue two instructions per cycle if there is no resource conflict and no data dependence problem. There are essentially two pipelines, with four processing stages (fetch, decode, execute, and store). Each pipeline has its own fetch decode and store unit. Four functional units (multiplier, adder, logic unit, and load unit! are available for use in the execute stage and are shared by the two pipelines and dynamic basis. The two store units can be dynamically used by the two pipelines depending on availability at a particular cycle. There is a look ahead window with its own fetch and decoding logic. This window is used for instruction look ahead for out-of-order instruction issue. [202]



(a) A dual-pipeline, superscalar processor with four functional units in the execution stage and a lookahead window producing out-of-order issues

```
Program order
I1.  Load R1,  A    /R1←Memory(A)/
I2.  Add  R2,  R1   /R2←(R2)+(R1)/
I3.  Add  R3,  R4   /R3←(R3)+(R4)/
I4.  Mul  R4,  R5   /R4←(R4)*(R5)/
I5.  Comp R6        /R6←(R6)/
I6.  Mul  R6,  R7   /R6←(R6)*(R7)/
```



(b) A sample program and its dependence graph, where I2 and I3 share the adder and I4 and I6 share the same multiplier

Figure 6.28 A two-issue superscalar processor and a sample program for parallel execution.

Figure 75 example of a superscalar processor organization [202]

(a) What dependencies exist in the program?

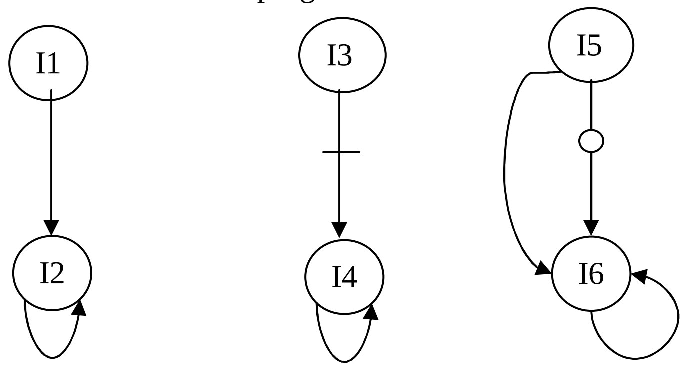


Figure 76 dependencies graph

From above graph, we know that true dependence exists between: I1 and I2, I5 and I6, and I2, I4, I6 which are self true dependence. Out dependence exits between I5 and I6. Anti dependence exits between I3 and I4.

(b) Show the pipeline activity for this program on the processor using in-order issue with in-order completion policies and using a presentation similar to the Figure.

The following table lists the detailed 5 pipelines activity for above instruction set. It will use 9 clock cycles to complete.

F		D		E							W		clock
I1	***I2***			***ld***	***log***	***m1***	***m2***	***m3***	***a1***	***a2***	***S1***	***S2***	***1***
I3	I4	I1	I2										2
I5	I6	I3	I4	I1									3
		I5	I6			I4			I2		I1		4
					I5		I4		I3	I2			5
						I6		I4		I3	I2		6
							I6				I3	I4	7
								I6			I5		8
											I6		9

From above table, we get the following points:

- I1 and I2 will start together in same clock cycle one and I1 will take 4 clock cycles (f1, d1, e2, s1). I2 has to stall one clock cycle at clock cycle 3 to wait for I1 to finish its execution part since I1 and I2 has true data dependency; that means I2 can't issue an instruction before I1 finish, and then it will proceed with the adder a1, a2 and s2.

- I3 and I4 will start on clock cycle 2. I3 will have to wait one clock cycle at clock 4 since I2 and I3 uses the same functional unit. After the wait, it will continue its operation while I4 does not need to compete for functional unit. So it will continue its operation without any stalling.

- I5 and I6 will start on clock 3. I5 will wait two clock cycles in order to allow I3 and I4 store its values since it can't finish storing before the previous instructions.

- On clock cycle 5, you can see it has more than 2 instructions in the execution units (a2, a1 and m2)

(c) Repeat for in-order issue with out-of-order completion.

Because of output dependence and anti dependence, the instruction set still uses 9 clock cycles with in order issue and out of order completion. But I5 doesn't need to wait for I3 and I4 completion because of in-order issue with out-order completion. The following table lists the detailed 5 pipelines activity.

F		D					E				W		clock
I1	***I2***			***ld***	***log***	***m1***	***m2***	***m3***	***a1***	***a2***	***S1***	***S2***	***1***
I3	I4	I1	I2										2
I5	I6	I3	I4	I1									3
		I5	I6			I4			I2		I1		4
					I5		I4		I3	I2			5
						I6		I4		I3	I2	I5	6
							I6				I3	I4	7
								I6					8
											I6		9

(d) Repeat for out-of-order issue with out-of-order completion.

	F			D					E				W	lock
I1	***I2***	***I5***				***ld***	***log***	***m1***	***m2***	***m3***	***a1***	***a2***	***S1***	***S2***	***1***
I3	I4		I1	I2	I5										2
I6			I3	I4		I1	I5								3
			I6					I4			I2		I1	I5	4
							I6		I4		I3	I2			5
								I6		I4		I3	I2	I5	6
									I6				I3	I4	7
													I6		8

	F			D					E				W	lock
I3	***I4***	***I5***				***ld***	***log***	***m1***	***m2***	***m3***	***a1***	***a2***	***S1***	***S2***	***1***

I6	I1			I3	I4	I5												2
I2				I6	I1				I5	I4			I3					3
				I2				I1		I6	I4			I3			I5	4
											I6	I4	I2			I3	I1	5
												I6		I2		I4		6
																I2	I6	7

With looking ahead window and out-of-order issue with out-of-order completion, the instruction set will use less clock cycle than in-order issue with in-order completion and in-order issue with out-of-order completion. The first case is that I1 and I2 are fetched first with I5 in the looking ahead window, total clock cycle is 8. The second case is that I3 and I4 are fetched first with I5 in the looking ahead window, total clock cycle is 7. The following two tables list the detailed 5 pipelines activity.

Summary

From this lecture, we review the concepts of superscalar processor architecture and pipeline, and learn how the dependence and instruction level paralleling work in the superscalar processor. We know how to use the graph to represent the dependence and the general procedure to solve the problem related to the superscalar processor.

Paper 1: The Microarchitecture of Superscalar Processors

Paper 2: Parallelism exploitation in superscalar Multiprocessing

Papers Analysis Section

Paper Analysis of Lecture #1a: NIOS II Architecture

The papers are going to be analyzed are:

- "System-on-chip: reuse and integration". The paper focus on the reuse and integration issues encountered when industry began to embrace new design and reuse methodologies.
- "Design Considerations for Soft Embedded Programmable Logic Cores". The paper focus on how integrated circuit designers synthesize a vendor supplied logical core using standard cells.

Paper 1 System-on-chip: reuse and integration

The flexibility of FPGA makes it possible to pre-configure and pre-verify hardware and software blocks. In the lecture, a conclusion drawn is that soft-core processors are more application specified than hard processors. This paper addresses on how to achieve even higher levels of productivity when design soft cores on system-on-chip (SoC) by using reusable IP cores and a methodology of integration. The paper introduces the history of SoP, reusability/designer productivity, challenges on integration, and testing/verification on the designs.

The authors first introduces that in order to handle increased complexity on very large-scaled integrated (VLSI) circuit design, engineers have developed new methodology and techniques to manage the challenge on the large chips. One of the answers is SoC design in which reusable intellectual property (IP) blocks are integrated. In the past, SoP meant that integrating functions on multi-chip system-on-board (SoB) into a single chip to reduce power, form factor, and overall cost. Nowadays, SoP mainly means the productivity gains through reusable design and integration of components. While SoP design is gaining increasingly more portion in the IC industry, it is not a direct competitor of system-in-package (SiP) and SoB since they have different tradeoffs in terms of "cost, power, testability, time-to-market, and packaging".

Reusable IPs requires more design effort than traditional ASIC design. Issues of designing reusable digital IP, analog IP, and programmable IP are introduced in detail. Digital IP are the most popular form of IP design; it includes 3 stages in design process: specification, implementation and full verification. The actual resulted digital IP are soft IP, hard IP, or firm IP blocks. Analog/mixed-signal (AMS) design is usually more time-consuming than digital design while more than half of the SoC design will involve analog/mixed-signal design. The productivity of AMS design can be improved using a mixed-signal SoC design flow employing AMS IP. AMS blocks are usually in the form of hard IP due to its sensitivity to environment, though firm IP are more appropriate and gaining popularity. Due to AMS design's nature, the IPs must be able to handle and transfer both design experience and heuristics form the original design to subsequent design derivatives. Programmable IP becomes more important while SoCs becomes larger and more complex. Prefabrication flexibility and postfabrication flexibility allow upper level modification during fabrication process and programmability after the chips are manufactured.

Platform based design process are introduced since reusable IP has its limit on gaining productivity (integration are time-consuming and expensive). Platform base design provides an abstract level of the design target so that new design can use the platform as a basis. In the new designs, many underlying detail are hided and can be quickly handle by some associated tools and flows. The method of the design was carried out at a university lab and resulted a reasonable design duration.

Communication between numerous IP blocks is another challenge since each block has its own separate synchronous boundaries. Asynchronous design may guarantee functionality while performance will be a tradeoff. Some special interfaces between the blocks have to be introduced for the system whose performance and latency are critical. Pipeline, parallel computing concept, streamlined and fast switches are used to realize network-on-chip (NoC).

Another important aspect of SoC design is the development of a test methodology. The SoC test are different from SoB test since all core block level and chip level test are conducted by the SoC designer; thus the complexity of the test is significantly increased. As the result, before SoC level test, IP core level test must be performed.

Verification challenges equally the SoC design and test since design size has grown exponentially and the target solution is through reuse. Reuse verification process is a goal of the effort. Simulation and emulation test, assertion-based verification, and some standardizing language are widely accepted by IC industry. Despite of the fact that research advances continuously, designer’s help for verification in early design process must be presented for better testability and verifiability.

As a summary, the paper explored the reuse and integration issues associated with digital, mixed-signal design, and programmable IP component. The platform based design concept, interconnect, testing, and verification were presented on the belief of the authors that reusable IPs and commercial tool will continue to advance to addressing system level design and verifications.

Paper 2 Design Considerations for Soft Embedded Programmable Logic Cores (PLC)

This paper focus on programmability for post-fabricated chips and is relates to field of reducing design cost for System-on-Chip. The idea is to amortize the cost of a single design to many related applications. PLC can be customized to implement any digital circuit after fabrication. Not many company provide PLCs due to lack of developing tools, even several company do provide PLC, the result are usually in fixed format and are not as efficient as hared-wired logic so that PLCs are far from mainstream. The authors of the paper believe the use of embedded programmable fabrics will continue to increase on both ASIC and SoC designs. Core vendors would provide soft descriptions of their PLC while the user could also develop the cores without much difficulty.

The author defined a soft PLC design flow. After designers receive an RTL description of the cores, they can use standard ASIC tools to implement the chips. The designers can also have the freedom to customize and position the blocks to better support target application. The disadvantage is larger area, higher power consumption and more speed overhead.

Two alternative architectures for a soft PLC were described. The first one is directional architecture where a small look-up-table (LUT) is used for efficiency and only one direction connection is allowed in a standard switch block. The other one is gradual architecture on which downstream LUTs have more possible inputs. The resulting architectures are the solutions of some observed problems.

The authors also utilize some algorithms to address placement and routing issues for after fabrication chips. The experimental results are presented and analyzed. The soft PLA has a 6.4X overhead that has been estimated compared to hard PLAs while the soft PLA is easier to migrate to new technology. The post-fabrication flexibility that the soft PLA cores provide will be vital as integrate circuits get larger and as masks get more expensive. The authors concluded that more work can be done to improve soft PLAs' area and speed; with new CAD tools are introduced, hard cores and soft cores will become equally important for future integrated circuits.

Paper Analysis of Lecture #1b: Introduction to Nios II Processor Architecture and Programming

Paper 1 Reconfigurable computing: architectures and design methods

The first survey paper analyzed in this report is “Reconfigurable computing: architectures and design methods” authored by T.J. Todman, G.A. Constantinides, S.J.E. Wilton, O. Mencer, W. Luk and P.Y.K. Cheung. It’s published in IEE Proceedings - Computers and Digital Techniques, Volume 152, Issue 2, page261 to 272, in Mar 2005. Since Stratix FPGA device and Quartus II integrated development environment (IDE), both from Altera, are chosen to be the physical and virtual environment for us to carry out all laboratory practice, an intensive study on architectures and design methods is suggested to all students of CEG4131 to enhance the understanding of the course context.

The survey paper is published in March of 2005. It is relatively up-to-date in the sense of technology which covers two state-of-art FPGA devices: Altera Stratix II – the next generation of what’s used in the course, and Xlinx Virtex 4 – the main competitor to Stratix II; and catches major matters in designing methods of general-purpose and special-purpose; at last, it discussed the main trends of this advancing technology.

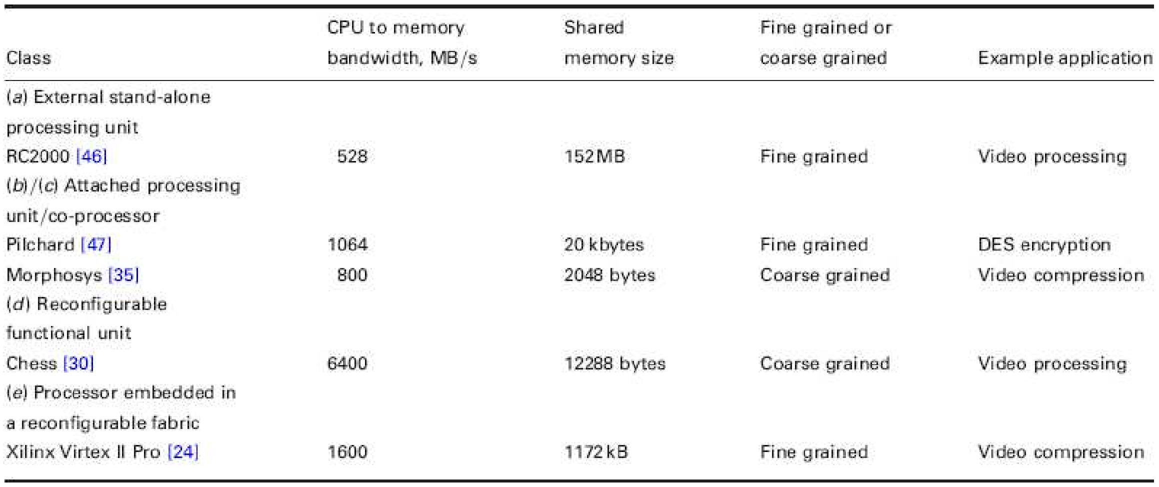
The paper starts with summarizing the advantages of implementing specific applications on today’s reconfigurable computing system, notably on the FPGA devices mentioned above. This topic is intensively discussed below in the purpose of helping students understand the importance of reconfigurable computing advantages, the reason that FPGA devices are heavily deployed in university’s laboratories, and its capability and potential within the scope of this course:

1. Reconfigurable computing system has achieved an impressive performance. Consider the point multiplication operation in elliptic curve cryptography described in the paper, a 66MHz FPGA system outperformed a 2.6 GHz dual-Xeon computer by 540 times faster [13]. Many applications today involved a soft-core processor such as the Altera NIOS II to improve performance. Soft core processors are specially designed for FPGA devices while allow customizing at application’s need for optimized performance.
2. Another compelling advantage is reconfigurable computing system reduces power consumption. Customized and optimized circuitry and moving critical software functions and loops to reconfigurable hardware results a 35% to 70% energy saving while achieving the speedup just described above [13].
3. Other advantages of reconfigurable computing system include reducing the cost by reducing average size and component count, faster prototyping, reducing lab-to-market cycle, and increasing product’s life-cycle by hardware’s flexibility and upgradability.

In contrast, general-purpose microprocessor is a generic hardware with high cost, high power consumption and low performance in many specific applications. It doesn’t meet many requirements in today’s mobile and battery powered computing envirnment. Another competitor to reconfigurable computing system is circuit application-specific integrated circuit - ASIC. It outperformed general-purpose microprocessor and reconfigurable computing device in the terms of efficiency and energy saving. It has "just the right mix of functions", described by the author. But it's very-high-cost to develop, therefore is only suitable for very high volume applications.

These advantages are evidenced more apparently in today's embedded system and its market, especially after the soft-core processors be introduced. With these advantages, reconfigurable computing system, such as Altera FPGA devices and its integrated development environment formed a low-cost, high performance, multi functioning and high fault tolerance solution to academic purpose. In previous year courses, such as CEG3150 and CEG3151, from microcontrollers to simple MIPS microprocessor are implemented on Altera FPGA system. These laboratory work largely helped to transform theory to practice, and further more shaping the view of today's computer engineering.

The architecture of Altera Stratix FPGA device is briefly introduced in lecture slide. The survey paper offered a more intensive view on this topic through system-level architecture, reconfigurable fabrication (logic element and logic array block) and new trends of reconfigurable computing. This topic will be briefly summarized in the table below because it's not heavily related to the course.



Class	CPU to memory bandwidth, MB/s	Shared memory size	Fine grained or coarse grained	Example application
(*a*) External stand-alone processing unit				
RC2000 [46]	528	152 MB	Fine grained	Video processing
(*b*)/(*c*) Attached processing unit/co-processor				
Pilchard [47]	1064	20 kbytes	Fine grained	DES encryption
Morphosys [35]	800	2048 bytes	Coarse grained	Video compression
(*d*) Reconfigurable functional unit				
Chess [30]	6400	12288 bytes	Coarse grained	Video processing
(*e*) Processor embedded in a reconfigurable fabric				
Xilinx Virtex II Pro [24]	1600	1172 kB	Fine grained	Video compression

Table 1: Major reconfigurable computing device architectures at system level [13]

Altera NIOS II is a soft-core processor "embedded" in FPGA device. It's in the 4th type in the survey paper. Through this course of studying parallel processing technology, students will be asked to reconfigure NIOS II to meet certain system requirement. It's taking the solo advantage of reconfigurable computing – flexibility on hardware level.

Another big topic covered in the survey paper is about design methods, which subdivided into general-purpose design and special-purpose design. It certainly helps for many students shaping their projects on FPGA devices, but again it's not largely relevant to the course.

In modern general-purpose programming language such as C or C++, hardware support libraries are facilitated to gain efficiency of prototyping and adoption of hardware/software interface. Two approaches are discussed: the annotation and constraint-driven approach and the source-directed compilation approach. The difference is how the high-level language is translated and preserved. Reconfigurable computing systems are developed at the rising trend of special-purpose design matters. The paper mainly discussed digital signal processing (DSP) related design on FPGA devices. Other design approaches, such as Runtime customization, multi-FPGA compilation is briefly discussed.

At the end of the design related section, an interesting topic about design matters with soft-core processors such as NIOS II is briefly covered. Soft-core processors are customized to support customized instructions which have two major benefits: First, they reduce the time for instruction fetch and decode cycle, and each custom instruction can combine several regular instructions. Second, additional resources can be assigned to a customized processor and instruction to improve performance. And these are the platform and methods to be used in this course to develop our parallel processors.

At last, main trends of reconfigurable computing system are outlined in the paper: The main trends in architectures are coarse-grained fabrics, heterogeneous functions and soft core processors. The main trends in design methods are special-purpose design methods, low-power techniques and high-level transformations.

At the end of this report, we conclude that this is a well written and formatted survey on today's FPGA technology and reconfigurable computing related matters. It is one step beyond the first lecture, and is giving enough amount of description to answer one important question in this course: Why we are using a FPGA device with its soft-core processor when studying computer architecture. The answer in one sentence is: it allows you to implement the processor we want with limited time and knowledge on the hardware we can afford.

Paper 2 Network-on-chip architectures and design methods

The second survey paper analyzed in this report is "Network-on-chip architectures and design methods" authored by L. Benini and D. Bertozzi. It's published on IEE Proceedings - Computers and Digital Techniques, Volume 152, Issue 2, page193 to 207, in Mar 2005. System-on-chip (SoC) can be simply understood as a computer system constructed on a single silicon chip. We have already constructed a simple computer system on an Altera Cyclone FPGA device in course CEG3151, control and data buses were employed to interconnect different blocks – the intellectual property (IP) cores. In this survey paper, new on-chip communication architecture Network-on-chip (NoC) and its recent development from existing SoC are introduced, and the difference and evolution from SoC is gradually analyzed. This paper is not as close to the course as the previous one was????? . As the matter of fact, it's quite beyond the infrastructure we are dealing with today. But we do believe this is a very good opportunity to introduce some fresh concepts to student what they'll deal with tomorrow as a computer engineer. Therefore it's highly recommended to all fellows who are taking the computer system approach of studying.

Bus was a communication path originally between computer components within computer or between computers. With the same set of wiring, devices are logically connected in a time multiplexing fashion. Today's computer system can be implemented on a single silicon chip. Instead of connecting devices, bus is connecting IP cores. In the FPGA devices introduced both in above article and in the first lecture, buses exist among Logic Elements, Logic Array Blocks, Memories and communication ports to the outside world. Today's multimedia application require on-chip buses to achieve higher bandwidth on top of mobility and energy saving. Traditional bus technology and topology suffered the limitation of scalability. Besides scalability, poor decoupling between IP cores and buses is a rising issue. Three main trends of evolution are developed to address these problems: 1) enhance parallelism; 2) enhance topology; 3) redefine and standardize communication interface. A case study on the evolution of AMBA bus is introduced in detail includes AMBA AHB and AMBA AXI. We are skipping these architectures here and jumping directly to more advanced and interesting NoC architecture.

Noc architecture is a packet-switched network among IP cores on a single chip. Like similar to a modern telecommunications network, using digital packet switching over multiplexed links, NoC is constructed from multiple point-to-point data links interconnected by switches (or routers), such that messages can be relayed from any source core to any destination core over several links, by making routing decisions at the switches (or routers). Three basic building blocks are the links, switches (or routers) and network interface associated with IP cores. Each component will be briefly discussed in this report.

Since the wiring in the modern SoC could be very long between cores, in most cases data is unable to reach the destination within one clock cycle. Such interconnect delay is popularly solved by implementing pipelines. Early NoC pipeline such as Xpipe is discussed in the survey paper. As we had a simple concept on pipeline in course CEG3151, it is suggested for all students to go through this part of the article.

NoC switches (or routers) design involves guaranteed throughput and best effort services concept and many other network issues such as flow control and error detection. Early NoC switched network including Aethereal architecture is discussed. We can relate many concepts from course CEG3180 when reading this part.

Network interface is designed with several critical tasks: 1) providing a standardized set of Point-to-Point transactions to cores; 2) efficient mapping of PP transactions into a possibly large set of network transactions; and 3) interfacing with the packet-based network fabric.

With the revolutionary change at hard-ware level, a change in design is mandatory to accompany the new environment for the optimized result. The key design challenge is to implement task-level parallelism and to formally capture concurrent communication in the new model.

Even the NoC architecture is still in research level, and it's far beyond the scope of this course, the concepts behind, such as packet-switching, is already familiar, and parallelism is the major concept we are going to learn in this course.

Survey Paper 1 and 2 Recommendations

We highly recommend above survey papers to all students and people who are take computer system as their approach as a computer engineer, especially who will involve graduate study in near future. We entrust that by taking a snap on the coming generation of computing system, we will be enlightened and guided by those invisible trends of advancing technology, no matter it's a mature technology with greater potential such as the reconfigurable computing system described in survey paper 1, or the fresh new architecture that is only in research described in survey paper 2.

Paper Analysis of Lecture #3: Cache Memory

Paper 1 The Processor-Memory Bottleneck

Problems and Solutions
by Nihar R. Mahapatra and Balakrishna Venkatrao

The Problem

The premise of the research paper outlines the problems and solutions associated with today's rapid growth of processor speeds compared to the growth of memory (DRAM specifically). In short-term comparison, the improvement of processor speeds shadow the respective increasing speeds of cache memory. As a result, the potential of computer system speeds no longer grow linearly due to this stunted relationship. In the past, the advances in processor-memory technology were the primary factor in computer system growth. No longer can this be depended on, and this truth will continue to impose a problem in future generations as well.

While the situation is evident now, the long-term effects include a Processor-Memory performance gap that will increase over time as the current trends continue. Should this occur, the obstacle would become impossibly difficult to overcome.

The division of the computer hardware industry contributes to the main cause of the growing gap between speeds. Microprocessor performance is increasing by 60% every year, while DRAM performance is only improving by 10% every year. As a direct result, the disparity between the speeds grows exponentially with every passing year.

In regards to the mechanics of the system, if a cache memory request misses, it will still take many cycles to fulfill. Therefore, higher processor speeds depend heavily on cache memory anyway to achieve above-standard performance rates. The system's performance becomes hindered by the request handling quality of the DRAM.

The Solution

The primary concept behind improving the relationship between microprocessor and cache memory is minimizing memory access time. As mentioned in CEG 4131, the success of memory performance depends on three factors: the request hit rate, miss rate, and miss penalty. The ability to improve such speeds is achieved by increasing either of these attributes.

The most practical method of increasing memory speeds is improving the hardware. In short, increasing the internal clock or data bus width will achieve the desired result. However, in modern technology we are approaching our limit in DRAM hardware upgrades (defined roughly by the 10% per year).

A larger on-chip cache will reduce the request miss rate since fewer requests are made from the system. This is similar to the previous method of improving memory performance in that the system's cost will still increase at the same rate.

A well received theoretical method involves dynamic access ordering. This means rearranging the order of requests received. The software component of the system (the compiler specifically) arranges the request stream for the processor to the sent through the SMC, which in turn dynamically organizes them into a sequence that maximizes bandwidth efficiency.

Another method involves a new memory architecture known as the Impulse Project, which improves the existing DRAM capabilities and efficiency. Impulse can be optimized for specific uses through physical address remapping. Corresponding applications can then configure Impulse to access cache differently. Also, Impulse can "prefetch" data at the memory controller. As a result, DRAM access is improved dramatically.

Relevance to Computer Architecture

The concept of the processor-memory gap is perhaps one of the most heated debates regarding the advancement of computer technology. Cache memory is an essential level of computer architecture, and as the course dictates, every level is dependant on every other level. The relationship between them are indirectly influenced by the other.

The basics of cache memory lie in the hit and miss rates when requests are made. The article discusses the optimization of cache memory through its basic qualities. How we store information in cache affects how fast we can retrieve information regarding clock cycles and how effective DRAM can synchronize with the microprocessor.

Paper 2 Cache Performance Analysis of Algorithms

By James. D Fix

Summary

The premise of the paper describes techniques to analysis cache performance of algorithms effectively. To measure algorithms accurately, a realistic cache memory model is needed. In addition, the analysis must adhere to constant factors related to hit and miss rates of cache requests.

A runtime analysis tool such as Atom can be used to confirm and verify the accuracy of the methods used. Different algorithms are developed based on associativity of block cache placement. Distinct patterns will usually outline the characteristics of one associative/algorithmic style, which lends itself to the second objective of the research: finding better algorithms. One such method is the combination of different patterns to optimize runtime, however that in itself is very complicated and unpredictable. An example of combined patterns is permutation traversals with a random access element.

Another important issue brought up is the overall design of computer systems and its flaws in regards to lack of consideration for cache memory when algorithms are implemented. As a result, many modern systems do not effectively use cache memory and end up hindering the performance of the system at large.

Relevance to Computer Architecture

In the course material, we study the different associative properties of direct mapping, full association and 2-way association. The implementation of cache memory algorithms is similar to the usages of each association. Through the different “patterns” that can be analyzed, we can determine the best implementation of cache memory through a combination of one or more patterns.

The study of cache performance serves the important purpose of calibrating how effective an algorithm is handling requests. The efficiency of an algorithm, theoretically, depends on its runtime. In modern computer systems, much of the runtime is spent transferring requests and information through cache memory. While an algorithm can be optimized by efficient code, the cache performance will dominate the equation since every transaction will be handled as a request. The hit and miss rate will determine how fast any algorithm will traverse each step.

Cache memory is a basic in computer architecture design and must be managed well for every design level above it to be optimized.

Paper Analysis of Lecture #4: Performance Analysis of Multiprocessor Architectures

Paper 1 Cascaded Execution: Speeding Up Unparallelized Execution on Shared-Memory Multiprocessors [23]

Introduction

According to Amdahl's law, the speed up of any system will be directly related to the time it takes to execute the sequential part of the code. Amdahl's law simply stated suggest that while increasing the number of processors for a given system the execution time for the parallel section of the code would approach a negligible execution time. Meaning the total execution time would be equal to the execution time of the serial section. The question arises what to do if your code doesn't have any parallel section or if there is too little parallelization in the code, is it possible to speed it up and obtain the results you want?

Analysis

This paper deals with how to efficiently execute loops which are done sequentially by reducing the load on one processor with one cache and putting this load on to 3 other processors which are sitting idle while one processor did all the work. Theoretically this would show speed up because it would drastically reduce the number of cache miss, and miss penalties resulting in a better execution time. Figure 1 below describes the manner in which the loop is divided and executed among the other processors.

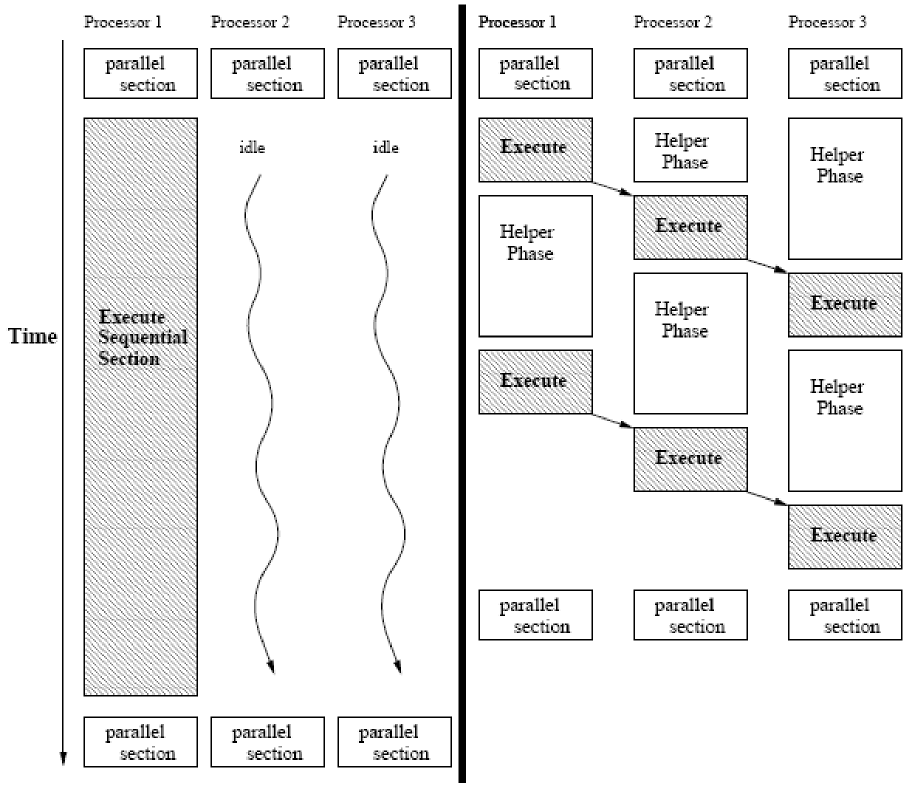
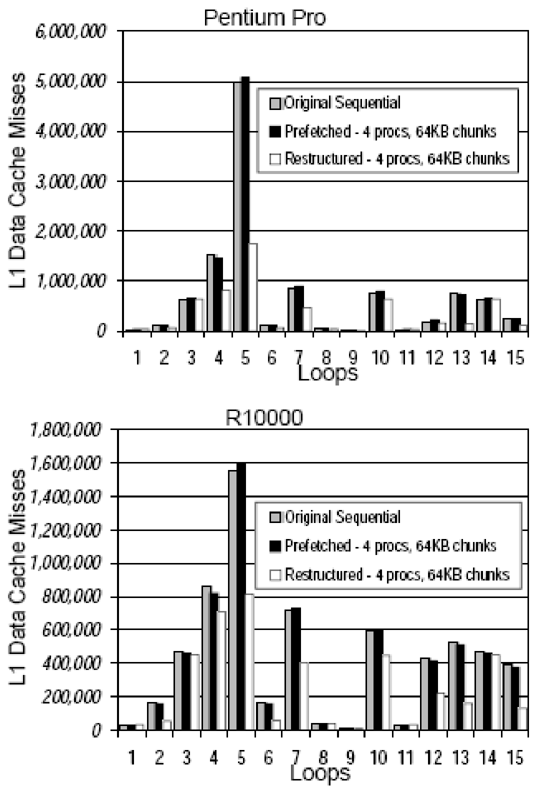


Figure 1 Standard Execution Vs Cascaded Execution

On the left hand side of figure 1 you can see that while processor 1 completes the task the other two processors are sitting idle which is not very efficient. On the right hand side of figure 1 you may notice that for the sequential part from before (left) it is now split among the 3 processor in a sequential manner. The reason why it is still in a sequential manner is because this part of the code couldn’t be implemented in parallel in the first place, but by forcing the execution to be done in a parallel manner each processor can use it’s cache fully to decrease the number of cache misses that would have been encountered on only one processor and one cache.

Testing these ideas with an experiment would give the following results.



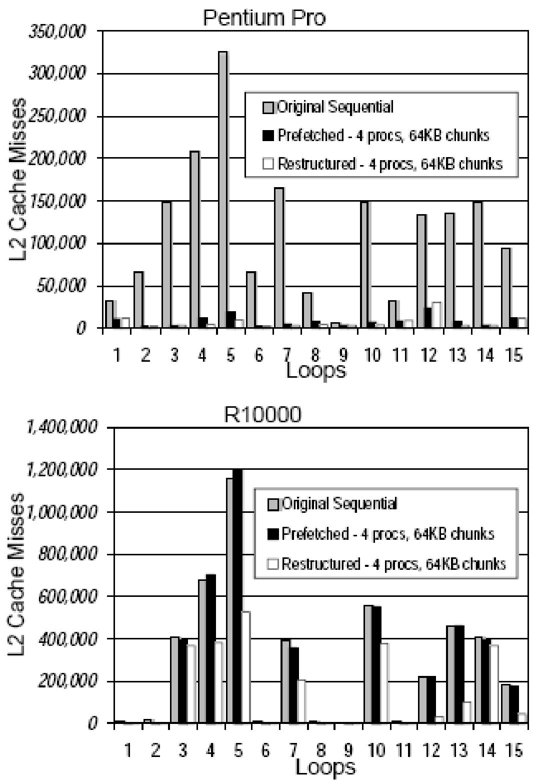


Figure 2 L1 Data Cache Misses Figure 3 L2 Cache Misses

Figures 2 and 3, above, compare L1 and L2 cache misses from the execution of the same algorithm but on two different machines. First execution is with a single processor followed by a 4 processor system. In both figures the original execution is depicted in light grey (left most bar), then by the prefetched bar in black, and lastly with the restructured bar in white.

Clearly it can be deduced from figures 2 and 3 that there is an improvement with the use of cascaded systems. It should be noted that these systems were tested using wave5 which is a Spec95fp benchmark application, and another synthetic benchmark.

The results of cascaded execution is as follows.
"We evaluate cascaded execution using loop nests from wave5, a Spec95fp benchmark application, and a synthetic benchmark. Running on a PC with 4 Pentium Pro processors and an SGI Power Onyx with 8 R10000 processors, we observe an overall speedup of 1.35 and 1.7, respectively, for the wave5 loops we examined, and speedups as high as 4.5 for individual loops. Our extrapolated results using the synthetic benchmark show a potential for speedups as large as 16 on future machines."[23]

Conclusion

I believe this paper is relevant to all the areas that were covered in the lecture but it shows a different aspect. In class the algorithm in question was completely separated between serial and parallel, and it was accepted that the speed up of the system would be directly proportional to the serial execution time. As this paper shows that there are sequential parts of the code that can be executed on multiprocessor system to give better results than to just run on a single processor while the others sit idle. All of the concepts that were covered in the lecture are present in this paper. Everything from Speedup to efficiency and even benchmarks are used to evaluate this system. The novelty of this idea is of great use because there are a great many of system that cannot be run in parallel and for those system there is a way to get increase speed up and efficiency with cascaded execution. Using the knowledge of speedup, efficiency, scalability, parallelism, amdahl's law, and benchmarks I was able to find this paper that shows all of these concepts used towards a new and different idea that wasn't covered in class.

Paper 2 The Consequences of Fixed Time Performance Measurement [24]

Introduction

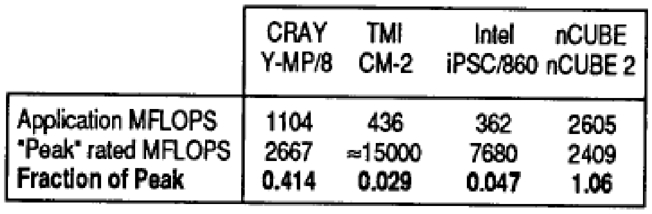
Fixed sized performance measurement uses a defined set of work and calculates the time required to complete that work. From this, we can calculate things like speedup and efficiency. The use of parallel computers is primarily for speed and speedup usually relates to time reduction. Consider the following example:

Major Premise: Sixty men can do a piece of work sixty times as quickly as one man.
Minor Premise: One man can dig a post-hole in sixty seconds.
Conclusion: Sixty men can dig a post-hole in one second.
The reasoning in this example is flawed until the problem is scaled to sixty post-holes. In this way, the sixty men are sixty times as productive. So instead, let us measure the work done in a fixed time, instead of the time reduction for a fixed problem.

Analysis

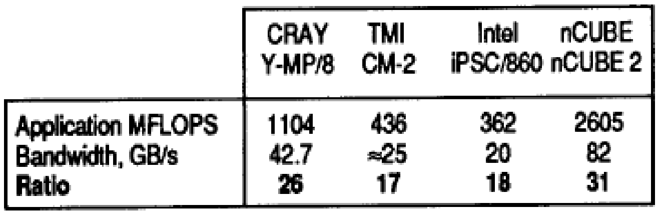
This paper proposes that performance measurement should be done using fixed time methods; the measurement of the amount of work done in a fixed time. The author first proposes that the "work" is not the operation count. Arithmetic on modern computers is completed so quickly that they are limited by their interconnections. This relates back to our lecture where communication overhead is taken into account when calculating speedup.

The author next proposes that the peak memory bandwidth is a better performance indicator than peak MFLOPS.



	CRAY Y-MP/8	TMI CM-2	Intel iPSC/860	nCUBE nCUBE 2
Application MFLOPS	1104	436	362	2605
"Peak" rated MFLOPS	2667	≈15000	7680	2409
Fraction of Peak	**0.414**	**0.029**	**0.047**	**1.06**

Figure 77: Peak and Application MFLOPS



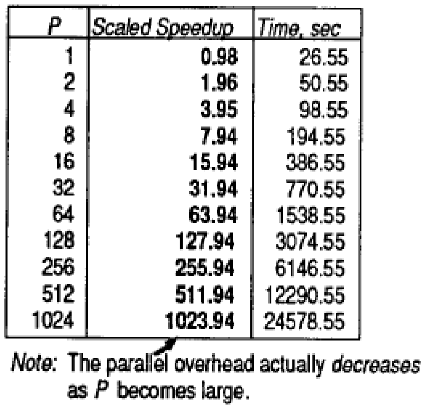
	CRAY Y-MP/8	TMI CM-2	Intel iPSC/860	nCUBE nCUBE 2
Application MFLOPS	1104	436	362	2605
Bandwidth, GB/s	42.7	≈25	20	82
Ratio	**26**	**17**	**18**	**31**

Figure 78: Application MFLOPS and Bandwidth

Comparing the ratios achieved in Figure 9 and Figure 10, we see that bandwidth is a much better indicator for performance. This reinforces how important the interconnection networks are.

The standard definition for speedup does not require a rigorous definition of work. As noted, using operation count as a measure of work performed is imperfect as it is not standard across different architectures. A "standard computer" is chosen and we define its execution time as the work.

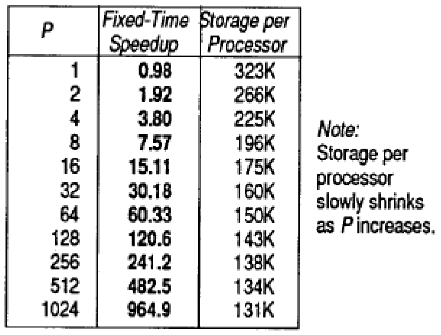
The author extends the formulas for speedup and efficiency to include work as a variable. Using time as fixed, we can calculate the fixed time speedup.



P	*Scaled Speedup*	*Time, sec*
1	0.98	26.55
2	1.96	50.55
4	3.95	98.55
8	7.94	194.55
16	15.94	386.55
32	31.94	770.55
64	63.94	1538.55
128	127.94	3074.55
256	255.94	6146.55
512	511.94	12290.55
1024	1023.94	24578.55

Note: The parallel overhead actually *decreases* as *P* becomes large.

Figure 79: Scaled fixed sized speedup



P	*Fixed-Time Speedup*	*Storage per Processor*
1	**0.98**	323K
2	**1.92**	266K
4	**3.80**	225K
8	**7.57**	196K
16	**15.11**	175K
32	**30.18**	160K
64	**60.33**	150K
128	**120.6**	143K
256	**241.2**	138K
512	**482.5**	134K
1024	**964.9**	131K

Note:
Storage per processor slowly shrinks as *P* increases.

Figure 80: Scaled fixed time speedup

Figure 11 and Figure 12 show the difference in calculating speedup using fixed sized and fixed time methods.

Conclusion

This paper seemed to be only slightly relevant to the topics studied in class. While it was interesting to see a different way of measuring performance, until it is universally accepted there is no standard.

Paper Analysis of Lecture #5: Parallel Computer Models

Paper 1 A Survey of Parallel Computer Architectures

by R. Duncan
Computer, February 1990
This survey paper discusses the innovations of computer architectures in the early nineties in comparisons to older architectures. The main approach of this paper is to show the alternatives to parallel processing of a high-level taxonomy, in which its main topic will be covering the most common of Flynn's taxonomy, MIMD. In order to address this topic, the authors will first try to explain how the principal types of architectures work through fundamental organizing principals.

This paper is directly relevant to the lecture in multiple ways. It first talks about general taxonomy and terminology of architecture design, from Flynn's taxonomy to parallelism, and, as well, it touches base on the different types of interconnection networks within different architectures for message passing. All main concepts used within this paper were talked about in the lecture.

The authors then discuss how a possible combination of architectures could generate further research, especially for its flexibility. For example, how a part of the MIMD architecture could be controlled like a SIMD approach. This amalgamation would be advantageous in parallel image processing and expert system applications.

The paper then goes on to discuss novelties from the lecture: dataflow, reduction and wave-front array architectures. This first architecture contrasts the traditional control-flow architecture. The execution of instruction is determined based on availability of input arguments to the instructions. The reduction architectures use a model where an instruction is enabled when its results are required for another instruction already executed. And for the last architecture, wave-front, it combines systolic data pipelining with an asynchronous dataflow execution archetype: the projection phase and post-projection phase are assigned to different processors and a significant time-saving can be achieved by the pipelining technique. These techniques are simply other types of possible architectures that have been more of use for research purpose and have not been exploited in any major computer architectures. These novel approaches simply show the possibility of other types of architectures that can exist.

Paper 2 Performance of Multiprocessor Interconnection Networks

by Laxmi N. Bhuyan, Qing Yang, and Dharma P.Agrawal
Computer, February 1989
This article concentrates on interconnection networks in parallel computer architectures. It has been written at the time when the parallel computers became a prominent research topic in computer development. The cost and limitations of VLSI designs were started to be realized and new alternatives to increase computer performance were considered. The article mainly concentrated on the way to analyze different interconnection networks using analytical models. At that time, the hardware or software simulation of various interconnection networks was too expensive and time-consuming to be considered efficient.

The authors of the paper start with outlining different multi-processor or multi-computer architectures. Their description is closely related to the classification introduced in the lecture. For example, the authors use the term 'multiprocessor' to describe a shared-memory system and the term 'multi-computer' to describe a message-passing system where each processor has a local memory.

As mentioned previously, the main topic of the paper is interconnection networks. After introducing the readers to parallel architectures, the authors give a classification of networks. The classification of the interconnection networks is related to the one given during the lecture. There is a difference worth mentioning. The authors concentrate on, what we now call, dynamic networks. It seems that the static networks, such as ring or star, were viewed as being useful in multi-computers (i.e. message-passing systems). The classification of the interconnected networks given in the article does not include division based on static or dynamic connections. Nowadays, dynamic networks are used in both multiprocessors and distributed systems. The description of networks based on timing (mode of operation), switching technique and control strategy is identical to the material presented in the lecture.

The lecture did not go into details on the analysis of various interconnection networks. The discussion of performance and cost of different networks were left for future discussions. The performance to cost analysis of dynamic interconnection networks is the topic of this article. Some of the parameters used to analyze the networks we have studied before, for example bandwidth and throughput. Others were less familiar to us, such as probability of acceptance and processor utilization.

As the performance of dynamic interconnection networks was not a topic of the lecture, we will not discuss the performance analysis given in the article.

The most relative parts of the article are the introduction to parallel computers and classification given to interconnected networks. Besides knowing and understanding the basis of network classification, it is also important to realize that different models are possible and the choice of the model depends heavily on the application. The performance analysis given in the article shows both advantages and disadvantages of various network architectures. For example, cross-bar network is the most expensive to implements (i.e. it requires N×M switches, comparing to NlogN required for multistage IN [N – number of processors, M – number of memory units]); yet, it is also the most time efficient. Besides being the most expensive, the cross-bar is not easy to expand and it is not the best in the fault-tolerance. Thus, the designers should consider different models and architectures for the implementation.

Although, the article was written more than 15 years ago, the concepts outlined are still valid today. The capabilities of the computers have greatly increased, and simulations are not that time-consuming and expensive as they used to be. Yet, the techniques to perform analytical modeling outlined in the article are still useful today as they provide a way to produce quick performance approximations. It is importance to note that the article mostly concentrates on analyzing synchronous networks. Asynchronous networks are harder to analyze and design, yet they are starting to become more important, especially with the increased development of distributed systems.

The article concludes with possible research topics in the parallel architectures and especially interconnection networks. Some of these research topics have been taken and during the lecture we did discuss some of them. For example, pre-fetching and cache techniques to speed up the communication.

Paper Analysis of Lecture #6a: Dynamic Interconnection Networks - Buses

Paper 1 Design and Analysis of Arbitration Protocols

Description of Paper

This focus of this paper is on the subject of arbitration protocols. Rather than providing an in-depth analysis of all possible arbitration protocols, Guibaly chooses to outline five of the major protocols in use by industry and those being studied in literature of the time. The five protocols examined include:

1. Equal-priority protocol
2. Unequal-priority protocol
3. Rotating-priority protocol (round robin)
4. Random-delay protocol (non-persistent CSMA)
5. Queuing protocol (FIFO)

Several areas are considered when discussing these protocols. Firstly, a summary is provided outlining the operation of all five protocols. Secondly, hardware implementations for four arbitration protocols are given. Thirdly, a mathematical analysis is given to show the performance of all five protocols. For the purposes of this paper, performance is measured by analyzing the time to access the bus and the data throughput. Finally, Guibaly shows the simulation results of the protocols. The author concludes by showing all protocols are suitable for multiprocessor systems with the exception of the unequal-priority protocol. Overall, Guibaly shows that the rotating-priority protocol is the optimal arbitration protocol. [45]

Relevance to Lecture

This paper is relevant to our lecture for several reasons. Our lecture was focused on discussing bus arbitration and some of the factors that must be considered when choosing a protocol. The focus of this paper was to provide a better understanding of how these protocols operate and how they could be implemented. Although individual systems will require different solutions, this paper concludes by providing a basis for choosing a suitable arbitration protocol more almost any situation. **[45]**

Similarity to Lecture

The lecture provided us with a brief introduction to several arbitration methods. This paper provided a more thorough explanation of how some of these protocols operate. Rather than summarizing the advantages and disadvantages of each protocol, the paper provides an in depth analysis as to why some protocols may be better suited for a particular system. [45]

Our lecture focused on defining three classes of arbitration: daisy chain arbitration, arbitration with independent requests and grants, and distributed arbitration. The paper discussed the unequal arbitration protocol, where each processor is assigned a unique priority. When a request is made, the grant is propagated across the priority bus and the highest priority peripheral is granted access. Thus, the issue fairness is disregarded, as low priority peripherals may never be granted access. This is similar to the daisy chain arbitration method discussed in class. [45]

Within the independent requests and grants class, our lecture discussed round robin and TDMA protocols, identical to the rotating-priority protocol and random-delay protocol methods discussed in the paper, respectively. [45]

Before looking at the actual performance of these protocols, the author first shows that all protocols do not necessarily consider priority and fairness, the two factors we discussed in class. [45]

Differences from Lecture

There are also differences from the material found in the paper and that of our lecture. Most notably, the paper offered a rigorous mathematical analysis for demonstrating the benefits of one protocol over another. This analysis was concerned with both the access time and the data throughput. Also, graphs were providing to illustrate how access time changed as more processors were added for each protocol. The paper also offered schematic diagrams to illustrate how these protocols could be implemented in hardware. In addition, the paper discusses one additional technique not covered in the lecture, the queuing protocol. In this protocol, the highest priority is assigned to the request that arrived earliest. Therefore, this protocol does not consider priority, since it is uniform across all peripherals. [45]

Paper 2 Architecture of a System on a Chip with User-Configurable System Logic

Description of Paper

The focus of this paper is to present a custom bus architecture (CBA) targeted for a system on a chip (SOC) that is combined with user-configurable system logic (CSL). These SOCs with CSL require a high and predictable performance bus that is able to scale easily with the system, and provide a simple interface in order to allow existing peripherals to be connected with minimal redesign. [46]

The paper outlines the preliminary constraints considered and addressed when designing the CBA. The design of the CBA is then described in the following sections: overall bus architecture; bus transactions; bus logic; and bus architecture within the CSL. Integration issues with selected existing systems is addressed, as some of the initial constraints dealt with the integration of this CBA into existing systems, as well as physical implementation of the system. The paper then concludes with a comparison of the CBA presented against other SOC buses. [46]

Relevance to Lecture

Understanding the design basics of the CBA presented in this paper allows us to compare a layout of bus components and logic different from those we have looked at previously. Concepts such as arbitration, timing, communication schemes and techniques were all applied when examining Altera's Avalon bus, and are all presented as major aspects of the CBA presented in the paper. In doing this, the paper also allows us to see how a different hardware specification brings about a difference in the application of these base concepts. In the paper summary to follow, I attempt to highlight the major similarities and differences between the Avalon bus and the CBA presented here. [46]

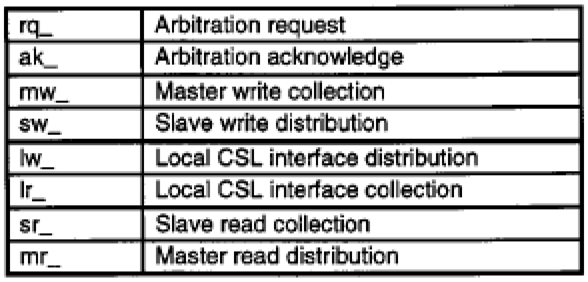
In addition to this, the CBA has high and predictable performance requirements, and thus the paper delves into the relationship between timing and sub-module hardware layout. This gives an added perspective of how data flow as time passes through bus registers and other hardware in order to accomplish predictable performance. [46]

The paper does however delve into aspects of the bus that do not have strong relevance to the material covered in our lecture. The main example of this is the extension of the bus architecture into the CSL module within the system. The paper discusses at length how the CBA is scaled into a CSL module, through distribution of its components and logic. Other non-relevant topics are DMA specifications met by the bus and physical implementation of the CSL on a die. [46]

CBA versus Avalon

As mentioned, the CBA in this paper is targeted towards SOCs with CSL. In order to create a bus architecture to work specifically with such systems, design constraints must be taken into account, impacting the final design such that it differs from other bus architectures. [46]

As user CSL has conventions for its logic and routing, one of the main constraints in designing new bus architecture was a simple interface to minimize impact on the design of other peripherals to be connected to it. For such a highly configurable and adaptable bus architecture, the number of signal groups within the bus is small, with the division between them clear and logical. Any device need only consider the signal groups applicable to it. The signal groups are shown in Figure 18. [46]



rq_	Arbitration request
ak_	Arbitration acknowledge
mw_	Master write collection
sw_	Slave write distribution
lw_	Local CSL interface distribution
lr_	Local CSL interface collection
sr_	Slave read collection
mr_	Master read distribution

Figure 18: Bus segments and signal name prefixes [46]

Another constraint in designing the CBA was the necessity to adapt the architecture to different processors with different bit-lengths. This was accomplished through the use of sub-word replication and merging when changing widths on the bus, and a memory management unit to map addresses from processors with smaller addresses spaces to the bus's larger address space. This is reminiscent of the addressing abstraction provided by Altera's decoding logic, which handles a single address, extracts the chip selection and addressing and provides the necessary addressing to the chip itself. [46]

The CBA also requires predictable high-performance, handling both single cycle transfers and variable peripheral response time. This is managed through the use of pipelining, as seen in the Avalon bus, yet goes beyond this by splitting the CBA to have separate read and write data lines to increase bandwidth. Variable peripheral response time is handled through the insertion of wait states, which is similar to the Avalon bus. [46]

In terms of architecture, there are other comparisons that can be made between the CBA and Avalon bus.

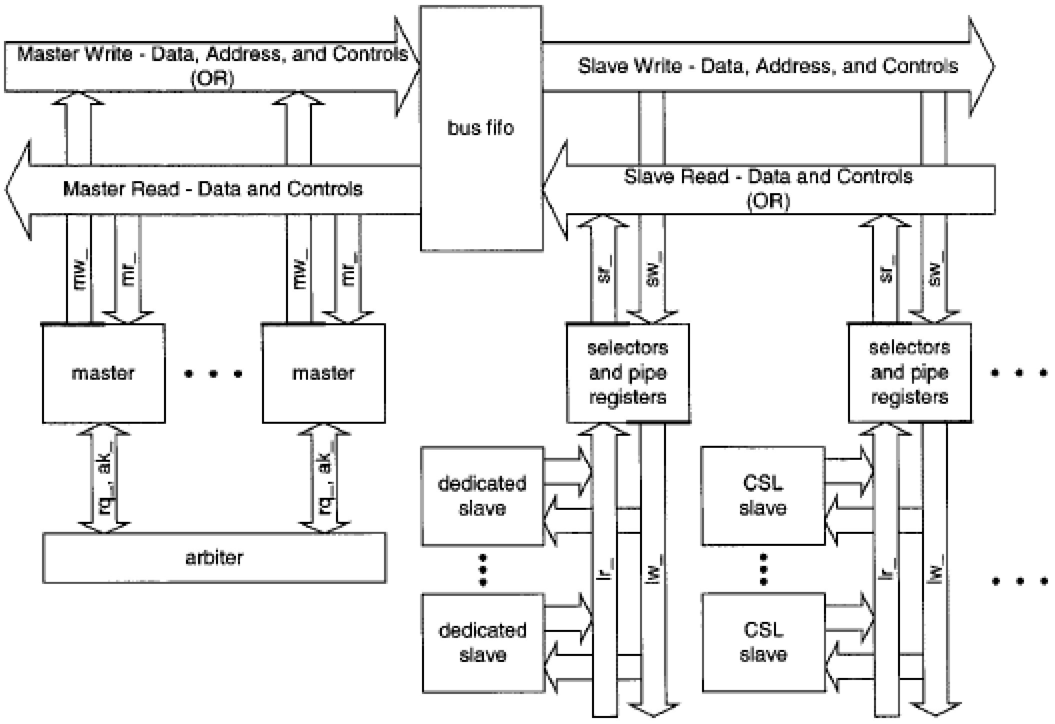

Master Write - Data, Address, and Controls (OR)
bus fifo
Slave Write - Data, Address, and Controls
Master Read - Data and Controls
Slave Read - Data and Controls (OR)
mw_
mr_
sr_
sw_
master
master
selectors and pipe registers
selectors and pipe registers
rq_, ak_
arbiter
dedicated slave
dedicated slave
CSL slave
CSL slave
lr_
lw_

Figure 19: Bus architecture and segments [46]

As can be seen in Figure 19, one of the major differences between the two buses is the placement of the arbitration unit. Arbitration is done on the master side in the CBA, as apposed to the slave-side configuration in Avalon. [40] This configuration requires all masters requesting the bus to place an arbitration request, which may or may not be granted in the same cycle. In terms of arbitration algorithm, any algorithm can be implemented for granting access to masters. [46]

A similarity between the two bus architectures is the pipelining of the bus. However, they differ in the extent to which masters are aware of response time. In the Avalon bus, variable response time support must be indicated by the master. [41]In this CBA, the bus consists of a first-in first-out (FIFO) stage that abstracts the communicating master from being aware of the insertion of wait states. Instead, the bus FIFO tracks the communications over the bus pipe, and signals to the appropriate master when a read has been completed by the peripheral. [46]

Paper Analysis of Lecture #6b: Dynamic Interconnection Networks: Buses

Paper 1[49]Performance of multiprocessor interconnection networks

Short Description of the Paper

This journal from 1989 is speculating on analytical ways to model interconnection networks performance. The major reason of their research is to avoid the high costs of interconnection network simulations using hardware or software implementations, to facilitate performance evaluations and ease the design and implementation phases. To do so, they explain the basics of three kinds of interconnection networks so we can understand what has to be taken into consideration for interconnection networks performance measures. They used three types of interconnection networks, which are crossbar interconnection networks, multistage interconnection networks and multiple bus interconnection networks so they could enumerate the advantages and drawbacks of each and then make a good comparison. Since the lecture was bus oriented, the focus will be kept on the multiple bus content of this journal.

Relevance to the Lecture

This paper is relevant to the lecture because we were introduced to numerous interconnection networks and this paper compared three types of interconnection networks very carefully, giving us a better point of view on which is best in what situation. Also, even though this paper concentrates on the performance calculations of the multiple bus interconnection networks, it relates to the content of the lecture because we saw how performance in a bus could be increased when using different bus allocation techniques. It will surely help us in the design process when we will have to choose a certain bus implementation characteristic based on the desired performance.

Same Concepts Covered in the Lecture

A lot of the basic concepts studied in the lecture were present in that paper. In 1989, when the paper was written, they were saying that systems' bottlenecks were the communication time between processors. Since 1989, processors and interconnection networks were ameliorated, but we are still in the same situation as they were, communication is still the limiting factor in modern multiprocessor systems. An important quote from the paper is "Performance of multiprocessor rests on the design of its interconnection network". The multiple bus interconnection network was recognized to be better than a single bus interconnection network for its higher performance, efficiency and reliability and better than the crossbar because of it's lower cost. In fact, Figure9, which was found in the journal, tells a lot on the performance of those three interconnection networks. On the left, we have the bandwidth of the crossbar that is higher than the other interconnection networks. But on the right side, the processor utilization of the crossbar is almost the same as the four bus network.

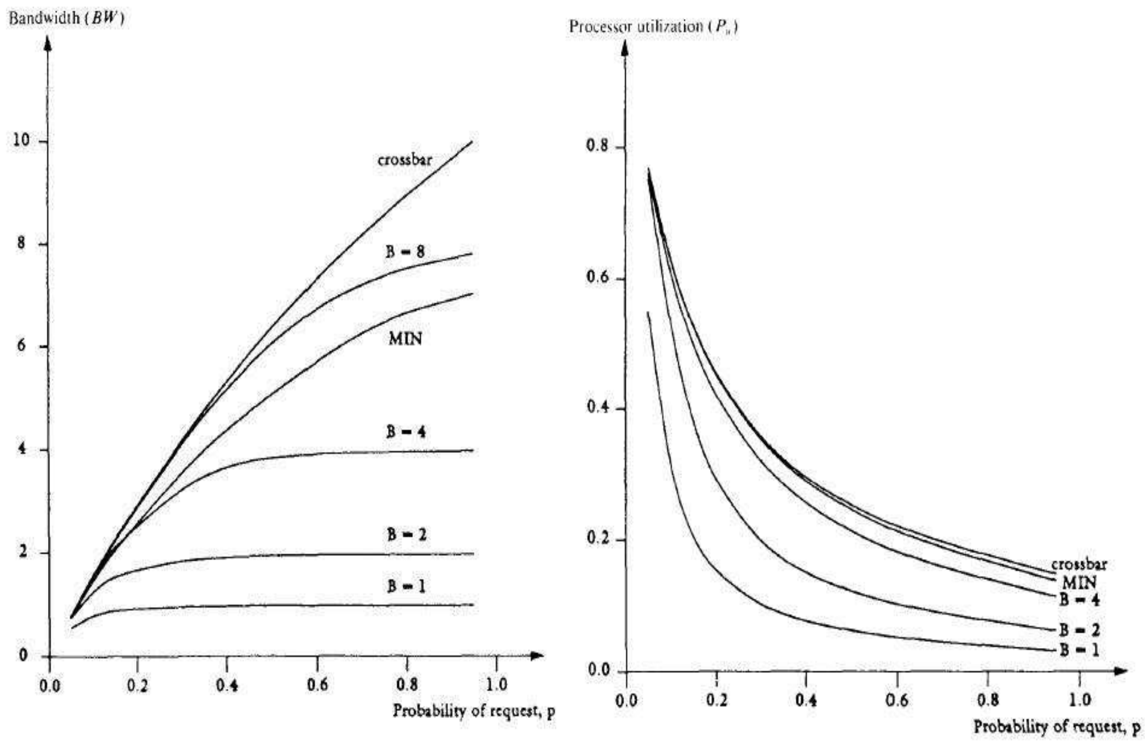


Figure9 Bandwidth and Processor Utilization of different interconnection networks [49]

Because single bus were the evident bottleneck in systems, the popular solution at that time, as it is today, was using multiple bus to increase the bandwidth and improve systems' fault tolerance. In the paper, both synchronous and asynchronous timing were described and it was told that synchronous was mostly used because it was well understood.

Differences and Relations to the Lecture

What is most different in that paper is that they don't talk about the arbitrator's influence on the overall performance of the buses. Also, they don't study any allocation techniques such as pipelining, split transaction nor burst messages. This paper only studies the actual bus performance that is related to bandwidth, or in other words, the performance of the buses when they are in full use. Their study is axed on "the number of bus increases the bandwidth" instead of "maximum resource utilization", which is what we saw in the lecture. Also, it is said that doing an analytical performance measure of a bus is a good and cheap way to ensure the quality of the design for a synchronous system although it is not the exact modeling of the bus. In order to make analytical observations, they need to make a few assumptions: all processors in the system are the same, and memory accesses are of an equal probability if there is more than one memory unit in the system. Analytical observations were not studied for asynchronous systems since they could not be modeled easily. The analytical way is taking account of the bandwidth of the bus, while we look at the whole "system", as it was said earlier. In short, the big difference found between the paper and the lecture was that they concentrated on the "bus" performance, while we did the same for the "system". A mix of these two approaches is needed to design the best system.

Paper 2 [50] Multilevel Bus Networks for Hierarchical Multiprocessors

The bus has become one of the limiting factors when designing a computer system. A single bus system is easy to implement and has very few connections, which limits the cost, but they are also very slow and have practically no fault tolerance. A multiple bus system is much faster and has a greater fault tolerance than a single bus system, but it costs more, because of the amount of connections, and takes a lot more space on the board than a single bus system.

In this paper, IEEE TRANSACTIONS ON COMPUTERS, VOL. 43, NO. 7, JULY 1994, Syed Masud Mahmud, published an article, named "Performance Analysis of Multilevel Bus Networks for Hierarchical Multiprocessors", on a new way to design a bus that has most of the best properties of both multiple and single level bus systems. He also proves mathematically that his system of bus design is better than other methods like multiple or a partial multiple bus system.

In 1983, Mr T. Lang found a method called partial multiple bus system to design a bus system which would be in between the multiple and single level bus systems. He proposed that you can divide all the memory modules (M) into groups (g). Each processors (N) connects to every bus but the memory modules are only connected to their assigned bus groups. Figure10 clearly shows how the connections are made. The amounts of connections in such a system is B*(N + M/g) because of the memory divisions into groups. The bus load in such a system is proportional to N+M/g. The problem with such a system is that there is still a lot of connections and the overall system performance is generally less than a multiple bus system. Mathematically, a multiple bus with M memory modules, B buses and N processors has a total number of connections of B*(N+M) with a bus load proportional to N+M.

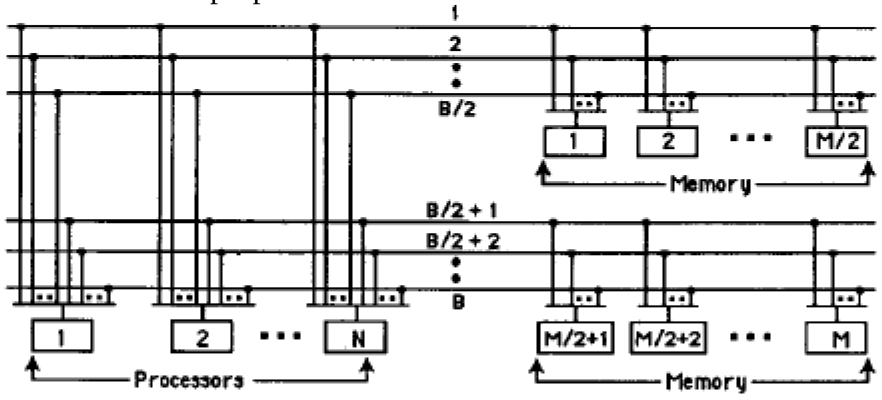


Figure10 Mr. T. Lang's partial multiple bus system.

Mr. Syed Masud Mahmud designed a bus system that can perform better then the partial multiple buses and almost as good as the multiple bus system and with fewer connections than both cases but it also has certain restrictions. The multilevel bus system can only be efficient if you have a NUMA system or a hierarchical multiprocessor system. A hierarchical multiprocessor system is when each processor is designated a specific memory module as his local memory. The processor has to access that memory module more often than the other memory modules. This principle is essential for the multilevel bus system to be efficient. The principle is that the processors and memory modules are divided into clusters. Each cluster includes N processors and M memory modules connected by B1 buses. Up to there it looks like a lot of small multiple bus systems. For the second level, every processor and memory modules from two different clusters are connected to a number B2 of second level buses. After that, it is an iterative method, two clusters from the (i-1) level together with Bi buses. Figure11 shows a theoretical example of such a bus.

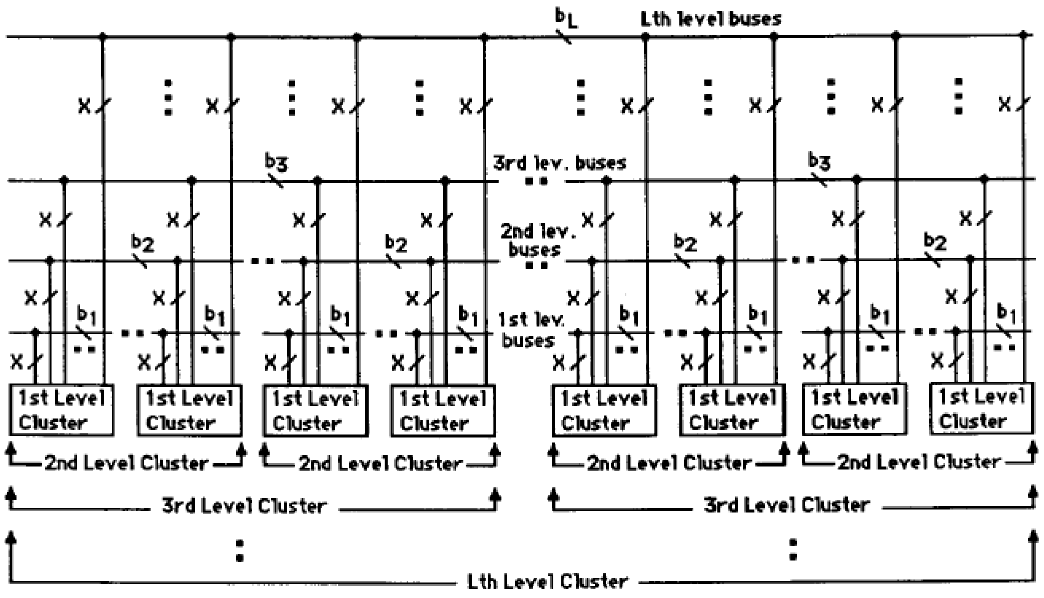


Figure11 A multilevel Bus system.

For such a system, the amount of connections are not less than the number you would find in a multiple bus system with the same amount of buses but the load per bus is divided by L (number of levels) compared to a multiple bus system.

Mr Syed Masud Mahmud goes on to introduce a second method of bus design based on the multilevel bus system called multilevel bus system with reduced loading. He figures that to minimise the amount of connections in the multilevel bus system is by not connecting every processor with every memory modules more than once. It gives the system less fault tolerance but reduces the amount of connections by half. Figure12 shows an example of both systems in their implementation.

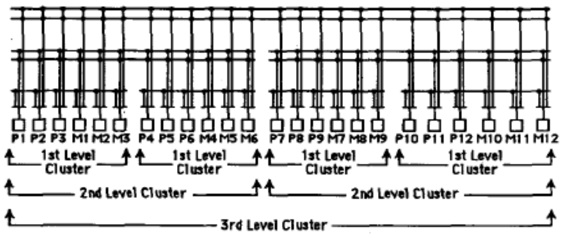

P1 P2 P3 M1 M2 M3 P4 P5 P6 M4 M5 M6 P7 P8 P9 M7 M8 M9 P10 P11 P12 M10 M11 M12
1st Level Cluster
2nd Level Cluster
3rd Level Cluster

Fig. 3. An MLB system with $n = 3, m = 3, L = 3, k_1 = 2, k_2 = 2, b_1 = 1, b_2 = 2$ and $b_3 = 2$.

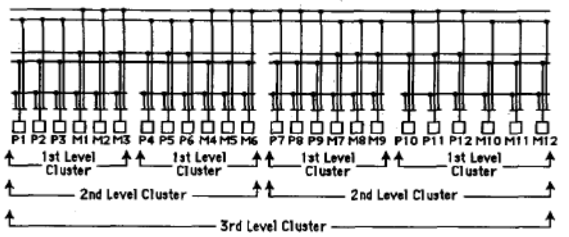
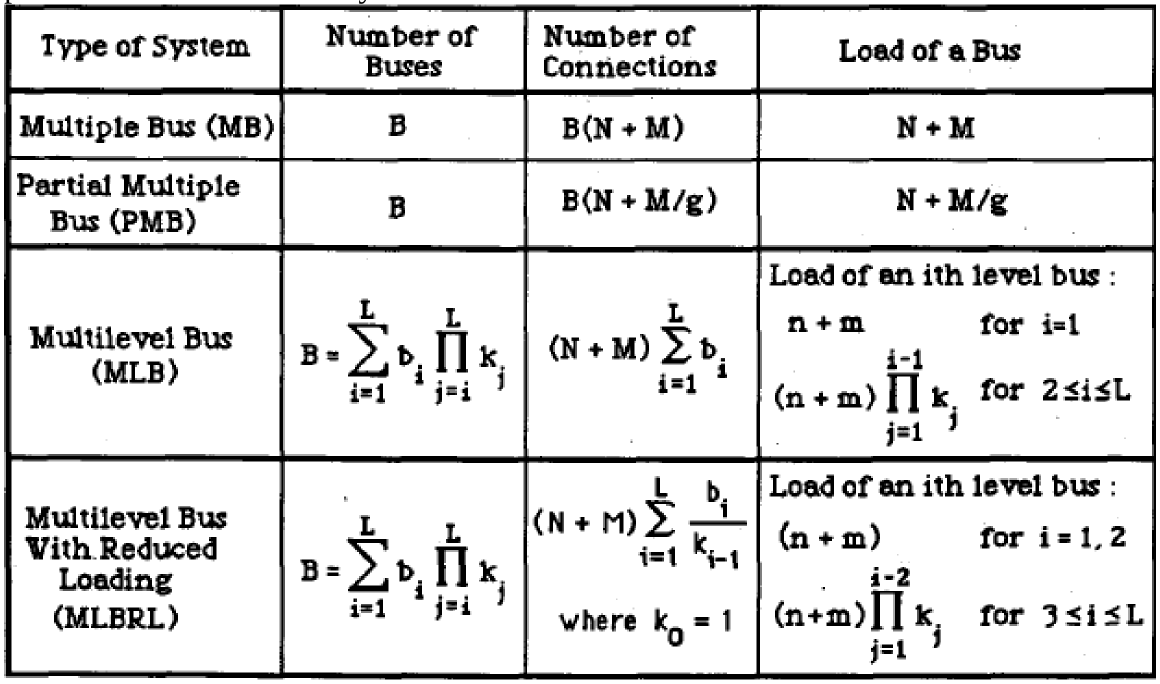

P1 P2 P3 M1 M2 M3 P4 P5 P6 M4 M5 M6 P7 P8 P9 M7 M8 M9 P10 P11 P12 M10 M11 M12
1st Level Cluster
2nd Level Cluster
3rd Level Cluster

Fig. 4. An MLBRL system with $n = 3, m = 3, L = 3, k_1 = 2, k_2 = 2, b_1 = 1, b_2 = 2, b_3 = 2$. (Note that $a_2 = 1$ and $a_3 = 1$)

Figure12 An example of both systems.

Finally, Mr Syed Masud Mahmud proves mathematically the advantages of his two designs. Figure13 shows the mathematical equations derived from the multiple bus, partial bus and both of his systems.



Type of System	Number of Buses	Number of Connections	Load of a Bus
Multiple Bus (MB)	B	B(N + M)	N + M
Partial Multiple Bus (PMB)	B	B(N + M/g)	N + M/g
Multilevel Bus (MLB)	$B = \sum_{i=1}^{L} b_i \prod_{j=i}^{L} k_j$	$(N + M) \sum_{i=1}^{L} b_i$	Load of an ith level bus : $n + m$ for $i=1$; $(n + m) \prod_{j=1}^{i-1} k_j$ for $2 \leq i \leq L$
Multilevel Bus With Reduced Loading (MLBRL)	$B = \sum_{i=1}^{L} b_i \prod_{j=i}^{L} k_j$	$(N + M) \sum_{i=1}^{L} \frac{b_i}{k_{i-1}}$ where $k_0 = 1$	Load of an ith level bus : $(n + m)$ for $i = 1, 2$; $(n+m) \prod_{j=1}^{i-2} k_j$ for $3 \leq i \leq L$

Figure13 Mathematical equations.

The multilevel bus system has many advantages but the development complexity is greatly increased. That is easily seen when you compare the mathematical equations. It also imposes a lot of constraints onto your design like the fact that your design has to be hierarchical and that farther a processor is physically from another module (processor or memory), the communication must be very limited or practically none existent.

This article was relevant to the lecture because it shows other efficient ways of designing a bus interconnection network. The concepts that we studied in the lecture that were found in this paper were related to single buses and multiple buses interconnection networks, which are described in the previous paragraphs. The novelties in this paper, compared to the material of the lecture, are that they design different bus architectures, which are more efficient under certain conditions. It relates to the lecture because they give us other ideas on how a bus can be implemented.

Papers Analysis of Lecture #7: Dynamic Interconnection Networks

Paper 1: Architectural Choices in Large Scale ATM Switches

Description of Paper

This paper discusses the design and analysis of switching systems related to Asynchronous Transfer Mode (ATM). Asynchronous Transfer Mode is a high performance technology and service for networking, based around switching connections and virtual circuits.

Switching systems are used in networks to reduce both cost and complexity. ATM introduces certain features that require certain considerations on switching systems, both through requirements for speed and capacity, and of course cost. The common switching systems design today use single stage systems, the most basic of which utilizes a simple static bus.

Bus-based switching systems are useful for low capacity systems, as they are relatively simple and cost-effective to implement. However, as seen in multiprocessor interconnection systems, when the capacity and performance requirements increase more sophisticated interconnection networks are needed. One design that can fulfill performance requirements for larger capacity systems is the Crossbar. Although crossbars can be more costly than other single-stage networks, it is importantly noted in the paper that cost is actually reduced in this configuration because similar performance can be achieved from a simpler, but faster, system. Figure 5 shows a crossbar-based switch for ATM network usage (the port processors can be seen in this figure).

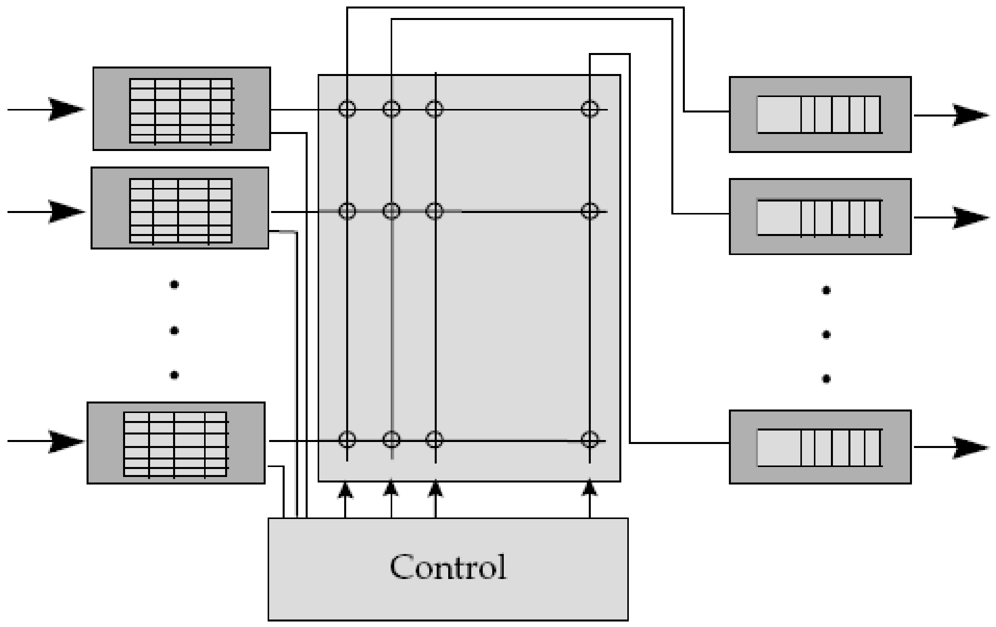


Figure 81: Crossbar-based Switch [54]

Single-stage switching systems however are limited because of their lack to handle simultaneous transfer operations. Although the speed of these can be improved, the greater parallelism offered by multistage switching systems is ultimately needed primarily to keep costs down. Dynamic networks offer cost effective systems for the performance they provide in large switching systems. An important feature of the ATM system is multicasting, where data from one input is sent to many receiving destinations. This utilizes the basic switch setting of broadcasting that was introduced in the lecture; however in the ATM system multicasting presents many challenges in assuring performance and cost reduction. When dealing with the amount of data capacity a modern ATM application is required to move, memory handling becomes a factor and any network contention becomes costly.

In the end, cost is of primary focus in this paper, with performance being a more inherent requirement in most actual applications. This is because there is usually a single requirement for performance, but various alternatives may exist that reduce cost.

Relevance to Lecture

Although application of switching systems for ATM operation was not discussed in the Dynamic Interconnection Networks lecture, many of the switching systems employ the same concepts as those that were identified for interconnection networks. The basic issues of performance and cost still apply, and each type of switching system will have its advantages and disadvantages, many of which are the same that apply to multiprocessor interconnection network systems. Network classification is again quite similar to those in interconnection networks, with the two primary classes being dynamic and static networks.

The evaluation of performance also relates to that of interconnection networks. As mentioned in the lecture the crossbar gains performance advantage because the amount of connections that are available, allowing elimination of network traffic blocking and also the ability to transmit over many connections simultaneously. This is also the case for the system described in the paper.

Differences from Lecture

The interesting thing to note here is that *port processors* are used to handle the buffering and routing of data for both input and output. Data entering the system goes through the Input Port Processors (IPP), which synchronizes data and routes it appropriately. Data exits the system (at the ATM level) through the Output Port Processors (OPP). These are mentioned as ATM-level components, and so is beyond what was discussed in the lecture.

The topology methods used are similar to interconnection networks, however one component introduced in the paper is the use of *buffers*. These are used both for queuing incoming data, and also for organizing outgoing data as the sequence may be out of order as they arrive at the OPP.

Paper 2: Performance Analysis of Future Shared Storage Systems

Description of Paper

This paper is to analyze 2 kinds of systems, BIP and KMIP. Each of them consists of several processors. Both systems inherit the concept of multiple processors sharing the same memory as what our lecture is focused on.

Interconnection networks for multiple processors basically share the same concept as other computer networks. Individuals connect to each other to share and access resources. Likewise, both systems are designed with either supercomputers or desktop computers, but they are considered as high performance or low performance processors. In order to do the data transfer and sharing with multiprocessors and multiple memory access, the bus network is introduced in both systems. The bus network here is a dynamic interconnection network since the connection is established during application as needed. The layouts use the crossbar networking methods. Both systems have not been reported to literature as of the journal’s date yet. This paper is analyzing the future concept of shared storage systems.

Relevance to Lecture

In our lecture, we discussed several dynamic interconnection topologies. Being dynamic interconnection network, we rely heavily on switches. Switches control the flow of the data; different topologies are just different layout of the switches. In the end, we are connecting multiple processors to multiple memory units.

In the two future storage systems introduced in this paper, they both apply the bus system with crossbar network architecture. The concepts used in this paper are the same as in our lecture. They both are switch based interconnection networks. However, our lecture focused on introducing the different types of dynamic interconnection networks and this paper focused heavily on calculating the performance of the 2 system concepts. There are many assuming and controlled environments in these systems as they were not really built as of the date of the paper.

BIP System

BIP stands for Billions Instructions per Second. This system consists of few supercomputers connecting to a large memory. The supercomputers are considered as several very high performance processors and they share a large memory and address space. Those high performance processors are connected to local cache memory units and all of them on the same bus architecture. High speed memories are also divided into modules and presented on the bus. The bus is then connected to large storing disks with speed matching buffer in between. An outline of a BIP system is shown in Figure 6.

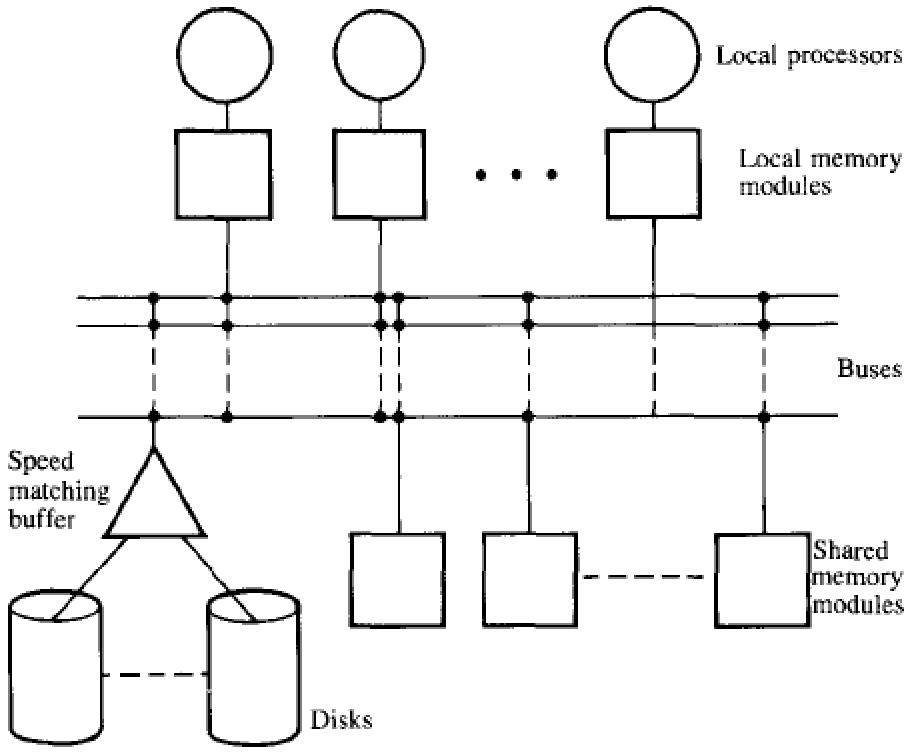


Figure 82: The BIP System [55]

The system is designed so that all processors would wait synchronously for the data to arrive from the shared memory. Performance formulas are indicated in the paper based on pre-built theories. The whole idea of BIP is to have extremely fast processing speed.

KMIP System

As opposed to the high cost of specialized supercomputer systems, the other way of the future is low cost, available-to-everyone systems. The *KMIP* system offers the same concept but with several 1-10 MIPS workstations. So individuals who do not need the extreme processing speed can still share memory with many other individuals at low cost in the future. A KMIP system is shown in Figure 7.

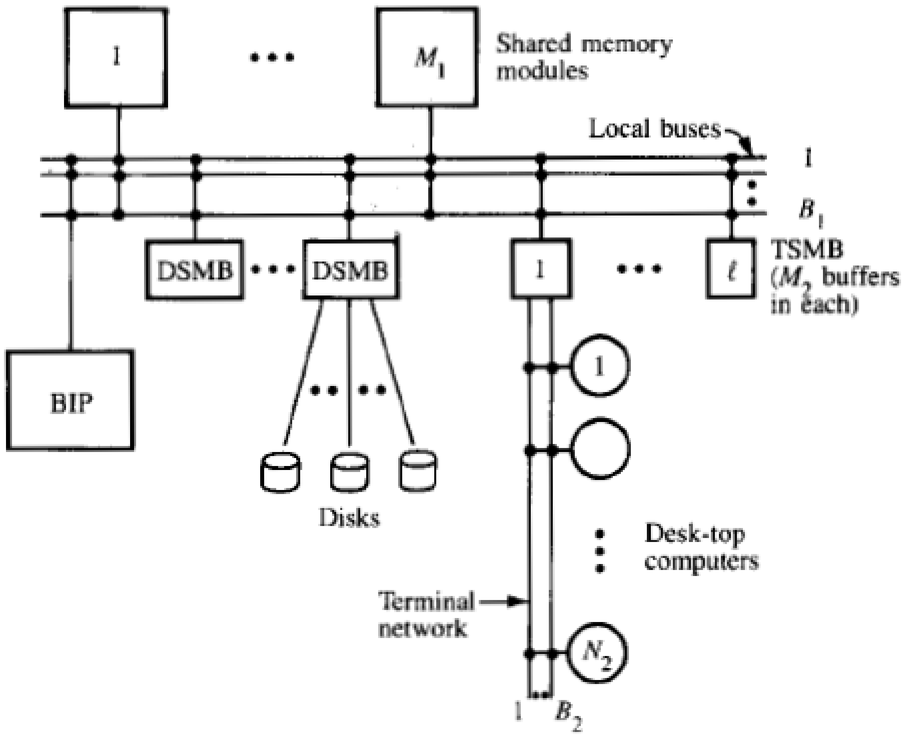


Figure 83: The KMIP System [55]

The shared memory is modeled to have the ability to process page requests from all the workstations. Data transfers and storage device access play a very important role in KMIP system because there are many workstations that may request a lot of files and access many memory locations at the same time. In a KMIP system, multiple workstations are modeled to illustrate as multiple processors.

BIP vs. KMIP

Both systems are required to have large random access memory besides the main storage spaces. Those random access memory are required to have high intelligent of accessing the storage devices and distribute the shared data. Both systems use the speed matching buffer to take care of the slow accessing speed of storage devices. There are a few key points to design a model for experiments. Many things need to be considered to design them. For example the design space is limited, so there must be a reduction in design spaces using quick parametric studies. Handling of memory access is important and many techniques can be used. These are the key consideration in the design model. As far as the differences between the 2 models go, they are still based on multiple processors connecting to multiple memory units with switches in between.

Paper Analysis of Lecture #8: Dynamic Interconnection Networks

Paper 1 Nonblocking Properties of Interconnection Switching Networks

1 Description of Paper

The focus of this paper is to analyze the different permutations of interconnection networks and develop a method to determine which of these permutations have the property of being non-blocking. These non-blocking permutations are of significant interest mainly because they allow for the maximum usage of the interconnection network. The paper also demonstrates a method, the "Balanced Matrices Characterization", which is used in Inverse Omega Networks to determine if a permutation can be completed in the given network. The paper also discusses the topological equivalencies between Inverse Omega Networks, Baseline and Flip networks. Lastly, it outlines some applications of the Omega Network in multicasting and how the characterization process of the networks can help establish non-blocking connections. [62]

2 Similarity to Lecture

This paper discusses several topics which were related to our lecture. The most important topic is the Inverse Omega Network (ION). This network differs from the Omega network viewed during the lecture because of the routing in between the switches. The routing in the ION is achieved using the opposite mechanism we studied for the Omega Network. The ION uses a right-shift to determine the link connection. The equation for the number of stages for an ION, $\log_2 N$, is also concurrent with what was explained during our lecture. It also describes the method that is used to determine the routing from one input to another by using the destination address. In the ION, you start with the LSB where in the Omega Network we start with the MSB to be used as a control signal for the switches. A 0 still represents the upper path and a 1 represents the lower path. [62]

3 Differences from the Lecture

The paper discusses in more detail the permutation possibilities of interconnection networks and how they can lead to blocking within the network. It shows how to use the WINCHECK algorithm to determine if there exists a possible non-blocking permutation by analyzing the Balanced Matrix obtained from the network. This is a scientific approach to find a non-blocking permutation. Our focus in the lecture was more to see if there could be blocking by choosing several different paths in the network and observing conflicts in the required settings for switches. The authors also relate the ION to several other interconnection networks such as the Indirect Binary n-cube which is topologically equivalent but was not covered in our lecture. [62]

Paper 2 Crossbar based design schemes for switch boxes and programmable interconnection networks

1 Description of Paper

The Focus on this paper is on programmable on-chip interconnection networks. More specifically, the authors deal with modifying and improving the crossbar network to what they call a meta-crossbar network. The paper begins by detailing the composition and functionality of the traditional crossbar network. It then goes on to explain the make-up of the meta-crossbar and how it can be used to implement any switch box. The authors then review the routing details, such as requirements, feasibility, and capacity. Finally, we are presented with an optimal meta-crossbar design. An application of the meta-crossbar as described in the paper would be for large interconnection networks where a meta-crossbar would be placed in the middle stage and a full crossbar on both sides. This would provide a 3-staged rearrangeable interconnection network. The design goal for the meta-crossbar is to provide a more flexible and practical switching module. [63]

2 Similarity to Lecture

The paper greatly details the classic crossbar network as reviewed during our lecture. The make-up of such a network was revisited when it explained an n x m crossbar has n horizontal wires that equate to the input, and m vertical wires which are the output, with a crossing switch at each cross point. Also detailed is the fact that an n x n crossbar network can perform permutations for n signals. Disadvantages of crossbar networks are also presented. Specifically, it mentions the high area cost of crossbar network implementations to add to the high cost of extra links compared to other interconnection networks outlined in the lecture. [63]

3 Differences from the Lecture

The document examined differs from what we have learned in class through the introduction of a new network through the customization of a pre-existing network. The authors provided a more flexible design for on-chip interconnection networks, detailing the limitations of current alternatives while providing sound reasoning on their choices. Also shown by the authors is how they satisfy the routing requirements through the presentation of theorems and proofs. [63]

Papers Analysis of Lecture#9: Static Interconnection Networks

Paper1 [67] Performance of Multiprocessor Interconnection Networks

This first paper is about how to evaluate network performance in multiprocessor designs and presents tutorials on evaluation tools to guide designers through their interconnection network designs for parallel computer architectures.
.
Authors of this paper divide general-purpose parallel/distributed computer systems into two categories; multi-processors and multi-computers. They present shared memory with multiprocessors and the message passing system is presented with multi-computer.. Since the paper is written in 1989 researches couldn't cover the modern techniques about parallel multiprocessors, but at that time parallel computing was an important research topic.

This paper references multiprocessor Inter Network topologies. They divide this subject in to two categories as synchronous and asynchronous control techniques.

Another subject of this paper is performance analysis of interconnection network techniques. Memory interference and bandwidth are main drawbacks of the designs. Multistage networks are preferred since they cost less than crossbar networks. Performance analyses of multiple-bus systems, crossbar interconnection networks and multistage interconnection networks were three major points this article mention.
In the lecture we were focusing on static networks only. Even the paper doesn't mention static networks that much it was decent paper on interconnection networks for 1989.

Paper2 [68]Static interconnection network extensibility based on marginal performance/cost analysis

This paper is introducing a new approach in the extensibility analysis of computer interconnection networks. The researchers compare the available static interconnection network performance in terms of marginal change.
"The paper examines the trade-off between cost and performance of a hypercube based computer interconnection network founded on a detailed cost analysis model. "
Authors of this paper introduce and define the Boolean hypercube, the nearest neighbour mesh hypercube, the Generalised hyper cube, and the Spanning multi-access channel hypercube. Usage of fiber optic between links are well studied in to prove that it is going to be cheaper and less work to build a multiprocessor interconnection network by using fiber optic links.

The main purpose of this paper is to build a multiprocessor system with many nodes for a relatively low cost by using the high capacity channels available with fiber optical communication techniques.

On the page 14 of the paper they demonstrate the scale in BW available with fiber optic communication links, structures consisting of a reduced number of more expensive, high capacity communication channel. Marginal performance over cost, bandwidth scaled for equalized graph, -Marginal cost, comparing topologies, bandwidth, and interconnection complexity graph – and the Marginal performance over cost, bandwidth scaled for equalized performance graph are used to demonstrate the marginal performance.

We have seen cube topologies under static internetworking topologies but this paper defines different types of cube topologies which we haven’t seen.

Papers Analysis of Lecture #10: Shared memory systems

Paper 1Transactional Memory Coherence and Consistency

by Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun

Stanford University, March 2004

This paper discusses a new shared memory model based on transaction: Transactional Memory Coherence and Consistency (**TCC**). This model borrows the principle of transaction from database transaction processing systems to create a model half-way between shared memory and message passing multiprocessors systems. The idea behind this is to split the code into small sets that will be processed in groups that are forced to be processed from one end to the other just like atomic actions. In a case where a transaction is committed and has accessed data that was no longer valid, a rollback is made requiring the application to restart the set of instruction from the beginning with the updated data. When a transaction succeeds, a broadcast message is sent to the other processors to inform them of the modified data. Such a model requires read and write buffers; analysis show that their size can be relatively small (4-12KB). The team behind the paper also did some simulations to prove the performance of the model.

This paper is relevant to shared memory as it is trying to develop a new model solving the problems of shared memory; cache coherence and complex hardware implementation.

With a TCC model, if a transaction commits and hits some modified data, it will restart from the top after doing a roll back. Hence, there is no need to think about cache coherency as multiple versions of the data variables exist among the system, all stored in each processor buffers. The drawback is that such a rollback decreases the speedup factor. To solve this problem, special care must be taken care in the software design to create smaller transaction when high probability of rollback can occur. Otherwise, transactions should be as long as they can be where the probability of a rollback is low, to reduce the bandwidth requirements of broadcast messages. The programmer should also make sure that the transaction includes every pair of load and store instruction. Hence, this is a disadvantage compared to the shared memory model that usually focuses on simple programming interface. Although, comparing it to the message passing model, the programming interface is less complex; which creates a middle ground between the two popular models.

The other problem of shared memory model that the TCC model tries to correct is the high complexity of the hardware required to manage cache coherency and data access synchronization. TCC simplifies the hardware requirement by eliminating the need for synchronization using conventional lock mechanism as it is all managed by the transaction model. TCC instead requires some buffers and some additional bits (Read, Modified and Renamed) to allow correct data validation during transaction commits. This hardware complexity is still less than that needed to implement the locking mechanisms needed for shared memory.

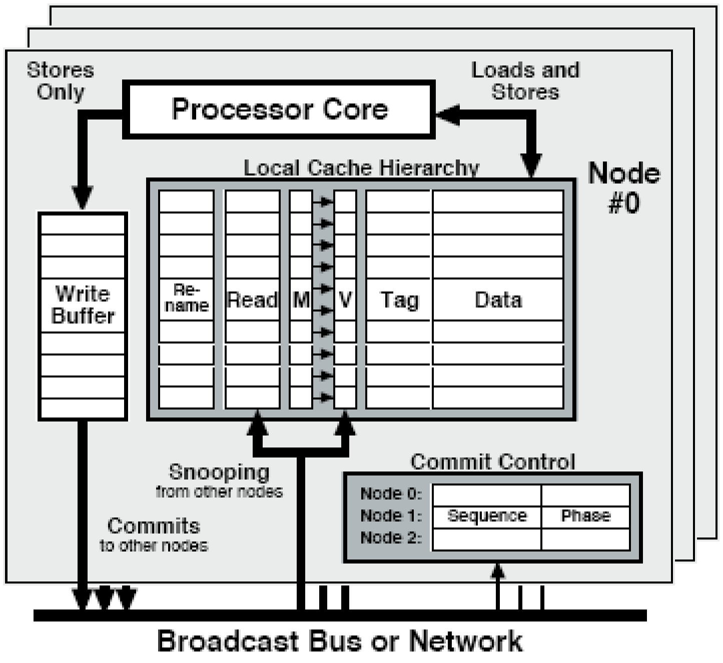


Figure 84 A sample 3-node TCC system

The following figure shows the added buffer and extra bits required for cache data validation. A commit controller must also be added to manage the various transactions.

The disadvantage of the TCC model is the high bandwidth requirement needed to broadcast the state of modified variables after each transaction. This disadvantage is minimal considering that the future technology will be equipped with high-bandwidth connection buses or networks; as stated in the paper.

In conclusion, TCC tries to improve the shared memory in many ways and succeeds at a certain bandwidth cost. The simulation results elaborated in the paper shows a good potential for this model and at worst will allow improvements of the basic shared memory model with some new strategies available to designers.

Paper 2 Performance Measurement and Modeling to Evaluate Various Effects on a Shared Memory Multiprocessor

By Xiaodong Zhang

IEEE Transactions on Sofware Engineering, Vol. 17, No. 1, January 1991

This paper revisits the existing models of multiprocessor performance, such as Amdahl's Law, which was studied by the class, and evaluates the effects of various other factors. These include mainly sequential code, barriers, cache related computing, and virtual memory demand-paging. Most of these factors have been presented in the material on shared memory systems, but they have not before been tied into a performance model for these systems. The author then proposes a new model for shared memory multiprocessor systems.

The author uses simulations on a multiprocessor system to determine the effect of each factor. The tests were run on an Encore Multimax system with 20 32-bit processors which uses a unix based operating system. This system uses a bus-based shared memory architecture.

Sequential code was the first effect analysed and was done with the existing Ware model. The author calculates an *overhead factor*, which is the ratio of sequential code to the entire program (1). He also states that the overall computing time increases with the number of sequential code sections. It is not clear however whether different numbers of sequential code sections were actually tested having an equal total amount of sequential code, which would test strictly the effect of the number of sequential sections and not just the amount of sequential code, as taken into account by the *overhead factor*. If this is a factor, it is not accounted for in the overhead factor and it is a flaw in Zhang's model.

$$r = \frac{t_s}{t_s + t_p} = \frac{1}{\alpha} \qquad (1)$$

The effect of barriers is examined next. Since this involves a number of processors waiting for each other, the waiting time will depend on the processor which takes the longest. The author calculates a barrier effect timing function (2) which is based on the mean time and standard deviation for processors to complete their work section.

$$t_b = \sigma\sqrt{2\log p} \qquad (2)$$

Cache related computing involves several factors, such as the amount of processor locality, the size of the code that needs to be cached in relation to the total cache size, and overhead caused by cache coherency algorithms, which we have seen can be a complicated problem. The author uses a related effect, virtual memory demand-paging, to determine a timing function.

After studying each effect, the author formulates a new function for execution time in a shared memory multiprocessor system. Starting from W. H. Ware's model:

$$T(p) = t_s + \frac{t_p}{p} \qquad (3)$$

The author then adds the factors for sequential code time section, $\hat{t}_s$, the barrier overhead time section, t_b, and the paging effect time section, t(m), to produce the resulting model:

$$T(p) = \hat{t}_s + t_b + t(m) + \frac{t_p}{p} \qquad (4)$$

Zhang proposes an innovative model, and it seems advantageous to its accuracy to take into account these new factors. However, for it to be established as a valid model for shared memory systems, it should be validated for a wide range of hardware. That the author only used one system in the development of the model puts into question its usefulness for analysis on other shared memory systems.

Paper Analysis of Lecture #11a: Open MP

Paper 1 A Practical Open MP Compiler for System on Chips

by Feng Liu and Vipin Chaudhary

SOC designs are becoming an increasingly popular shared memory parallel architecture. OpenMP standard cannot support many features of SOC and there is no other standard that can. This article focuses on the design of an OpenMP compiler for System-on-Chip (SOC), or more specifically for Software Scalable System on Chip (3SoC) from Cradle.

3SoC is a SOC with programmable I/O for interfacing to external devices. Multiple processors are connected by two level buses. A cluster of processors called a Quad is connected by a local bus and shares local memory. Each Quad consists of four RISC-like processors called Processor Elements (PEs), eight DSP-like processors called Digital Signal Engines (DSEs), and one memory Transfer Engine (TWE) with four Memory Transfer Controllers (MTCs). The most important feature of 3SoC for parallel programming is that it provides 32 semaphore hardware registers to do synchronization between different processors within each Quad and an additional 64 global semaphores. A block diagram of 3SoC is shown in Figure 2 and a Quad block diagram in Figure 3.

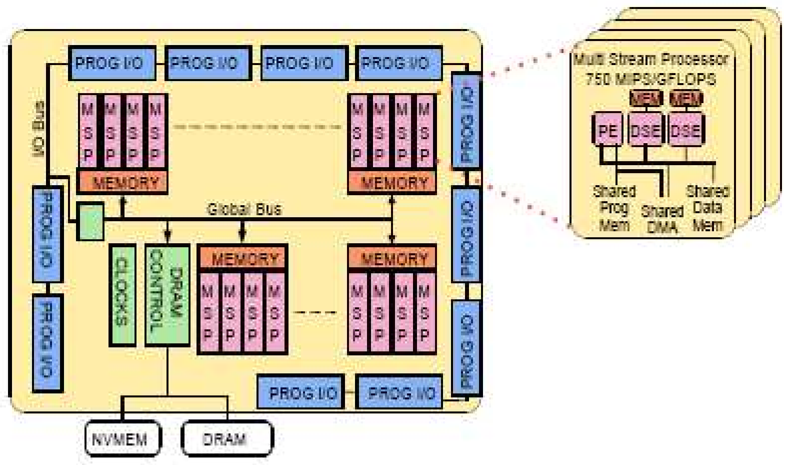


Figure 2: 3SoC block diagram [76]

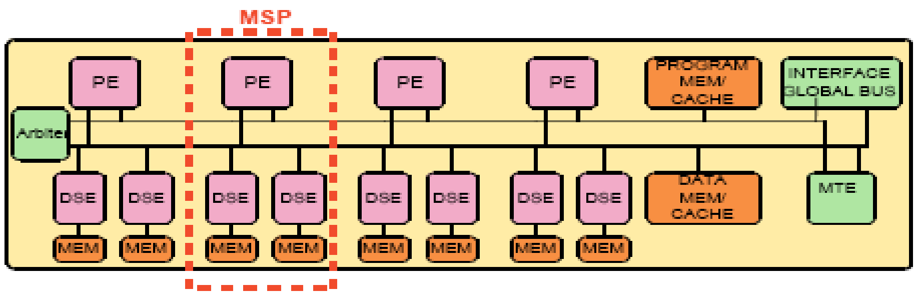


Figure 3: Quad block diagram [76]

The OpenMP compiler was designed with special focus on synchronization, scheduling, data attributes, and memory allocation. On 3SoC, one major synchronization pattern that can be implemented is by associating one hardware semaphore for locking and unlocking, then the user can define a critical construct which is mutually exclusive for all processors. Another pattern is done by combining one semaphore along with global shared variables, and then an OpenMP barrier construct can be achieved across all processors. The barrier implementation in the second pattern is important for OpenMP. To implement this we first allocate all semaphores into local shared memory to improve data locality and then we allocate semaphores dynamically at run-time.

Scheduling is implemented both statically and dynamically on 3SoC and the user can choose which one through the OpenMP compiler. For chip multiprocessors (CMP) each thread is a separate processor which results in static scheduling providing the best performance on average. Dynamic scheduling involves more synchronization which interrupts the processors more often.

Memory allocation must be done dynamically in order to produce better performance. This is done using the "request-and-grant" model during compilation. At compilation, the compiler gets all of the memory allocation requests from each parallel region and then assigns different memory.

Several extensions to OpenMP were designed to take advantage of some of the new features of CMP. OpenMP extensions include a set of DSE directives to deal with the heterogeneity of different processors within CMP. Using these OpenMP extensions helps because it provides high-level abstraction of parallel programs by hiding DSE implementation details.

In 3SoC, the MTE processor is a memory transfer engine that runs in parallel with all other processors. It transfers data between local data memory and DRAM in the background. MTE extensions to the OpenMP compiler allow the programmer to write parallel programs which control actual data movement dynamically at run-time, resulting in a significant performance speedup..

To implement an OpenMP compiler for 3SoC, there are four major steps.
Parallel regions – Each parallel region is assigned a function number that corresponds to a different processor.
Data range – The data that is accessed by the program is used by the compiler to determine the data range of functions and to assign memory allocation.
Work sharing constructs – Using processor id's, the compiler assigns a portion of work to each processor.
Synchronization – The compiler sets up any process synchronization by allocating a number of hardware semaphores either statically or dynamically.

The article also summarizes a performance evaluation that shows that 3SoC architecture is suitable for large intensive computation over multiple processors on one chip and that performance is scalable. Results show that extensions to OpenMP lead to a performance improvement on 3SoC and that OpenMP compilers would reduce the burden for programming SOCs.

OpenMP is very efficient at simplifying parallel programming. Currently, programming SOCs can be very complex and this article provides a solution to this. It presents an OpenMP compiler targeting 3SoC which provides high level abstraction and improves the overall efficiency of parallel programming for SOCs.

Paper 2 OpenMP Implementation and Performance on Embedded Renesas M32R Chip Multiprocessor

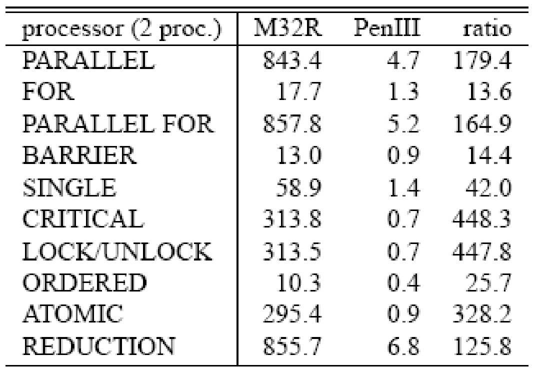
by Yoshihiko Hotta, Mitsuhisa Sato, Yoshihiro Nakajima, Yoshinori Ojima

This paper reports implementing OpenMP compiler for an embedded Renesas M32R chip multiprocessor as a parallel programming environment. This paper reports the preliminary performance of OpenMP benchmarks, including scientific and multimedia applications on the M32R CMP. We found that OpenMP allows users to easily obtain reasonable performance improvement using multiple CPUs in the CMP with just a few directives inserted. The possibility of OpenMP threads run-time scheduling and some compilation techniques for power-aware computing on the CMP is also discussed.

OpenMP is an easy-to-use simple parallel programming environment that makes use of the multiple CPUs in the CMP (chip multiprocessors) with multiple processors on a single chip. One of the research interest was is power-aware computing using a CMP. Recently, there has been tremendous interest in power-aware computing for mobile embedded systems such as PDAs and cellular phones. Even in high-performance Parallel computing systems, it is very important to reduce the power consumption for cooling and high-density packaging. The recent research shows that a cluster of low-power processors can obtain better performance than a single high-performance processor in terms of the power-performance ratio. CMP allows us to exploit both high performance and low power since it can achieve high performance while keeping a low clock frequency behaving multiple processors on a single chip. An interesting feature of OpenMP is that the execution of OpenMP programs is independent of the number of actual physical processors. This feature can be a useful tradeoff between application performance and power consumption in the CMP. For example, when an application is required to reduce the power consumption, which may cause the application to run slowly, we can run the OpenMP program at low power by stopping the power supply to some of processors. The roles of OpenMP for embedded CMP are as follows:

• Providing a portable parallel programming environment. OpenMP provides a standard API in C/C++ and FORTRAN for a shared memory machine used for embedded applications.
.• Providing an easy to use parallel programming model. The original sequential program may be parallelized easily by inserting OpenMP directives. Actually, the POSIX thread library is often used to develop embedded applications. An OpenMP-style programming model can reduce the cost of parallel programming by eliminating tedious codes for managing and synchronizing threads.
• Exploiting parallelism on embedded multimedia applications. Time-consuming multimedia applications often have much parallelism so that the parallelization makes these applications faster.

M3T-M32700UT is a system that consists of an M32RCMP processor in [3], LCD, network interface, etc. TheM32R processor is a product of Renesas Technology and is a very low-power processor for embedded systems. OpenMP was implemented on the M32R CMP, using the Omni OpenMP compiler. The Omni OpenMP compiler is a translator that takes OpenMP programs as input to generate a multi-threaded C program with run-time library calls. The generated program is compiled by the native cross compiler and linked with the run-time library. The run-time library includes micro tasking and synchronization primitives on top of the different thread libraries, which includes POSIX threads. Since Linux/M32R supports POSIX threads, it was very easy to port the run-time library to the system. Table below shows the overhead of Synch bench using the EPCC micro benchmark. For reference, we also measured the performance on a Pentium-based SMP platform (Pentium 550 MHz). The overhead of this result, we found dramatically difference between M32R CMP (2 processors)and Pentium III (2 processors). In particularly “CRITICAL” and “LOCK/UNLOCK” spends over few hundreds of times time. On the other hand, we found little difference in “FOR” and “BARRIER” overhead. But in all case, we found the ratio of each processors overhead is large than the difference of processors clock frequency. So, this result means as compared with high performance processor, M32R CMP has large overhead in case of using OpenMP.



processor (2 proc.)	M32R	PenIII	ratio
PARALLEL	843.4	4.7	179.4
FOR	17.7	1.3	13.6
PARALLEL FOR	857.8	5.2	164.9
BARRIER	13.0	0.9	14.4
SINGLE	58.9	1.4	42.0
CRITICAL	313.8	0.7	448.3
LOCK/UNLOCK	313.5	0.7	447.8
ORDERED	10.3	0.4	25.7
ATOMIC	295.4	0.9	328.2
REDUCTION	855.7	6.8	125.8

Table 2. Results of EPCC micro benchmark [micro sec]

OpenMP on CMP for Power-Aware Computing [78]

The typical requirement in real-time applications is to execute a reserved job within a certain period. Although parallel execution runs a program faster than sequential execution does, it is not necessary to execute in parallel with respect to power efficiency. Even though it may take a longtime to meet the deadline, the program can run on one CPU with other CPUs in the standby mode. For such run-time thread scheduling, co-scheduling with OpenMP run-time and kernel thread scheduling on the CMP will be required. This experiment measured power consumption which was 3 Watts all the time The difference between the OpenMP execution and the sequential execution was not much, because most of the power was used for devices such as the LCD and the network, and the power consumption of the CPUs is only on the order of a few hundred mill Watts(mW). In addition, there is no kernel support to put the CPU in standby mode. In the near future, kernel support and OpenMP thread scheduling for power aware computing would be investigated.

Papers Analysis of Lecture # 12 a: Parallel Programming Models

Paper 1: Parallel Programming Models for a multiprocessor SoC Platform Applied to Networking and Multimedia. [83]

Summary
This paper discusses the novelty of the Parallel Programming models using a tool, The MultiFlex system, which is application-to-platform mapping tool. It integrates heterogeneous parallel components into a homogeneous platform programming environment. The most important part of this paper is to show how the MultiFlex is supporting the parallel programming models. There are two models supported by this tool: a distributed system object component (DSOC) object-oriented message passing model and a symmetrical multiprocessing (SMP) model using shared memory. The main factor of this tool is to design an application-free tool; it maps application-to-platform that integrates parallel components. The task level communication may be blocking or non-blocking; this allows the designers to trade off between the efficiency and portability. Also, this tool provides a threading API for parallel execution in shared memory. [83]
The MultiFlex approach professes four key contributions above previous parallel programming models:
"1) Interoperable distributed objects and SMP programming model support, with abstractions more inline with widely adopted industry standards (POSIX threads and distributed object systems such as CORBA).
"2) […] Novel hardware accelerators [are provided] for message passing, context switching, and dynamic task scheduling and allocation.
"3) Support of homogenous programming styles […] via a system interface definition language (SIDL) and an associated compiler, which support a neutral data format for message passing.
"4) All application programming may use a high-level language (C or C++, with high-level calls to the parallel programming model APIs)." [83]

In SMP implementation, the cost of forking a thread or synchronizing must be balance properly in order to avoid granularity of the task. There are SMP overhead due to small tasks. Also, SMP supports tightly coupled programming with languages such as Java and C#. In conclusion, DOSC and SMP models are using message passing and shared memory respectively. [83]

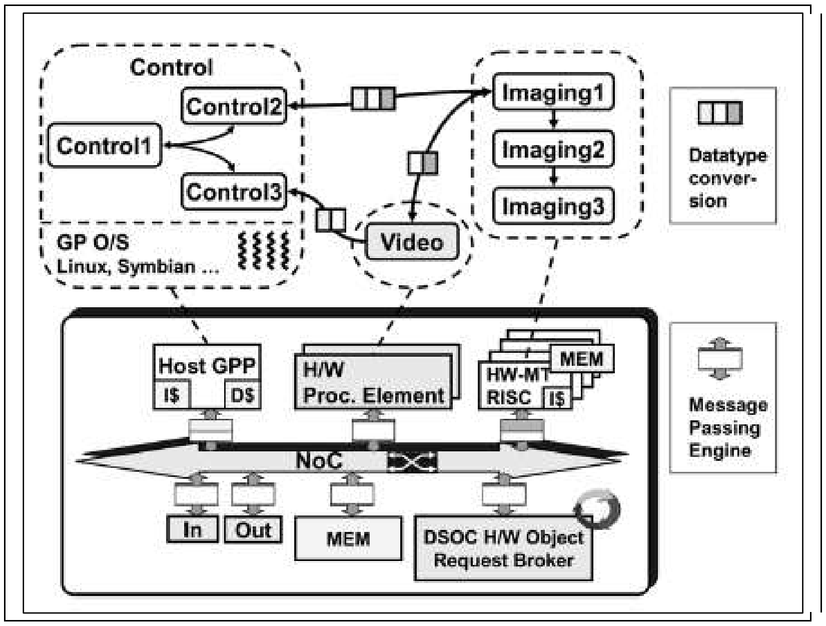


Figure 85 - DSOC Model [83]

As shown in Figure 85, DSOC model relies on parallel communication between objects: hardware message passing and object request broker and thread management. The blocking comes here when the hardware thread is delayed until data is ready in another thread; DSOC non-blocking objects will execute in parallel. Each hardware element, compiler creates data conversion hardware model and it links them to NoC interface. [83]

The MultiFlex is a platform programming environment for the StepNP SoC. It consists of configurable hardware threaded processors, shared memory, and network oriented inputs and outputs using NoC. Since it can be mapped on variety of the processors as configurable, it communicates at lower cost and higher processor utilization. The paper concludes with current application and future research on the MultiFlex Technology. Currently, MultiFlex applied on multimedia and wireless applications. Future researches include memory architecture, configurable processors and priority-based scheduling. [83]

MultiFlex has developed hardware to support the programming models. The StepNP multiprocessor SoC architecture explores efficient SMP and message-passing implementations. In order to eventually support virtual machine styles of programming like Java or any of the Microsoft Common Runtime Languages, hardware support will be needed to make the parallel programming more transparent. These include hardware support for locks and context switching as well as task scheduling which are greatly faster than software implementations. The experiments on hardware implementations have increased efficiency from 30% to 63%.[83]

Relevance of topic

This paper is selected to analyze the parallel programming model because it is directly linked to the lecture in several ways. The lecture covered few parallel programming models: Shared memory, message passing, Threads, Data parallel and Hybrid models. The authors talk about the message passing models which is in DSOC and shared memory model which is in SMP both are covered in the lecture. There are few other concepts used in the lecture such as blocking, non-blocking, granularity and synchronization also covered in this paper. [83]

The similarity of the lecture with this paper is that both talk about Shared memory, Massage passing and thread models. The paper and the lecture talk about the shared memory and the massage passing models in detail compare to other models. The paper is not talking about hybrid or data model unlike the lecture does. [83]

Paper 2: THE STANFORD HYDRA CMP [84]

Summary

By Moore's law, single processor chips are getting faster and contain more transistors. However, this increases the area and therefore the wire-distance required to get from one element to another. As chips are getting faster, this implies that eventually, it will take more than one clock cycle to get information from one part of a single processor to another. The hydra CMP designs processors that take small areas on the chip. [84]

Parallelism is often achieved by seeking non-dependant instructions, that is instruction-level parallelism (ILP). The overhead required in identifying ILP tends to demonstrate that the benefits of such approaches are limited. Instead, the Thread-level parallelism (TLP) executes completely separate sequences of instructions simultaneously. It is possible to combine the two approaches to some extent for optimal performance. [84]

Designing more complex single processors is an increasingly difficult task. CMP's can be more easily design by combining existing and proven processors. The hydra CMP uses four MIPS processors that are synchronized with load-locked and store-conditional instructions. The processors communicate between each other over a central bus. [84]

"In the chip implementation, almost all buses are virtual buses. While they logically act like buses, the physical wires are divided into multiple segments using repeaters and pipeline buffers, where necessary, to avoid slowing down the core clock frequencies." [84]

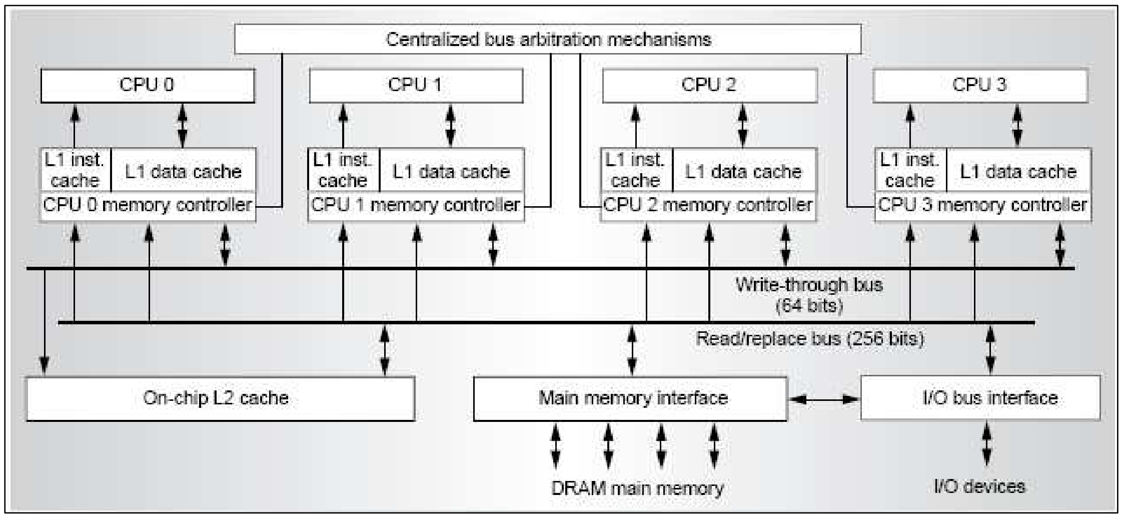


Figure 86 - An overview of the Hydra CMP [84]

Thread-level parallelism is used on the hydra CMP. Hydra CMP uses write bus to update the global object using shared memory, but it needs to find a balance to reduce the interprocessor communication latencies. As a result of multiprogramming for the hydra, the speedup is obtained almost linear. [84]

Since Hydra uses caches to speed up the process, it needs the memory consistency to write to the bus. All the access must go through the bus, it needs to be made globally, so it needs to update the shared memory. The communication between the different caches will cause communication latencies. Hydra must find a reasonable mid-point between complexity and the communication latencies. During the implementing of the hydra, the thread will be crated automatically; it will require a large burst of data communication.

To control overlapping the write and read thread, there is a buffer allocated in the hardware is integrated in the MIPS. [84]
Now that the hardware support for parallelism is established, it must be shown that the CMP will improve the performance of benchmark software. In fact, the goal is not only to improve the performance of programs that are designed for Threaded performance such as database queries and web-services, but also programs that are normally suited for single processor performance. To do so, some mechanism is required to transform a random, unknown program into a parallel program. Hydra CMP proposes Thread Speculation:
"Speculation allows parallelization of a program into threads even without prior knowledge of where true dependencies between threads may occur. All threads simply run in parallel until a true dependency is detected while the program is executing. This greatly simplifies the parallelization of programs because it eliminates the need for human programmers or compilers to statically place synchronization points into programs by hand or at compilation. All places where synchronization would have been required are simply found dynamically when true dependencies actually occur." [84]

To support speculation, the hardware must handle a number of requirements:
Forward data between parallel Threads,
Detect when reads occur too early,
Safely discard speculative state after violations,
Retire speculative writes in the correct order,
Provide memory renaming. [84]
Tests have shown that speculation on the Hydra CMP perform at least as well as on large single-processor chips. Of course, the Hydra CMP can also run parallel programs without speculation, which the single-processor chip cannot. It is also conceivable to get both parallel programs and serial programs (using

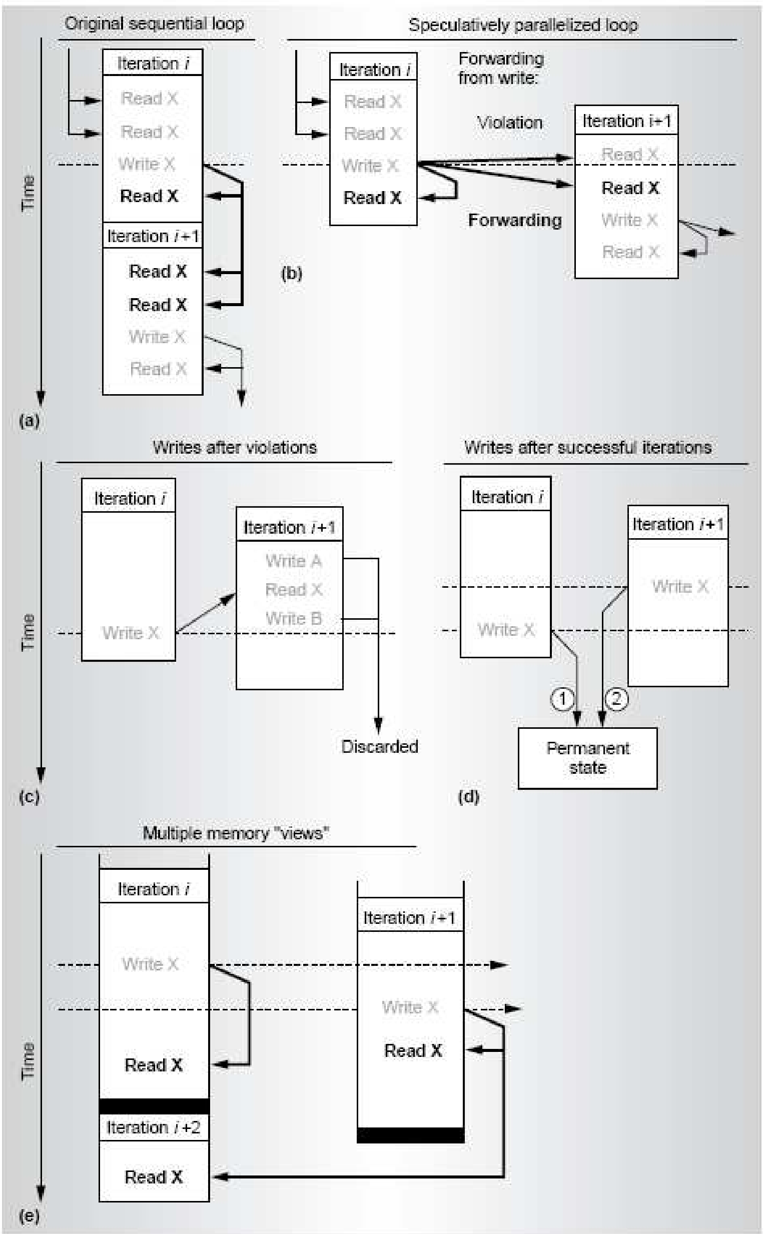


Figure 3. Five basic requirements for special coherency hardware: a sequential program that can be broken into two threads (a); forwarding and violations caused by interaction of reads and writes (b); speculative memory state eliminated following violations (c); reordering of writes following thread commits (d); and memory renaming among threads (e).

speculation) to work simultaneously. The prototype for the Hydra CMP is not yet complete, so there is much room for improvement and innovation. [84]

Relevance of topic

Many operating systems and virtual machines already exist to implement parallelism regardless of the chip on which it is running (as discussed in CSI 3310, Operating Systems). However, these require a great deal of software overhead and the performance suffers thereby. There are many elements of this overhead that can be handled by the hardware such as context switching and making the parallelism real instead of logical. However, programming models are required to handle this hardware and this usually means special parallel programs as opposed to simple sequential programs. The Hydra CMP is an attempt to better fill the hardware requirements for parallelism regardless of the programming paradigm.

This article was relevant to the lecture because it shows data synchronization such as forwarding data between parallel threads; also, the main topic of the paper is synchronization which is also a vital topic in parallel programming.

In conclusion, this article is concentrated on the Hydra CMP with hardware and software synchronization as well as speedup and shared memory. The speed up is main thing in this article. Similarity between the article and the lecture is the shared memory and thread level model; both are explained in detail.

Both the lecture and this article discussed both hardware and programming considerations. Whereas the article explains hardware support in details, the lecture focuses much more on the programming strategies.

Papers Analysis of Lecture #12b: Parallel programming models

Paper 1 Global Arrays: A Non-Uniform-Memory-Access Programming Model for High-Performance Computers

Description of Paper

This paper focuses on the description of a parallel programming approach called Global Arrays (GA). This approach combines the better features of the message passing and shared memory programming models. It is portable (like message passing) and simplifies coding (like shared memory). It also gets rid of the tradeoffs of both message passing, inability to be used for complex applications, and shared memory, lack of portability and little control over inter processor data transfer costs.

The key concept of GA is that it provides a portable interface through which each process in a MIMD (Multiple Instruction, Multiple Data) parallel program can asynchronously access logical blocks of physically distributed matrices, with no need for explicit cooperation by other processes.

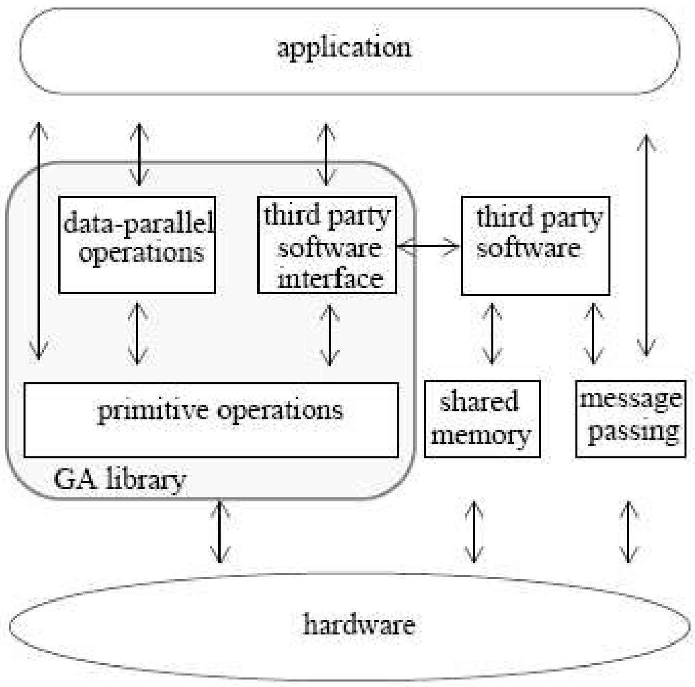


Figure 9. Structure of GA based program

Relevance to Lecture

This article is relevant to our lecture in the sense that our lecture was about parallel programming models. And this paper actually talks about a parallel programming model, the Global Arrays. Not only the GA is a parallel programming model but it was designed to be a combination of the better features of two parallel programming models that we have seen in class, the shared memory model and the message passing model.

Similarity to Lecture

The lecture focused on describing 5 parallel programming models: threads, message passing, shared memory, data parallel and hybrid. This last one is a combination of any of the above. This paper is similar to our lecture in that it briefly describes the strengths and weaknesses of shared memory and message passing models. Shared memory model is where multiple processors can communicate together by reading and writing from a shared memory which can be equally accessible by all processors in the multiple-processor computer system. Message passing model, unlike shared memory systems, is where processors communicate together by explicitly exchanging messages.
Also, it talks about the GA model which combines features of both message passing and shared memory, therefore GA is a hybrid (combination of two parallel programming models).

Differences from Lecture

There are differences between the paper and the lecture since in addition to their similarities they talk of other different things. The lecture talks about threads and data parallelism and the paper doesn't. Also, the GA model the paper talks about even if it is a hybrid model the lecture doesn't mention it anywhere in its discussion about hybrid models. Another difference would be that the paper goes into very much more details in its discussion than the lecture which gives only brief idea.

Paper 2 A Comparisons of Three Programming Models for Adaptive Applications on the Origin2000

Description of Paper

This paper takes three parallel programming models for two classes of adaptive applications on an Origin2000 system, and compares their performance and the programming effort required.

Adaptive applications are application that can adjust their resource usage to compensate for variations in the level of service that they receive. To implement them in a parallel manner is therefore a challenging task.

However, the results of the comparison using any of the three models, message passing, cache-coherent shared address space and SHMEM are comparable.

These models can be compared by implementing them either on the same system or on different ones. However, the latter choice makes direct comparison more difficult.

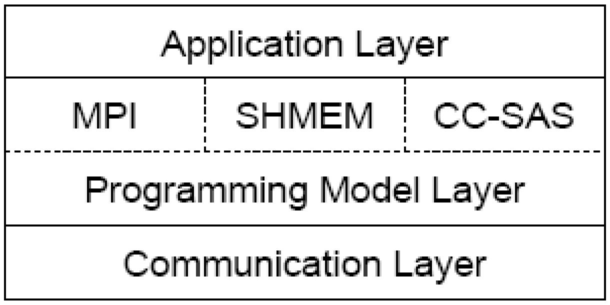


Figure 10. Layered framework for comparing different programming models

Relevance to Lecture

The relevance of this paper to our lecture comes from the fact that they both talk about parallel programming models. Our lecture focused on describing a number of them (message passing model, shared memory model, threads model, data parallel model and the hybrid model which is the combination of two parallel programming models).
The paper discusses about a number of models too, message passing and cache-coherent shared address and compares their performances and the programming effort required for each.

Similarity to Lecture

The similarity to the lecture resides in that they both talk about parallel programming models and they both describe the message passing model. The descriptions they give about the message passing model are similar (Message passing model, unlike shared memory systems, is where processors communicate together by explicitly exchanging messages).

Differences from Lecture

This paper differs to our lecture in many ways, even if they are both about parallel programming models. Our lecture only described the different models while the paper in addition to describing models also compares their performances. Also, the different parallel programming models described and compared in the paper are different from those in the lecture, except for the message passing model.

Paper Analysis of Lecture# 13: Cache Coherence Snooping Protocols

Introduction

In this document we will present two IEEE journals and we will discus and present how those two documents are related to caching coherence. Both journals been obtained from IEEE, first document is "Design of adaptive cache coherence protocol for large scale multi processor"[95] and second one is "Effects of Cache Coherency in Multiprocessors"[99].

Paper 1 Design of an Adaptive Cache Coherence Protocol for Large scale Multiprocessors

Processor caching has a very important rule in increasing system performance, today caches are not only implemented in processors, but also they exist in hard drive also to improve over all system performance. In multi processor system with private cash for each processor it is extremely important to handle and update the changes that happen in local caches and main memory, and avoid cache coherence. There are two solution proposed to solve cache coherence.

software solution and hardware solution, software solution are based on compiler analysis, mainly we will have cache inconsistency at processors synchronization points (including processor task assignment), the proposed idea is we insert invalidation instruction at each loop, those set of instruction will invalidate the affected local cache addresses, the down side of this approach is this might lead to high miss rate ratio which interns will decrease efficiency and also need a very complex compiler design .

Based on what multiprocessors scale we have, there are multiple hardware solutions been proposed. For a small scale of multi processor with single shared bus snooping protocol is an efficient way to solve cache coherence, as the number of processors increases and these processors still share the same bus, bus access will be a bottleneck, write once protocol would solve this issue and make it a good solution for medium scale multi processor system, as for large scale multiprocessor systems, we could global directory schemes.

Most of the new systems have multiple caches at different levels; this will make dealing with cache coherence more complex than one cache. Typically at each level there is a bus that connect the caches

As the system gets bigger and bigger in scale, hierarchical approach could be considered this is where multiple levels of cache and buses could be employed, as we can see from figure below every cache have a snooping control that will monitor the higher level bus and receive requests from the lower levels. The down side of this approach is a delay will be generated due to inter bus communication.

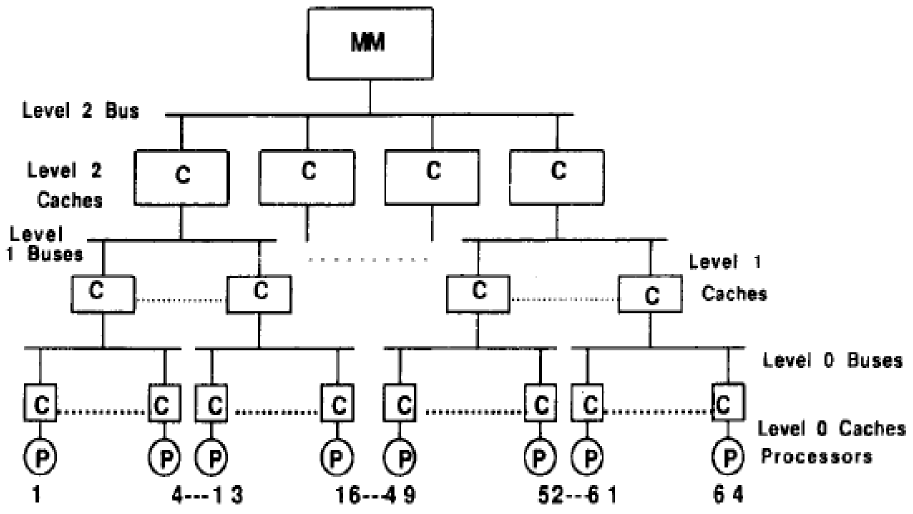
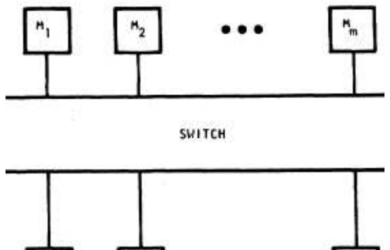


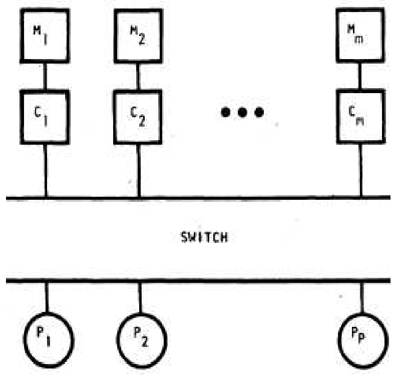
Fig. 1. A three-level 64-processor hierarchical structure of degree 4.

we conclude from the above methods that each approach have it's own benefit and down sides, we also conclude, for a large scale multiprocessors systems we will have to implement hierarchical structure buses, we can either have multiple copies of data across caches (high performance but lots of overhead) or we could store one copy only (low performance due to memory miss but very low overhead), this document presents new protocol where we can combine both ideas, the idea is to combine processors into clusters and treat them as a medium scale multi processors system, on each cluster we will have single local copies of synchronization variables, and for the whole system will have different set of caches. The idea of partially shared memory will guaranties high performance by decreasing memory access miss, and also will decrease the overhead work to be done in order to ensure consistency.

Paper 2 Effects of Cache Coherency in Multiprocessors

The use of caches in single processor system proved that it increase system performance and throughput significantly, this issue has promoted their use with multi processor system, there are two types of caches private and shared caches, private cache is only accessed by the processor associated to it, while shared caches are accessed by all processor.





Shared cache Private cache

Since all processors have access to all shared caches, Cache coherence on shared cashes are restricted between the actual cache and the main memory only, since there will be only copy on any of the caches, in private caching, cache coherence will exist between caches themselves and the shared memory. This will make the design of multi processor system with private cache more complex.

The idea of cache consistency in shared caches multi processor system is to make a processor own the data before it can modify it, to enforce that we will set two level of access to each data block, read only and read write, when a processor fetch a data block it will have read only (RO) level of access, data block can be read by the processor, when the processor wants to write to the data block it will first has to make sure that no other processor have read write (RW) level, only one processor at a time have RW or RO property to any data block. To implement that kind of approach global table will be established for all processors to consult to invalidate their data blocks.
When a processor wants to copy to a certain data block there will be two cases where this processor will have to wait, first is when there is another processor have RO right to that data block, second is when there is another processor have RW right to that data block, the later case will take more waiting time, since write back to the main memory have to done to before the processor can copy it.

As for cache consistency in private caches multi processor system the document where implementing a very complex algorithm and set of flags few compliers that implemented that approach, and that due to the date of the document, new approaches been presented which have more efficient way to handle this program, such approaches is cache snooping.

The idea behind cache snooping is to make each cache update their data blocks whenever a processor write to its private cache, when processor wants to change its local copy of data it has to write back to the main memory and since all caches share the same bus, once the a processor do a write back, the rest of the caches will update their copies.

This algorithm is considered very powerful for small and middle scale multi processor systems. The reason it’s not very suitable for large scale multi processor systems is because bus system will eventually become a bottleneck.

We conclude from this document that cache coherence algorithms been improved dramatically in the last twenty years, and that’s due to the fact that multi processor systems are being more widely used then before.

Paper Analysis of Lecture #14a: Cache Coherence Directory Based Protocols

Paper 1: Cache coherence for large scale shared memory multiprocessors

Description of paper

In this paper the author compares a full map implementation of the centralized directory protocol with a chained directory implementation of a distributed directory protocol. The paper analyses the overhead costs of each directory as well as their performance.

The text states that when using a full map directory, each memory line must have present bits associated to each of the N caches in the system; therefore the memory cost of such a system is O (M N) where M is the size of the memory. On the other hand in a limited directory or a distributed directory, the memory cost is greatly reduced.

The distributed directory protocol uses a linked list in which each node contains a pointer field. The pointer field is used to point to the next cache in the list. In this type of system the memory cost for the pointer is log(N) therefore the overall memory cost for the system becomes O (M (1 + (Nc/Nm) log(N)) where M is the total size of the main memory, c is the size of cache, m is the size of each node and N is as before the number of caches in the system. Furthermore, if we assume that Nc/Nm remains constant, we obtain the overall memory cost O (M log(N))

In this paper the author gives a special attention to the traffic required over the interconnection network in order to ensure coherence amongst the different caches. The paper refers to a general configuration in which every node is comprised of at least one processor, a cache and a network controller.

The paper explains how and why the distributed directory protocol can solve the bottleneck problem which occurs when a centralized directory is used. This problem can occur, for example, when a write miss occurs. In such an event the state of the cache line of every cache containing a copy of the shared variable must be established, this is done by checking the centralized directory, in the event that a cache line is dirty, it must be flushed before the correct data is sent. While this process is happening the memory line must be locked therefore other incoming messages must be either buffered or bounced back.

Since, in a distributed directory, the information about which cache contains a copy of a shared variable is distributed among the cache line, servicing different requests does not require the locking of resources as is the case in centralized directories. This practically eliminates the risk of the bottleneck problem.

The paper also explains how the resource utilization is more efficient when a distributed directory protocol is being used. The protocol described in this paper does not assume that the interconnection network preserves the order of different messages; this allows adaptive routing and therefore increases the network's performance. Also, by allowing the messages to be buffered, combined with the out of order processing of messages, this protocol makes it possible for the processors to proceed without having to wait on resources.

To demonstrate the efficiency of this type of protocol the author uses different speed ratios between the SRAM and DRAM in the system. When tested on the centralized directory, the slower DRAM resulted in a slower execution. However, when the DRAM speed was reduced in the distributed directory system the effect was negligible. This is due to the fact that in the distributed directory, most data transfers are cache-to-cache therefore, since there is no centralized directory, the bottleneck problem does not occur. This in turns means that the main memory can be slower and cheaper without affecting the overall performance of the system.

The paper concludes that for large scale shared memory systems, a distributed directory is more appropriate.

Relevance of the Paper

The topic of this paper is very relevant to that of the lecture. This paper deals with both the centralized directory and the distributed directory protocols. The major difference between this paper and the lecture is that in this paper the emphasis is on explaining why a distributed directory is better in a large scale shared memory multiprocessing system. In other words the paper is concentrated on one particular situation and the lecture was intended to make us aware of the tools at our disposal to solve the coherency problem in any type of system; the lecture did not restrict itself to large scale systems.

Nevertheless this paper is very helpful in understanding the differences in the implementation of the centralized directory protocol versus the distributed directory protocol. The distributed directory protocol studied in this paper uses a chained directory. It is explained that this type of directory is most efficient in an environment with frequent write operations. This is attributed to the fact that when write operation is performed and a chained directory is being used we must wait to invalidate every cache in the list. If we have frequent write operations, the sharing list does not have time to grow to a large size.

This paper also brings to mind the speed of the interconnection networks as well as that of the memory being used. The paper explains that in a distributed directory protocol the majority of the traffic occurs between caches and therefore a slower DRAM does not impede performance as much as if a centralized directory was being used. The author also discusses the advantages of buffering the messages before transmission over the interconnection network.

This paper describes the subjects discussed in the lecture and goes the extra mile to compare the different protocols. This paper is very useful in understanding the advantages and disadvantages of the different directory protocols at our disposal.

Paper 2: An Evaluation of Directory Schemes for Cache Coherency

Description of paper

This research paper proposes that directory-based protocols are a sufficient implementation for cache coherency and more efficient in large-scale multiprocessor systems than snoopy protocols. This paper was created in 1988 when snoopy protocols were more popular than directory protocols.

In this paper, there were four different cache coherency protocols tested; Write-Through-With-Invalidate, Dragon, the Archibald and Baer scheme and a no-broadcast directory-based scheme with one pointer per index. The Write-Through-With-Invalidate protocol is a snoopy protocol that uses the write-through policy. The Dragon scheme uses an update protocol. Update protocols preserve consistency by updating the out of date cache data with new data. Writes are broadcasted on the bus to the cache where it determines whether or not the data is shared. At the time, the Dragon protocol was often considered having the best performance of snoopy cache protocols. The Archibald and Baer scheme is a directory-based protocol. This protocol uses broadcasts to complete invalidate and write-back requests. Even if this protocol uses broadcasting, it does not use as much broadcasting as snoopy protocols. The directory of an Archibald and Baer scheme are simplified where there are only two bits to specify whether the data in the shared memory is uncached, clean in one block, clean in multiple blocks or dirty. The final scheme tested is a directory scheme without broadcasts. This scheme allows data to be held in a single cache at once by allowing only one pointer per index. The directory holds a pointer to the cache where the block is contained. This directory is abbreviated to $\mathrm{Dir}_1\mathrm{NB}$ in the paper to signify a *dir*ectory-based protocol, *1* pointer per index and *N*o-*B*roadcasting.

These protocols were chosen because they represent the extremes of both the snoopy and the directory protocols. Write-Through-With-Invalidate is considered having lowest performance in snoopy protocols where the Dragon is considered having the best performance. The $\mathrm{Dir}_1\mathrm{NB}$ directory protocol is the simplest directory-based protocol and is scalable to large multiprocessor systems due to its simple consistency scheme. The Archibald and Baer scheme is opposite to the $\mathrm{Dir}_1\mathrm{NB}$ scheme in its complexity and scalability.

It is shown through the tests that the Write-Through-With-Invalidate and $\mathrm{Dir}_1\mathrm{NB}$ are inferior to the Archibald and Baer and Dragon. For the snoopy protocols, updates are less costly than the invalidation and the following miss. For the directory protocols, $\mathrm{Dir}_1\mathrm{NB}$ allowing a block to reside in a single cache creates high read misses that hinder performance compared to the Archibald and Baer. In these tests, a four processor multiprocessor system was used. The results showed that the average of bus cycles per transaction used in snoopy based protocols were less than that of the directory-based protocols. However, these directory-based protocols require fewer transactions and proved to be almost as good as snoopy protocols in the four processor system.

Since snoopy protocols do not pass the bandwidth restraint imposed on small-scale multiprocessors using a single shared bus to memory, they are sufficient in this case in providing cache coherency. However, when more processors are added, the scalability of the snoopy cache scheme decreases because of the broadcasts used by snoopy protocols can congest the interconnection network in the multiprocessor system. Incrementing the bus speed to keep up with broadcasting is insufficient since most consistency protocols rely on a low-latency broadcast, which in turn makes hardware implementation of snoopy protocols in large-scale multiprocessors disadvantageous. Snoopy protocols also hinder the connection between the processor and the cache since the protocol needs to check the coherency in every cache stopping the access from the processor.

Using directory protocols is more beneficial in large-scale cache coherent multiprocessor systems. In the paper, quantitive data was unavailable indicating directory protocols as being a sufficient means for cache coherency in a large-scale multiprocessor system. However, the tests done for the paper show that these protocols are scalable. The basis is that by proving these protocols are scalable will show that they are superior to snoopy protocols in large-scale multiprocessor systems.

Similarities to the lecture

- The snoopy protocols' limited scalability leads to the need of directory based protocols for cache coherency in large scale multiprocessor systems. This is caused by the broadcasting method used by the protocol which creates congestion in the interconnection network. The paper goes further by explaining the result onto the cache to processor connection and how it is hindered by broadcasting (cache can only be accessed by one component at a time).
- Directory protocols are scalable and therefore better for large-scale networks. Directory protocols can stop the previous problem by validating or invalidating only the caches which contain the block.

Differences from the lecture

- The specific protocols used in the paper and the specific protocols in the lecture are different. In the paper, the directory protocols introduced in the lecture were not even acknowledged.
- The categorization methods used in the lecture is different. The paper actually suggested a method created by the authors of naming policies.

Relevancy to the lecture

This paper shows proof of the scalability of directory-based protocols as discussed in the lecture. The tests show that directories are better used than snoopy for larger scale multiprocessor systems and have roughly the same performance for the smaller-scale systems. A directory policy's ability to communicate to only those caches that need updating make a directory scheme a wise choice for large systems. The paper goes into detail on why by using percentages of write misses/hits, read misses/hits, etc. and assigning penalties. Afterwards, the penalties are added so that the higher the penalty, the more time is taken to perform the instruction. This is a great article for supplementary information on directory and snoopy protocols, how they can be implemented, how they can be tested and when to use either protocol.

Paper 1 Cache Coherence for Shared Memory Multiprocessors Based on Virtual Memory Support

By Karin Petersen Kai Li

6.1 Description of paper

This paper discusses a cache coherence protocol that uses virtual memory. The architecture consists of a processor, a cache, a memory management unit (MMU), an interconnection network and the virtual memory. The MMU manages the virtual memory system by giving each process its own page table and translation look-aside buffer reload. Figure 10 shows this architecture.

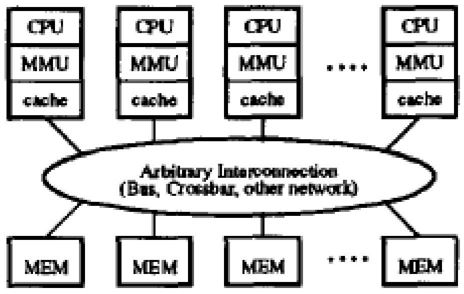


Figure 10: VM architecture

They elaborate on three models: the sequential consistency, the relaxed consistency and release consistency models. In the sequential consistency model, all processors start in the nil state. Once a processor reads a certain page in the page entry table (PTE), it loops through all the processors and tries to find if someone is writing to that page. If so, both processors have to go into the read state else only the reading processor goes to that state. When a write occurs, it loops through all the processors and tries to find if someone is reading that page. If so, the reading processors have to go into the nil state and the writing processor goes to the write state. Figure 11 shows those transitions.

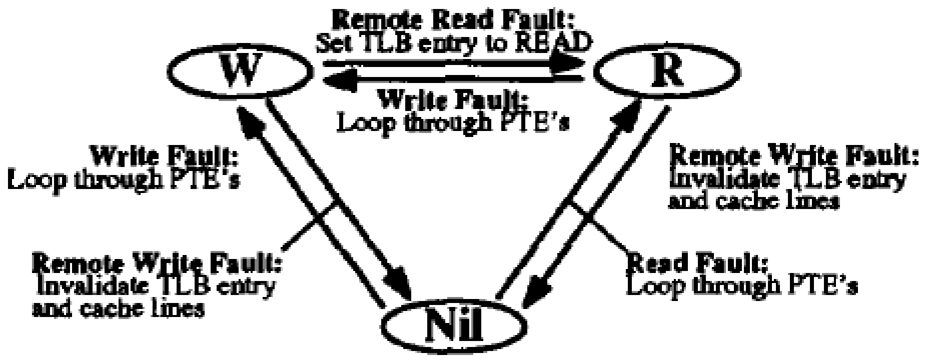


Figure 2: Transition diagram for SC.

Figure 11: Sequential consistency model

In the relaxed consistency model, the memory can provide access to processor in any order and the synchronization access will ensure the correct use of the data. This reduced access time to data structure and reduce page fault. The release consistency model focuses on two types of synchronization access: acquire and release. This model state that an “acquire” must be performed before a load or store access and a load or store access must be done before a “release” access.

6.2 Relevance of the topic

Using virtual memory protocols provide another way of implementing cache coherence in a system. This approach is economical because it doesn’t require using a lot of hardware and easy to change. When there is not a lot of memory contention this protocol can perform well. But where there is, it slower then the directory based and snooping protocol. The writing and reading procedure are almost the same for the sequential consistency model and the directory based directory because when a processor write to the memory it invalidate the rest of the cache in the other processor that contains a copy of the data. There also a similarity with both models having tables. The directory base has directory and virtual memory has page entry table.

Papers Analysis of Lecture #15a: Message Passing Models

Paper 1: MPI: A Message Passing Interface.

Description of the paper:

The paper [119] is the result of deliberation between the members of the MPI forum which is constituted by the meeting and the email discussion between the members of MPI working group. We will mention how this group was formed.
The paper presents an overview on MPI. The advantage of this standard which deals with distributed system are: the portability, and ease-of-use. These advantages are achieved by the use of clearly defined set of routines that can be used on all stations or nodes of a distributed system. In addition of the functionality that the wildly-used message passing systems provide, MPI is characterized by its flexibility and usefulness. The standardization process started in the workshop on standards for message passing in a distributed memory environment, help in April 1992. After discussing the basic features essential to a standard message passing interface, a working group established to continue the standardization process. The MPI standard was presented for the first time in Supercomputing conference in November 1993.

MPI is a standard message passing interface runs on MIMD distributed memory computers network. Even though MPI does not support explicit support for multithreading, it can be implemented efficiently for a multithreaded environment.
The concepts of process groups is defined as an ordered collection of processes where each process has a unique ID within the ordering called rank, this ID range between 0 and number of process-1. The use of groups is important in the collective communication that will be mentioned later; also they are important in task parallelism where different groups perform different tasks. If different programs are used into different groups we call this MIMD, if each group executes its own code we call this SPMD. The processes cannot be created and destroyed, but the group can, so one process can be a member in different groups at the same time. A new group can be created by listing the rank of the processes of the parent that form the new group, or by partitioning a group using a key. Other routines are used to perform synchronization using a barrier, inquire about the size of the group and the rank of the calling process in the group. A communicator is used to define the extent of a communication operation. Inter-communicator binds a context and a group together; Intra-communicator binds a context and more than one group together. The communicator object is passed as a parameter to many routines.
MPI provides an explicit support for general application topologies such as graphs and Cartesian, also for no topology groups.

Point-To-Point communication:
The message in P-T-P communication is characterized by: the source process, the tag of the massage and the communication context. The source and the destination are specified

by the group and the rank. The intra-communicator bound together the group and the context, while the inter-communicator different groups are bound together with context. There are three communication modes for sending a message: in the standard mode it is not necessary that the receiver is ready to receive the message, the message is delivered whenever the receiver asks to receive. In the ready mode, the receiver process should be ready to receive before the sender sends the message. In synchronous mode, the send operation does not return until the receiver receives the message. The other criteria for classifying the send routines are: blocking VS unblocking mode. In blocking mode the routine will not return until it is safe to reuse and modify the application buffer that is used to buffer the message to send it. On the other hand the unblocking routine returns immediately. Blocking and unblocking are considered the two main modes used for the receiving.

Collective communication:
The collective communication routines are responsible to coordinate the communication among a group. All the processes inside the group should call the collective routine. When a process has completed its task it may continue with other tasks. To prevent that, we can use barriers to synchronize the processes. All the collection communication routines are blocking. There is two main types of routines:
Collective data movement routines: The three types in this case are:

- Broadcast where the data is sent from one process to all other processes in the group.
- Scatter: distributes distinct messages from a single source task to each task in the group.
- Gather: Gathers distinct messages from each task in the group to a single destination task. This routine is the reverse of the scatter operation.

Global computation routines: A function should be specified to be performed. The two main types are: reduce and scan.

Relevance to lecture:
Although not all the advantages mentioned in the lecture are explained here, but the paper talked about the portability as the main advantage. Also the paper explains the concept of groups and communicators more detailed than the lecture.
The collective communication concept is defined in the lecture as a special type of communication, the same idea was found in the paper but in other words when it mentioned that a collective communication can be implemented using P-T-P routines so it is a special case of the P-T-P communication.

Difference from lecture:
The main difference with the lecture is that the lecture classify the blocking and unblocking modes of communication as the only criteria of P-T-P communication modes, but in the paper another criteria is introduced, this criteria has three types: standard, ready and synchronous. So using these two criterias a send communication can have six modes. The paper mentioned that in the collective communication, a routine is always blocking while the lecture didn’t mention that.

Papers Analysis of Lecture #15b: Message Passing Models

Paper 1 Tutorial

The first paper analyzed was a tutorial provided by LLNL. The following is a summary of the tutorial and its relevance to the lecture topic.

The tutorial looked at key areas of Message Passing Interface (MPI). It first described MPI and its function. It elaborated on the environment that MPI is best suited for. It talked about groups and communicators, an integral feature in MPI. Finally, it discussed some of MPI's virtual topologies.

Message Passing Interface (MPI) is a specification for developers who use message-passing libraries. It provides a specification on what a message-passing library should contain and how it should function. It has been developed for both C/C++ and FORTRAN. It is supported by almost of the HPC platforms. It is very portable and does not require for code modification when porting an application to another language. It exploits individual hardware features to provide great performance. The MPI library consist of over 125 routines which provide both vendor and domain specific implementation.

The MPI Model is a perfect implementation of a distributed memory parallel programming model. It uses a series of messages to communicate with different nodes and each node has private memory reserved specifically for message passing. Parallelism is explicit, therefore the programmer is responsible for implementing the parallel methods. The programmer must initially identify the areas of the program that can be processed in parallel. Once these areas are determined a static number of tasks is sent to each processing element. No new dynamic tasks can be created.

The following is the graphical depiction of the MPI programming structure.

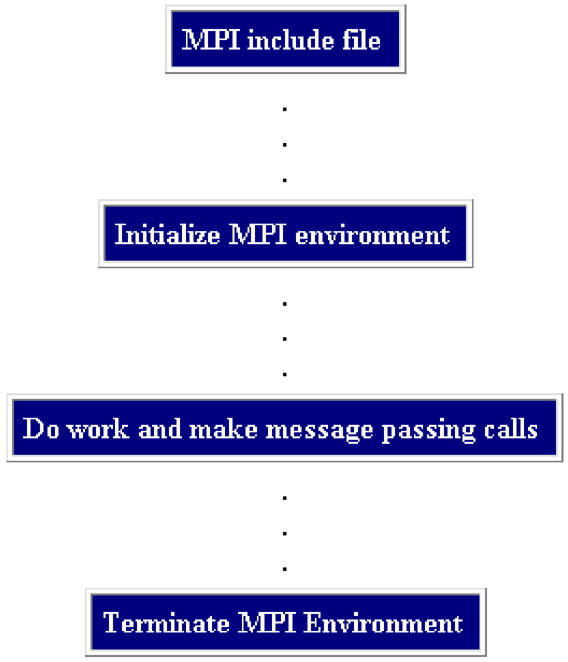

MPI include file
Initialize MPI environment
Do work and make message passing calls
Terminate MPI Environment

Figure 6 – MPI Programming Structure [125]

In the MPI environment, communicators are predefined for all MPI processes. They are used to communicate with other processes inside the group. The group is a set of processes that have a rank, which communicate with each other. A group is represented in memory as an object and accessible using a handle. A groups is always associated with a communicator. The processes are created before the MPI initialization phase. During the initialization phase they are assigned a specific number. Furthermore, since dynamic creation of processes is not possible, each process knows about every other processor and communication is allowed only between these set number of processes. When viewing this from the programmer level, groups and communicators are represented as one. The programmer must specify the communicator of the group.

MPI uses point-to-point communication to send messages between processes. Usually, one process is sending while the other is receiving. Buffers are implemented by the system, ergo the programmer need not worry about it. The has a buffer to store the data that cannot be received by the other process element. Once the process retrieves the data from the system buffer it deals with the received data accordingly. This type of buffer is known as a system buffer and is managed by the system; there is also a user buffer that must be managed by the programmer.

MPI has defined number message passing routines. The first being blocking message passing, where the flow of the program will only continue once the program notifies the process that it is safe to do so. When using non-blocking message passing, the program will continue on almost immediately after the message is sent. Order of messages is guaranteed with MPI, therefore no message sent out can overtake another; the order in which they are sent out is in the order in which they are received. Unfortunately, MPI does not provide any fairness measures therefore the programmer must assure that starvation doe not occur.

MPI processes ordered and mapped in a specific way to form shapes that are are conducive to message passing. The grid and graph models are two topologies that are supported by MPI. They are built upon the concept of communicators and groups. The developer must program these topologies. They are used out of convenience in arranging groups and communicators but also for their efficiency of communication.

This tutorial provided by LLNL is perfect for anyone who wants to explore the world of MPI. The tutorial gives more detail to the lecture topic as it provides some definitions to some key terms of the Message Passing Interface. It also provides a nice collection of routines and their uses and functionality. It rounds that out with some nice programming example that will help you get the handle of message passing at either a C or Fortran level. This site is the perfect follow up to Dr. Bolic’s lecture and will provide the reader with an excellent knowledge of MPI and its uses.

Paper 2 MPI Communicator

In this paper the authors described an extension to the Message Passing Interface that would generalize the concept of the MPI communicator. A communicator is a group of processes that may communicate with each other. The extension proposed in the paper was to extend the original implementation of communication between these communicators.

MPI communicators are referred to as local communication objects (LCOs). It is noted that these cannot be directly manipulated. It is suggested that these be transformed into an MPI data type such that it would become a generalized "communication port". Then, with further extensions to the MPI, by adding routines, one could control these communication ports. They could create, edit and destroy these LCO's for a given process. This would give us more flexibility than the current implementation because with MPI, the communicators (or LCOs) can only talk with one other port. With this improvement the LCOs will have multiple communication ports. The LCOs would be able to talk with many other LCOs. This could then help us create a wide variety of virtual network topologies as pictured bellow:

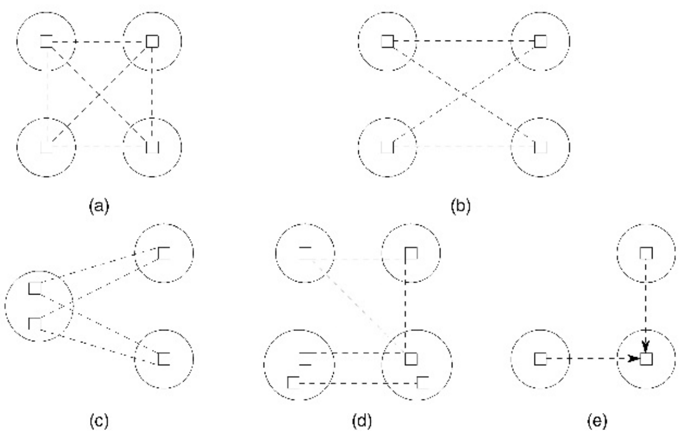


Fig. 1. The two types of commuincation structure can be specified in MPI: (a) the fully connected communicatior and (b) the intercommunicator's bipartite graph. Three different structures that can be specified by using MPI extended to support ports. (c) A fully connected communicator with more than one LCO per process, (d) a regular communicator coexisting with a dynamically created communicator connecting two LCOs, and (e) a communicator structure that allows two senders to communicate with a single receiver.

Figure 7 – Communication Structure [126]

The paper describes the syntax and routines it will use to create this extension. It also describes the series of events of how communication will occur between these new LCOs. It also goes a great way to show that while implementing new functionality they have maintained backwards compatibility.

This paper is pertinent to our lecture because it provides an efficient and working extension to MPI to allow us to create the network topologies that we studied in previous lectures. Originally, MPI would have struggled and not been able to provide us with all the functionality to create these topologies. However, now with the extension one can take message passing to the next level. Creating a flexible system with multiple communication ports with little to no overhead compared to the original MPI implementation.

Papers Analysis of Lecture #16a: Message Passing Architectures and Routing

Paper 1 Impact of Virtual Channels and Adaptive Routing on Application Performance [134]

This paper performs an impact analysis of virtual channels and adaptive routing for shared memory and message passing systems. This paper claims to be one of the first papers written that provides analysis of performance for real world applications that utilise parallel architectures instead of using models of traffic flow (i.e. tornado) that may not be realistic for real world applications. The conclusion of the paper is that while performance gains can be seen when using virtual channels and adaptive routing, the gains aren't significant enough to warrant the cost of implementing these methods. Rather, more effort should be concentrated on improving network bandwidth and network interfaces, where there is more room for improvement.

To obtain better routing speed, the authors have identified several areas that should be concentrated on for improvement including simple designs, fast designs, few or no virtual channels, simple routing schemes, fast message-header decode times, small crossbar sizes, fast channel interfaces (i.e. putting flits on and removing flits from the channel), and multiple channels.

The lecture described the different metrics that are used when looking at interconnection network performance. A simple example was presented that provided some context on how the metrics are obtained and used. The example provided a way of calculating these metrics for deterministic and oblivious routing techniques, but not much information was given for adaptive routing techniques due to their complexity. This paper takes the ideas learned in the lecture and expands on them, going into detail on measuring performance for adaptive routing techniques. Also, instead of using a model for interconnection network traffic that may be an over-simplification of actual application network traffic, this paper uses realistic traffic flows to evaluate the performance of the adaptive routing technique.

The lecture noted that one of the most important aspects of determining accurate metrics for interconnection networks is the traffic flow pattern. The paper also stresses this as one the main part of models used in previous studies that was over-simplified. To remedy this, the authors will use network traffic generated from real world applications in their model. The network traffic patterns generated from five different applications were input into the model for shared memory and another five for message passing. Also, the authors of the paper use a two-dimensional mesh, which is a network topology that has been discussed in previous lectures, for their interconnection network.

To properly model the network traffic encountered by an interconnection network, the authors identify three metrics of importance. These metrics were not presented in the lecture. The metrics introduced are the “message interarrival time distribution with the corresponding generation rate, the spatial distribution of messages, or the traffic pattern and the message size or volume”. The authors go on to state that “typical synthetic environments have assumed that the temporal distribution is exponential, the spatial distribution is uniform or a few select localized communication patterns, and the message size is a fixed number of bytes”, but these assumptions limit the applicability of the results obtained from these studies.

The system that they are using as a model for shared memory is cache coherent non-uniform memory access (CC-NUMA) architecture, using invalidation based coherency scheme with a full-map directory. The message passing model uses processors implemented using MPI.

The switching technique used in the model are 5x5 crossbar switches, four of which are used to connect inputs to outputs and the fifth used to connect the processor at the node. The switches support one or more virtual channels per input/output and each virtual channel has one or more flit buffers. The model network is synchronous and utilizes fair arbitration between competing virtual channels.

The paper takes two routing algorithms discussed in the lecture, oblivious and adaptive, for the study. Adaptive algorithms were only really defined in the lecture. This paper uses Duato’s fully adaptive routing algorithm, which is supposed to provide deadlock free adaptive routing for wormhole networks.
The paper also discusses the impact of virtual channels on application performance. Although virtual channels had not been presented in the lecture, or any previous lecture, it was presented in a subsequent lecture.

Paper 2 A Survey and Comparison of Wormhole Routing Techniques in Mesh Networks [135]

Multiprocessor architectures are becoming increasingly popular with the demand for higher processing in engineering and scientific applications. Multiprocessor systems consist of a series of processing elements connected together through an interconnection network. These processing nodes must communicate amongst each other. One method of doing so is by using message passing. Messages are passed from one node to another through the interconnection network. In order for the system to perform at a high level, this communication must be done as efficiently as possible.

There are many different methods for routing messages through the interconnection network, and it is difficult for a designer to choose the best one. One of the main routing techniques is referred to as wormhole routing. The purpose of this paper was to provide an introduction to some of the wormhole routing techniques that exist for a mesh network. The paper does not go into great detail about any of the techniques. It is meant to provide an overview and to introduce some of the options that exist when choosing a routing algorithm.

The interconnection network that connects the processing elements in a multi-computer network is characterized by its topology, switching, flow control, and routing. There are many different possible topologies to use when designing a network. These include mesh, hypercube, and so on. Only the mesh is considered in this article. A mesh topology is important because it is very scalable. Switching refers to the technique used to move data around in the network. Flow control refers to how buffers, channel bandwidth, and other resources are allocated, and how packet collisions are resolved. The resolution of a packet collision could be buffering a packet, dropping a packet, or re-routing a packet. Finally, routing involves determining a path for the packet through the network. This article deals with wormhole routing.

Wormhole routing is a technique that involves advancing a message immediately from the incoming to the outgoing channel. Other techniques involve buffering the packet until the whole packet is received, but wormhole routing will forward the packet as soon as the outgoing channel is vacant. A packet is subdivided into flits (flow control digits). The first flit in the packet includes the routing information and is referred to as the header. All other flits follow the header as it travels through the route. Each channel has a flit buffer to store the flits. If the header flit encounters a blockage on the network, the flow control of the network will cause the flits to be held in the flit buffers along the established route. They will not be removed from the channel as in virtual cut-through routing. The advantages of wormhole routing are that the latency to send a message is reduced (as long as there are no blockages) and the buffers requirements of a router are much smaller. The disadvantage of this technique is that by buffering the flits along the channel, more resources are being hogged and there is an increased probability of deadlock.

There are many different variations of wormhole routing that could be used in a mesh network. The techniques described in this article include: Dimension-Ordered Routing, The Turn Model, and the Node Labelling Technique. A big concern when routing packets is to try to avoid deadlock situations. These algorithms all have unique strategies for avoiding this unwanted scenario.

Dimension-Ordered Routing involves routing the packet in one dimension at a time. The packet will be mapped in one dimension until it reaches the correct coordinate, and then will be mapped in the other dimension. For example, in a mesh network, the packets will be routed along the X direction followed by the Y direction. Enforcing this strict routing order helps to guarantee deadlock free routing.

The main idea behind the Turn Model is to try to prohibit the smallest number of turns such that cycles are prevented. The west-first routing algorithm involves disallowing west turns. So, if a packet needs to travel west it must do so initially. By enforcing this rule, cycles are not possible and therefore this routing technique is deadlock-free.

The Node Labelling Techniques involves identifying nodes that could cause routing difficulties. Nodes that present a potential routing problem, such as those connected to faulty nodes, etc., are marked as deactivated. Packets will not be routed to a node that is marked as deactivated. Packets are routed through the healthy parts of a network in an adaptive manner, and are routed around faulty regions.

This article evaluates these routing techniques, along with some others, based on such criteria as whether they are progressive or backtracking, minimal or non-minimal, or completely or partially adaptive. Progressive techniques have limited ability to back up from a node that has been reached, while backtracking techniques search the system systematically and backup as necessary. The difference between minimal and non-minimal techniques is that minimal considers only profitable links for routing a packet, while non-minimal considers profitable and non-profitable links. Non-minimal can lead to a packet being routed on a longer path in order to avoid network congestion. A fully adaptive technique can use all paths in its class, while a partially adaptive technique will not allow all messages to use any shortest path.

This article provided a decent overview of several different routing algorithms. It did not provide great detail for any of them, but the goal was to provide an overview and provide some insight for a designer trying to choose what type of routing algorithm to use. It also elaborated on certain topics covered in course lectures concerning wormhole routing and interconnection networks.

Paper Analysis of Lecture #16b: Message Passing Architectures and Routing

Paper 1: Adaptive Routing Algorithm for Lambda Switching Networks [135]d

Description of the Paper
In the research paper *Adaptive Routing Algorithm for Lambda Switching Networks* published by S. Sartzetakis, C. I. Tziouvaras, and L. Georgiadis, optical networking technologies and architectures are examined. They focus their research on routing algorithms and the performance in optical networks, which place emphasis on the effectiveness and function of an adaptive routing algorithm compared to other routing algorithms.

The interest in using optical components to build a network comes from the expected cost-effectiveness and scalability of optical switching fabrics compared to electrical switching fabrics when scaling to several thousands of ports. [135]d

Relevance to the Lecture
In what the research paper refers to as "fixed routing" is the same definition as given in the lecture notes [135]f as deterministic routing. That is, there exists only one path between a source node and a destination node, which was pre-calculated to be the shortest distance between the nodes.

In "fixed alternate routing", a set of paths for each source-destination node couple are taken into consideration. For one node couple, there exists at least two paths, where the primary path was calculated to be the shortest path and the secondary path was calculated to be the second-shortest path and so on and so forth. If the primary path is not available (blocked due to traffic load), then the next available path in the set is used. This is routing algorithm is synonymous with the oblivious routing algorithm defined in our lecture notes.

The Proposed Adaptive Routing Algorithm
The adaptive routing algorithm examined here using called the "fixed paths least congested algorithm" and its application in dense wavelength-division multiplexing optical networks is studied. In this algorithm, the load on a node link determines the state of the network. For each available node link, number of allocated wavelengths (resources on an optical network) on a node link called the "cost" of the link is summed and the

node link with the least cost is chosen. The cost algorithm applied on this network was the “lexicographically optimal allocation”. This cost algorithm is a type of min-max allocation algorithm. An example from the research describes the lexicographically optimal allocation algorithm:

“Assume that four wavelength paths must be chosen from source node A to destination node H. Notice that the min-max allocation (Figure 7) selected two routes and totally six links have two allocated wavelengths. In the lexicographically optimal allocation (Figure 8) four routes are chosen and only two links have two allocated wavelengths. Intuitively the lexicographically optimal allocation leads to a more balanced network.” [135]d

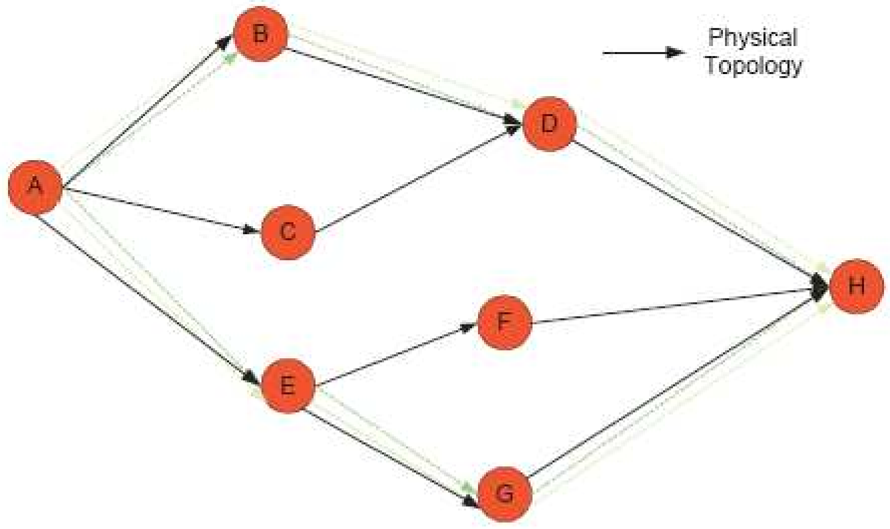


Figure 7 Min-max allocation

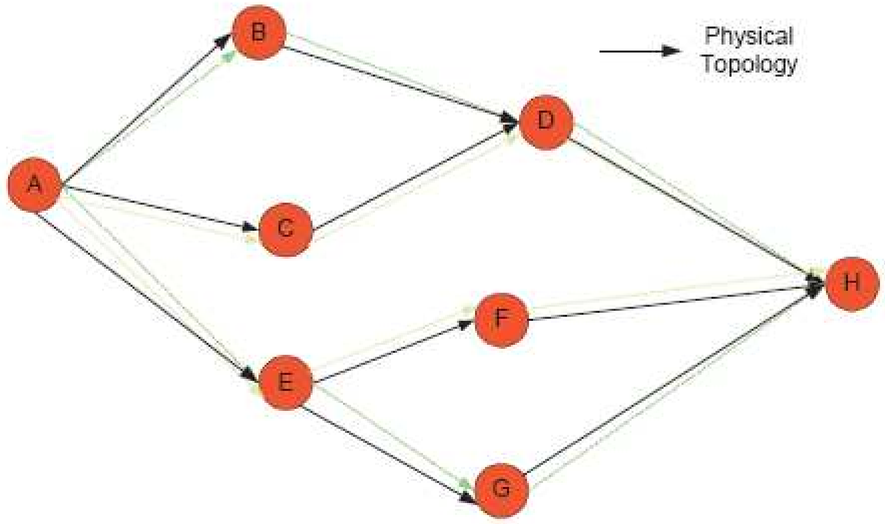


Figure 8 Lexicographically optimal allocation

In order for the network to maintain information on the state of the nodes across the network, each node maintains a routing table to base its routing decisions. This way, the whole network can adapt to the load from each periodic update of the routing table information.

Summary
Although the adaptive routing algorithm was assumed to be optimal routing method in certain dynamic and static traffic scenarios, the cost of computing the vectors and handling the dynamic routing can be more difficult to implement. The research paper recognizes that the difficulty arises when constraints include making the approach the simplest and quickest (to handle requests) in implementation. This conclusion echoes the suggestions put forth by Dally and Towles [135]a and as described in the lecture notes on routing [135]f that although deterministic "fixed" routing is a simple implementation, an adaptive routing algorithm such as the "fixed paths least congested" algorithm can provide a more optimal routing performance based on network traffic and the state of the network. An adaptive routing algorithm is more difficult to implement than "fixed" routing", but it has its advantages in application of optical networks.

Paper 2: Fault Tolerance of adaptive routing algorithms in multicomputers [135]e

Description of the Paper

This paper shows that adaptive routing algorithms better serves the purpose to tolerating failures in systems with medium to large granular communication. Adaptive routing is a method of routing, which gives several options of paths to route and then chooses the optimal path based on the state of the network.

The performance loss and the cost are major factors for weighing one method over the other. Here performance loss is analyzed by taking into account several different types of failures in systems. The effects of link failures, node failures, varying Bandwidth message time and effects of synchronization is analyzed through simulation on 3 different models. The adaptive routing algorithm used here is based on routing table since it uses only two alternate routes between source and destination.

Whereas when a link fails, there is a loss of communication between the two nodes involved. Adaptive routing algorithm makes sure that when there is a failed link, routing tables gets updated with the current state and the spare nodes are connected to the system, thereby the messages can be routed through the new spare nodes. The experiments proved that the performance degradation was directly proportional to the frequency of communication. If there was a loss of 10 links, the system lost 25%of available links. In model 3, synchronization was performed in each step resulting in very high loss of performance. The synchronization considered in model 3 is 100%.

When a node fails, the links at that specific node and the node's processor loses all of its functionality. When there are node failures the spare node replaced and functioned on behalf of the failed node. For model1 and model 2 the performance degradation was less than 10%. But model 3 had large performance degradation. In the experiment conducted, there is a loss of 4 nodes. It is equivalent to loss of 16 links. Here performance degradation is equivalent to losing large number of links.

Bandwidth and message completion time are used to measure performance degradation. In model 1and 2 had random communication and were asynchronous. If bandwidth is a given constant value and the network is not saturated, then message completion time is very high for link failures. At a given message completion time, if the network is almost saturated then the loss in bandwidth was minimal. Model3 uses localized communication and synchronization at each step. At a given bandwidth, messaged completion time is directly proportional to the number of link failures. There is a performance loss of 20% when there is 10 link failures. In other words there is 25% link failure ratio.

In reality synchronization is performed for every 5 steps resulting in a net synchronization is 20%. If synchronization is less than 20% the performance degradation was obtained to be lesser than 10%. But if it was more than 20%, the performance degradation starts to increase. Since model 3 assumed 100% synchronization, it is proven that in reality(with systems of less than 20% synchronization) it would have more tolerable performance degradation

Adaptive routing is achieved by adding hardware components such as link drivers. Since link drivers are expensive, the main cost of Adaptive routing comes from adding more links. Adaptive routing hurts performance only when there are failures. Performance degrades gracefully on occurrence of failures.

This article proved the effectiveness of adaptive routing techniques for fault tolerant systems. . Adaptive routing is a method of routing, which gives several options of paths to route and then chooses the optimal path based on the state of the network. This routing method yields the system with graceful degradation when workloads are considered.This routing method showed that the problem completion time when failures have occurred in medium to large communication granularity. There was no loss of performance when there is a mismatch between problem communication structure and physical communication structure. Thereby proving adaptive routing techniques is better option for fault tolerant systems.

Relevance to the Lecture

In the lecture we discussed about several routing techniques, one of which is adaptive routing.

Differences from the Lecture

In class though adaptive routing was not discussed in depth, this paper relies heavily on the performance analysis of adaptive routing, when there is node or link failures and effects of synchronization and bandwidth .

Paper Analysis of Lecture #17: Flow Control and Deadlock

Paper 1: Flit-Reservation Flow Control [144]

Description of the paper

The focus of this paper is to present a flit-reservation protocol, where control flits traverse the network before the data flits. These control flits would reserve bandwidth and buffers. In order to achieve this task, some on-chip wired control signals could be directly connected from node to node, or the control flits could be pipelined before the data flits. This paper starts by going over the most popular flow control techniques and it then describes in detail the flit-reservation flow protocol. The authors also give the results of several flit-reservation protocol simulations under various conditions. Finally, the paper compares the flit-reservation flow control technique with virtual-channel flow control and discusses the effects of packet length and scheduling.

Relevance to lecture

This paper is relevant to the lecture for several reasons. The first part of our lecture was focused on buffered flow control. Some techniques of buffered flow control were presented in class and this paper presents a novel technique that was not covered. This paper compares flow control techniques presented in class, such as store and forward, wormhole, and virtual cut-through, with the flit-reservation flow control. Like the flow control technique presented in class, flit-reservation is dynamically scheduled. Furthermore, this paper details the problems of the most popular flow control methods and explains how this protocol resolves some of them.

Flit-reservation protocol

Before discussing the flit-reservation protocol, the paper describes the other techniques which inspired flit-reservation. One technique discussed was statically-scheduled flow control. In statically-scheduled flow control, the compiler schedules the allocation of buffers and bandwidth before the program execution. This has the effect of removing the time necessary to perform routing and arbitration. Conversely, this method loses in flexibility because it cannot support data dependent communication.

Flit-reservation flow control provides many advantages of statically-scheduled routing protocols while supporting the flexibility of dynamically routed protocols. Following the example of a statically-scheduled network, the flit-reservation flow protocol schedules the buffers and channels of the network in advance, before the data packet arrives.

Unlike statically scheduled networks, the resources are not scheduled at compile time but the decision is taken only when the data is ready to be sent. This allows the schedule to adjust depending on the packet size, the destination, and the route that must be taken. Statically-scheduled network and flit-reservation protocol both use buffers efficiently. They hold the buffer only during the buffer usage unlike other methods that holds the buffer from the moment the flit is forwarded until the credit is received. Data latency is reduced because routing and arbitration decision are made in advance. The biggest advantage of flit-reservation over statically scheduled networks is its flexibility.

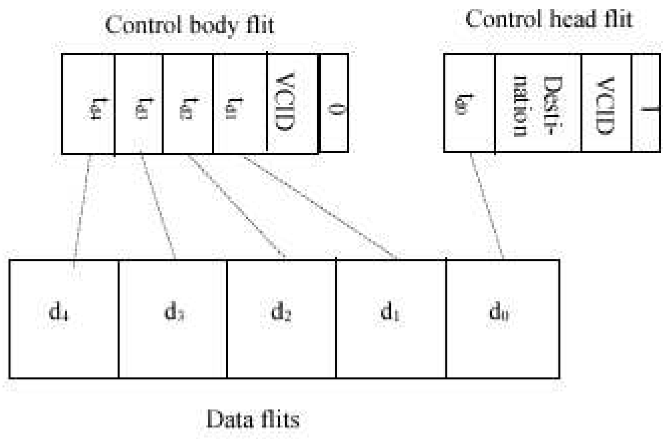


Figure 87. Flit-reservation packet [144]

Figure 87 demonstrates the packets of flit-reservation flow control. A packet consists of at least one control flit and zero ore more data flits. The first control flit contains information related to destination and specifies the arrival time of the data flits. The other control flits contain the arrival time of the subsequent data flits. The data flits, like the other forms of flow control, contain only data. The data flits are identified by their arrival time.

Upon arriving at a router, the control flits need to go through routing, output scheduling and input scheduling. To determine the output port on which to forward the packet, the destination field of the first control flit is used and that output port is stored in a table with its virtual circuit identifier. The other control flits look up the output by using their virtual circuit identifier. Every time a data flit is successfully scheduled, its arrival time field is updated with the scheduled arrival time in its control flit.

The paper claims a higher throughput using flits reservation protocol than virtual channel flow control. A simulation with wired control signals with 8 flit buffers per input shows the flit-reservation protocol can extend the 63% throughput achieved with virtual channel flow control to 77% which is an improvement of 20%.

Paper 2: FC3D: Flow Control-Based Distributed Deadlock Detection Mechanism for True Fully Adaptive Routing in Wormhole Networks [143]

Description of the paper

This paper specifically addresses deadlock strategies in wormhole networks. First, it introduces the two main ways to deal with deadlock, namely avoidance and detection. It proceeds to identify shortcomings in both approaches, including the need for additional resources, wasted resources, or excess congestion. A novel technique for deadlock detection is then presented which uses only local information provided by flow-control mechanisms in the network and gives rise to false-positive deadlocks with much lower likelihood than current schemes. The validity of the method is then proven mathematically, and it is finally tested against other schemes for deadlock detection to gather some performance metrics.

Relevance to the lecture

This paper is relevant to the lecture for several reasons. The second part of the lecture dealt with deadlock and means for dealing with it; the paper presents recent research in this area. The first sections of the paper serve to review concepts that were gathered during the lecture, including the different ways of handling deadlock, the core components of each one, and advantages and disadvantages of each method. The lecture served mainly as an overview of the mechanics of the different deadlock handling techniques without treating performance and implementation issues in any great detail; this paper, by presenting a specific protocol for deadlock detection, provides an in depth analysis which exposes many of the practical issues that interconnection network and SoC designers would encounter in the process of creating a performing yet robust network.

Similarities and supplements to the lecture

In the background section, the paper expands upon several concepts introduced in the lecture. During class, we noted under what conditions deadlock could arise, but not the likelihood with which it could arise. Although mentioned in [137], the authors of this paper put particular influence on the fact that deadlocks occur with small probability given freedom of routing paths. Thus, standard deadlock detection mechanisms that give false positives allow rare occurrences to have a negative impact on network performance. If some approach could be used which is virtually free of false positives, this would be a preferable solution.

In the lecture, it was also mentioned that the deadlock detection algorithm should be simple; those proposed included simple timeouts or counters. However, the authors of this paper add a second constraint on deadlock detection: locality of information. Because deadlock occurs due to congestion, trying to gather remote information to determine if there is deadlock could congest the network even more. Although none of the methods described in the lecture contradict this notion, it is important that it be explicitly stated as a design constraint.

Timeouts also present another problem not treated in the lecture; how long should they be? In networks with fairly homogeneous packets and load conditions, setting one timeout value for all packets is reasonable. However, in many embedded systems, some tasks involve long transactions while others involve short ones (see [145] for an interesting account of how tasks with different communication characteristics on a bus architecture led to the failure of the Mars Pathfinder). Also, because the message sources have no idea what other packets they will run into en route to their destination, there is no way to locally set an intelligent timeout, and this could lead to deadlocks lagging in the system for a long time at one extreme to the detection of many false positive deadlocks in a congested network at the other.

Also not considered in the lecture is the consequence of detection at more than one node. As we saw in the examples presented in class, more than one agent has to be involved in deadlock, and usually locked resources are distributed over several nodes. If, say, two agents detect that deadlock has occurred, it may prompt both to release all of their resources when really only one needed to do so to eliminate the problem.
Novel approach to deadlock detection
The basis of the flow control based distributed deadlock detection (FC3D) approach presented in this paper is straightforward. If an agent is blocked by a resource held by an agent that is itself blocked, then deadlock *should not* be detected by the former agent. If, on the other hand, an agent is blocked by an agent that is still making progress, and the latter agent blocks sometime later, then deadlock *should* be detected by the former agent.

The approach seems simple, but it has an important consequence. Let us consider the scenario depicted in Figure 88. Here, the dotted lines indicate the next length of the path to the destination, and the solid lines represent virtual channels that are already occupied. We see that deadlock is imminent, and that a cycle is about to be closed when A tries to claim the virtual channel held by D. In the diagram on the left, we see the status of the network just before deadlock occurs. D has blocked on C, which has blocked on B, which has blocked on A. According to the protocol, C and D should not detect deadlock because they have blocked on resources owned by other blocked agents. When A does block on D in the diagram on the right, deadlock occurs. A, which blocked on D, does not detect deadlock since D was already blocked. Thus, it is B that detects deadlock; A was advancing at the time that B blocked, and then got blocked afterwards. With normal deadlock detection schemes, it is possible that B, C, and D would have all timed out simultaneously. Assuming regressive recovery, these three agents would have all been eliminated from the network. With the new approach, only one agent (B) would be ejected, and the deadlock cycle would be broken at much lower cost.

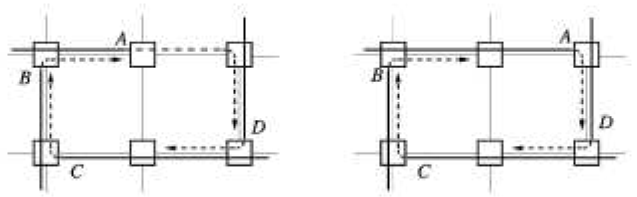


Figure 88. Deadlock detection with FC3D [143]

The method is implemented with very little additional hardware when compared to a scheme that uses timeouts; in fact, only three extra bits are required to indicate the following information:

Is the current agent blocked?
Has the current agent been blocked for a long time?
Did the current agent block on a progressing resource?

Deadlock recovery techniques should be taken on the agent only if the answer to all three questions is yes (i.e. all three bits are set).

Results of a comparison between the proposed mechanism and standard timeout detection with regressive recovery generated favourable results. The scheme was tested on irregular networks containing 16, 32, or 64 switches. Not only were false positive detections reduced by up to two orders of magnitude because of the use of the three constraints instead of a timeout, but FC3D also increased throughput by up to 10% because fewer messages had to be retransmitted. Also, the timeout method needed a fairly long threshold value to prevent the generation of too many false positive results. Because of the other flags in FC3D, the thresholds could be shorter without adversely affecting network performance.
Summary
This article provided an interesting discussion of the drawbacks of existing deadlock recovery schemes in wormhole networks and expanded on many of the topics discussed in class. It elaborated on the practical design constraints involved in creating such a mechanism. A novel approach to deadlock detection was presented that significantly reduced the detection of false-positive deadlocks, detected deadlocks with a much smaller timeout threshold, and improved network throughput when compared to conventional approaches.

Papers Analysis of Lecture#18: Network on Chip

Paper 1: A Network on Chip Architecture and Design Methodology

Description of Paper

This paper proposes a packet switched platform, which they called Network-on-Chip (NoC). The author proposes a m × n mesh of switches where the resources are inserted inside the grid. Each switch is connected to four other switches and one resource. A resource can be a processor core, memory, DSP core, a FPGA or virtually any IP block that can fit. These blocks need to be able to interact with the interface of the NoC. They claim this type of architecture scales well with an increasing number of processing elements.

Relevance to lecture

In this lecture, we discussed the trend followed by the industry which is the development of NoC. We discussed the reasons why a network topology on a chip becomes necessary. The mesh topology is then introduced as one solution to link the different resource blocks. This topology, called CLICHÉ, has heterogeneous blocks which are linked together. This article also suggests this type of interconnect.

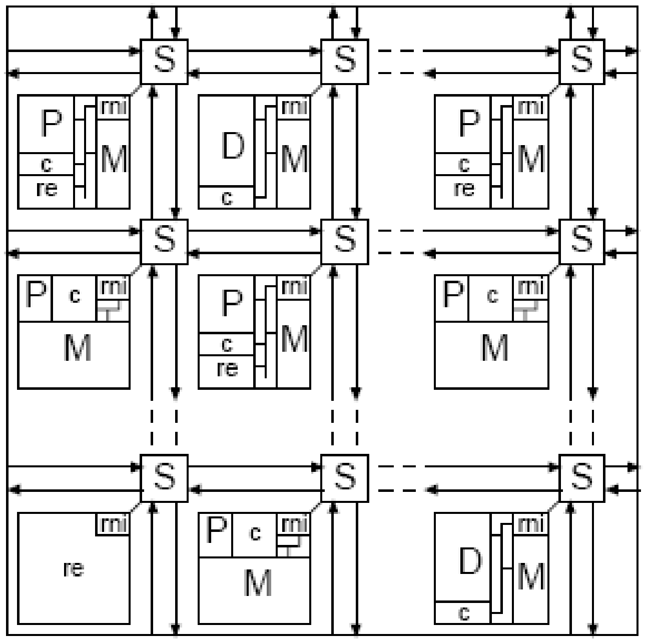


Figure 89 Example of a CLICHÉ topology [146]

The previous figure shows the CLICHÉ topology consisting of a mesh network connected to different resources. Each resource is different.

The lecture was oriented on the architecture, the topologies and the different layers of abstraction the on-chip network has. This article puts more emphasis on the development of such a device. It proposes a platform to design and implement a NoC more easily. This platform is discussed in more details later in this document. It also stresses the fact that the industry will have to adapt, by using different design/verification tools and methodology. Finally, the article shows simulation results where different buffer sizes in the switches are tested. As discussed during the lecture, this simulation shows tangible proofs that buffer sizing is an issue to be addressed seriously.

Backbone-Platform-System Methodology

The author also suggests a new concept called Backbone-Platform-System methodology (BSP). The main goal behind BSP is to enclose the design work into reusable platforms. BSP is divided into two main phases: platform development and application mapping. Here the author stresses the necessity to raise the level of abstraction in order to work with a network of SoC. The design is first divided into three different blocks: the backbone, the platform and the system. The backbone's main goal is to provide a physical communication network such as channels, switches. Here, the emphasis is put on the interconnection network itself. Area taken, number of wires, buffers and synchronization are problems that are addressed in this section. The platform design concentrates on the creation of a computation platform for an indented application. Resource blocks are added at this stage, implying the area must now be taken into consideration. Performance metrics, utilization and capacity are evaluated at this stage. The system design stage puts emphasis on control and functionality of the network. NoC designers face the same problems encountered with parallel or distributed systems. Resource allocation, network usage optimization, performance metrics are issues being addressed at this level.

Methods and Tools

Tools and design/verification methods are then discussed. EDA tools will have to adapt to this new approach. Network-related issues such as distribution and parallelism will have to be taken into account. Designers will have to decide how to map the functionalities between the resources and how to verify them. Abstraction models will be needed at the system level to limit the complexity of the design. The computational capacity of design tools aren’t and won’t be able to handle the simulations and analyses of a whole NoC if some parts aren’t reduced to a behavioural model.

Drop Probability and Buffer Size

The graph below shows the relation between the drop probability and the buffer size in switches. It was supposed that the maximum traffic between two switches is 200Mbits/sec. It was observed that the dropped traffic rate is very close to zero when a four packets buffer is used. It was noted that message delay is more sensitive to traffic than to buffer size. It was also impossible to reach 0 percent dropped packet when the network load was higher than 50% its maximum capacity.

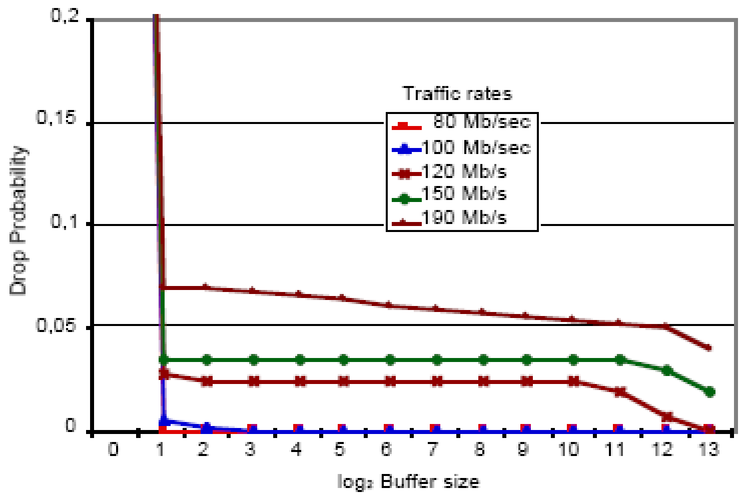


Figure 90 Experimental results for dropped packets vs buffer size with different traffic load [153]

Paper 2: “Performance Evaluation and Design Trade-Offs for Network-on-Chip Interconnect Architectures” [154]

This paper is about a consistent and meaningful evaluation methodology to compare different network-on-chip (NoC) architectures regarding their performances and their characteristics. By comparing NoC topologies with realistic traffic models, the authors are able to demonstrate the various design trade-offs and the impact on their performances.

Instead of using the common shared medium arbitrated bus where all IP blocks use the same transmission medium, five recently proposed NoC topologies will be evaluated regarding different criteria (see Figure 91). Their network-centric approach will prevent a propagation delay that could occur with a large number of IPs and lead to exceeding the target clock like in the bus topology case. Functional IP modules (white squares) are able to provide a system with high-performance parallel computing by using infrastructure IPs (black square in Figure 91) which communicates in the form of packets to each other.

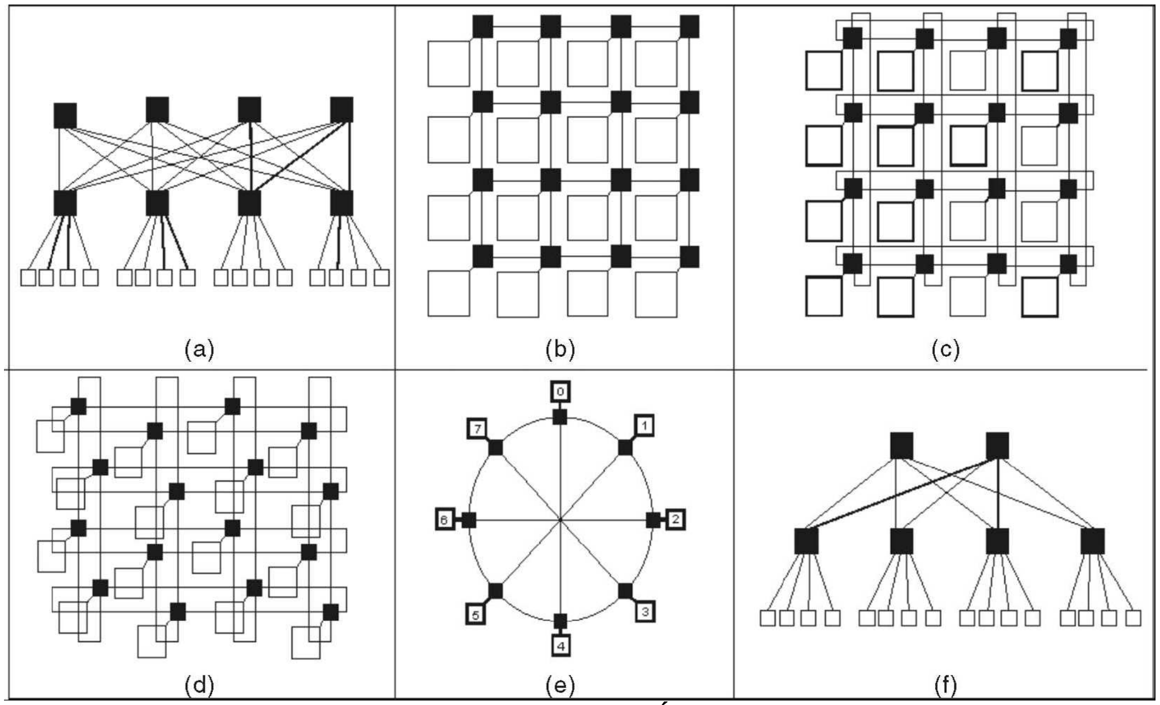


Figure 91 NoC architectures. (a) SPIN, (b) CLICHÉ, (c) Torus, (d) Folded torus, (e) Octagon, (f) BFT.

The first topology is called SPIN (Scalable, Programmable, Integrated Network) and is a tree with each node having four children. A mesh-based interconnects architecture called CLICHÉ (Chip-Level Integration of Communicating Heterogeneous Elements) is the second one and consists of switches connected to their IPs. Each pair has four links except the ones on the edge. The third topology is a 2D torus which is like the previous one but with the switches at the edge connected to the opposite ones (Figure 91c). To avoid delays created by those new links, the torus can be folded (Figure 91d). The OCTAGON (Figure 91e) consists of eight nodes (a switch and an IP) and twelve links so all nodes are at most two hops away from each other. Finally, the last topology is called Butterfly Fat-Tree (BFT). IPs are the leaves of the tree and each switch has 2 parents and 4 children.

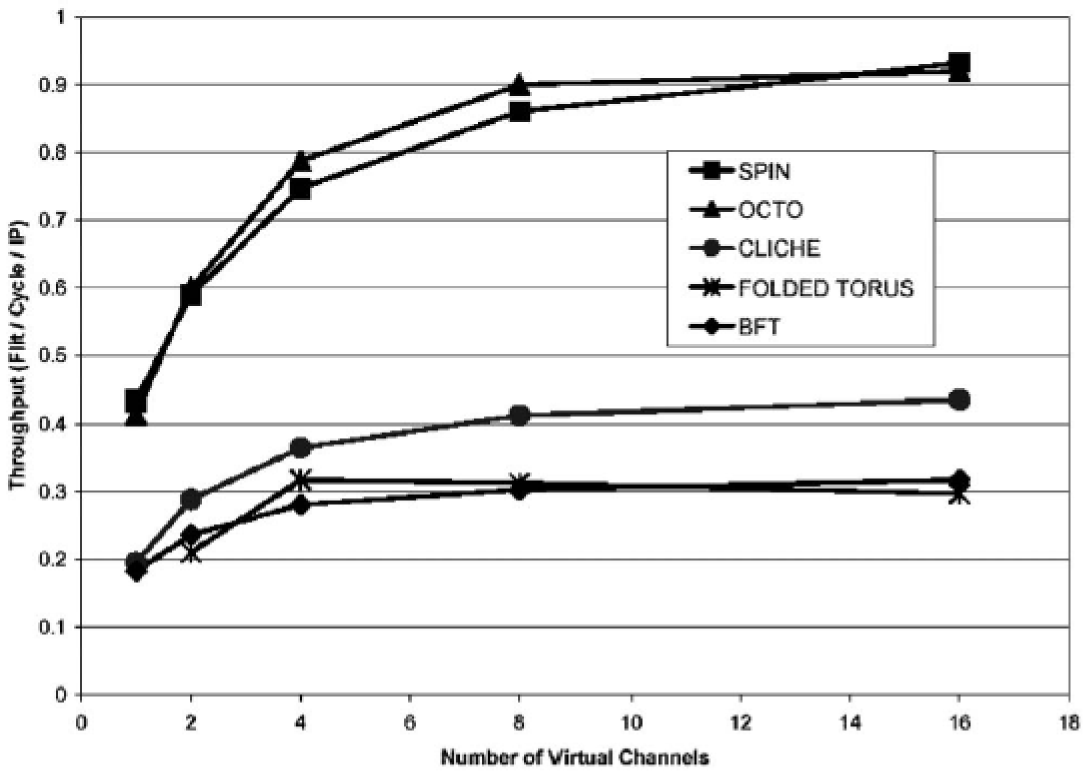
Three performance metrics will be compared between the different topologies:
Message throughput
Transport latency
Energy

Throughput

We compare the throughput of the different network topologies under Poisson and self-similar traffic injections. First, let us define the throughput as the maximum traffic accepted by the network. It relates to the peak data rate sustainable by the system. Under uniform traffic, SPIN and Octagon are doing better since more links exist between the source and destination (Figure 92). However, if we use a certain percentage dedicated to local traffic (which is more realistic in a SoC environment) and using four virtual channels, CLICHÉ, Folded Torus and BFT are doing a lot better and catch up in performance with SPIN and Octagon (Figure 93). Note that both figures use a Poisson traffic injection but they also get similar results under a self-similar traffic except with a slightly lower average data rate.

Latency

Latency is closely related to throughput. As injection load approaches the accepted traffic (throughput) limit, there will be more message contention and latency will increase (Figure 94). With a traffic localized, we see again a major improvement of the CLICHÉ, Folded Torus and BFT topologies concerning the latency. This will allow higher injection of traffic.



.
Figure 92 Throughput under Poisson traffic

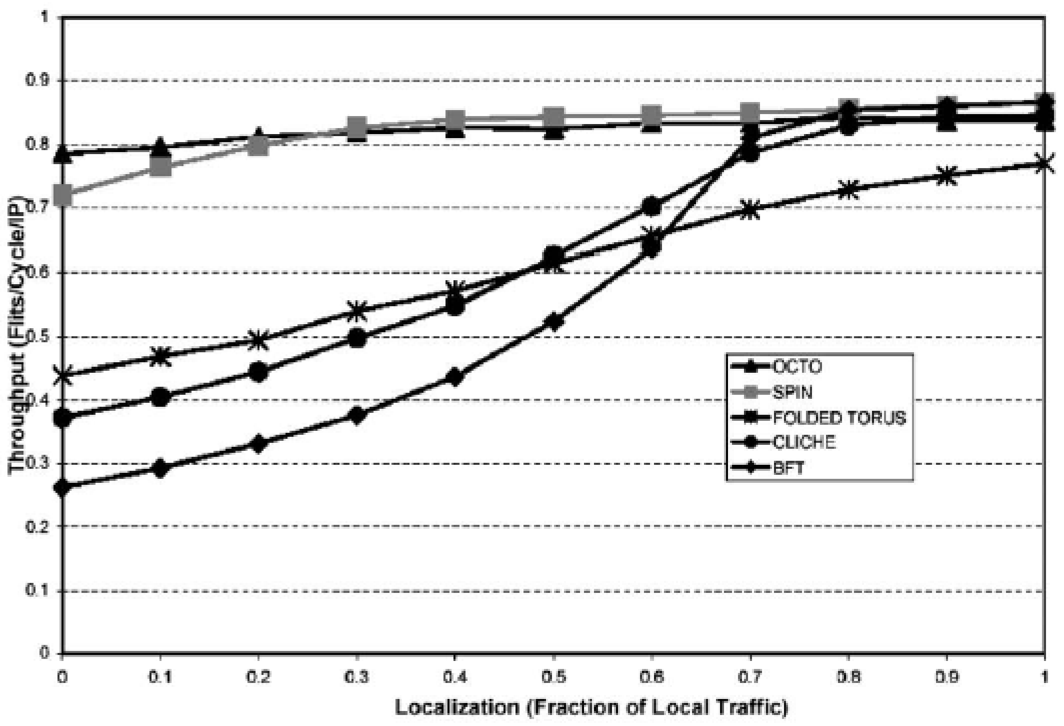


Figure 93 Throughput under Poisson traffic and four virtual channels

<u>Energy</u>

To evaluate the energy dissipation of a packet, we plot the average dynamic energy dissipated when a packet moves between a pair of source and destination IP blocks. This is influenced by the number of virtual channels and the injection load. When the network is operated at the peak sustainable data rate, the energy dissipation increases linearly with the number of virtual channels for all the architectures (Figure 95). The energy varies proportionally with the injection load of the network. It will saturate when the injection load reach the throughput limit since no additional packets are sent. It will also improve with the localization of the traffic since the packets will have smaller path to arrive to destination.

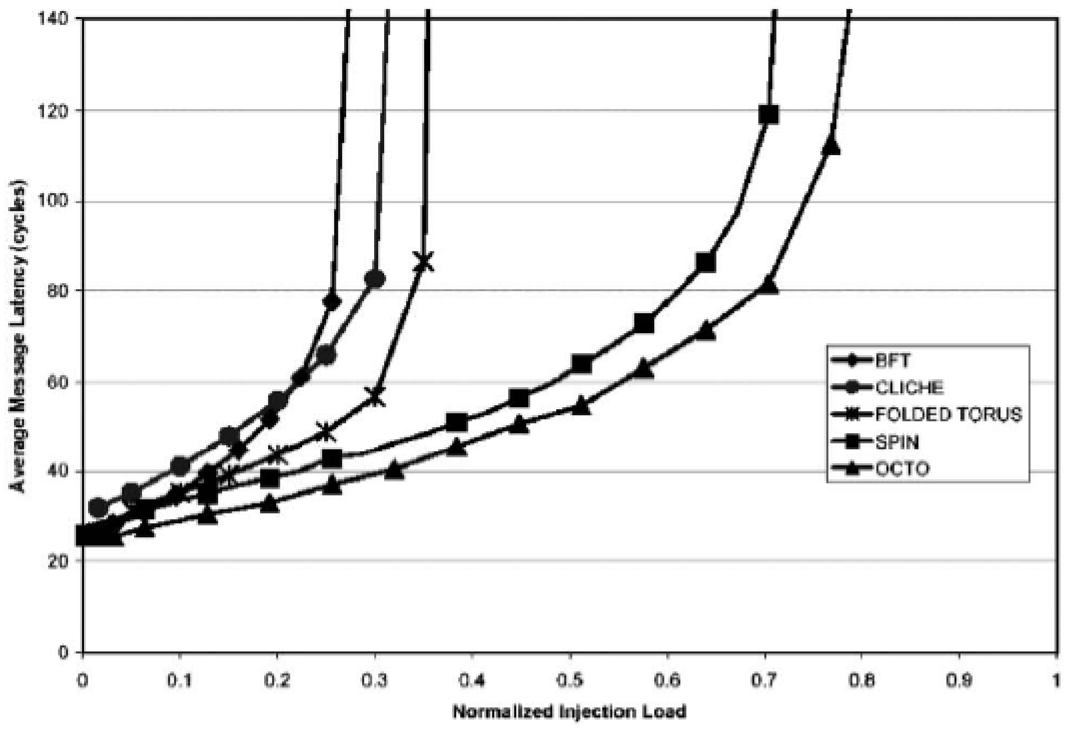


Figure 94 Latency with 4 virtual channels (Poisson)

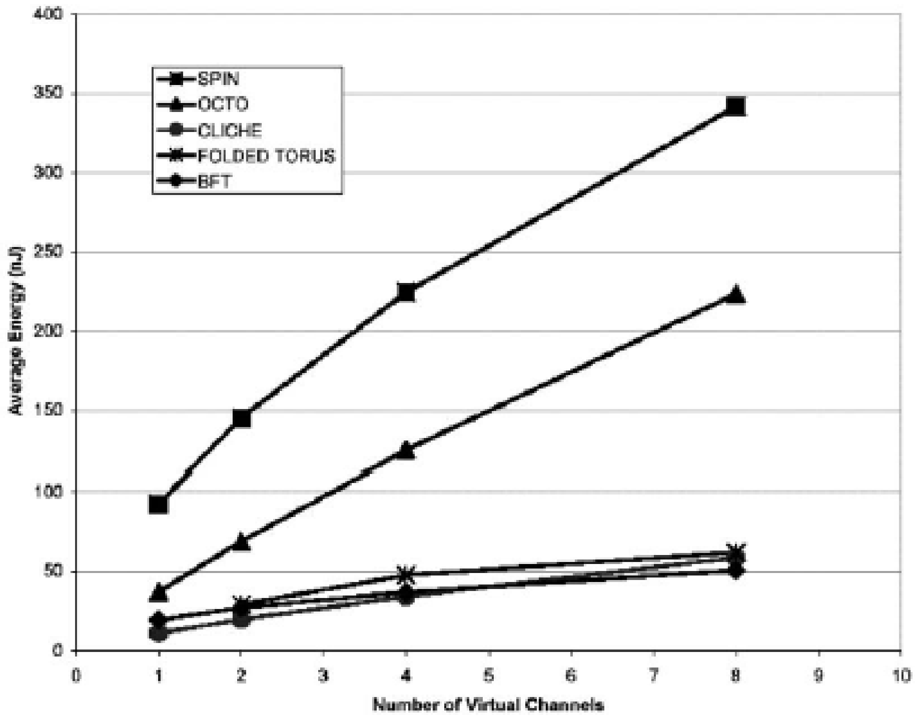


Figure 95 Average energy dissipation per packet

To conclude, we can see the importance of choosing the right network architecture and the trade-offs that each topology presented in the slides of the course will give to the designer of the NoC regarding throughput, latency and energy dissipation. We also notice that some network architectures are performing much better when traffic localization is high. Other characteristics could also be considered when choosing the network topology like area requirement, testability, dependability or reliability just to name a few.

Papers Analysis of Lecture #19: Scheduling and Dependence

Paper 1 Deterministic Processor Scheduling

by Mario J. Gonzalez, Jr. [157]

The above survey paper discusses the scheduling of tasks on single-processor and in multiprocessors. To start, it discusses the representation of set of tasks that are classified using the following categories: the number of processors, the task periodicity and the number of resources. It also shows what will happen when using a single-processor schedule and a multiprocessor schedule in a multiprogramming environment. Algorithms are used to picture those results and essentially most important is the heuristic algorithm solution.

In some ways, the paper is directly relevant to the lecture. First of all, it talks about general concepts like processor scheduling where tasks (also called jobs in this survey) are represented as nodes that are to be assigned to a processor for execution at a particular time.

Another important concept discussed in the paper that is also related to the lecture is the representation of (a set of jobs or) the tasks using a task graph to represent the relationship between the tasks. In fact, the task graph is the most popular representation in the scheduling of tasks. More precisely, the author mentioned that the nodes in the graph can represent independent operations or the parts of a single program which are related to each other in time. Each node is associated with a number that represent the computational time required to execute the code (represented by the node). The number of processors determines the amount of time required to execute the tasks. A node must wait for its immediate predecessors to finish before it can be executed. The nodes can represent tasks of equal or unequal duration. The individual nodes within a graph can be related to each other in a number of different ways. For example, it is possible for all tasks to be independent of each other. In other situations, the graph is structured in such a way that every node in the graph has at most one predecessor or at most one successor.

Throughout the paper, the author then discusses about displaying schedules with a timing diagram known as Gantt chart. It's a diagram that is pictured in the following manner: the tasks assigned to each processor and their order of execution and their execution time are represented by horizontal lines and the tasks identification is adjacent to each processor. The author emphasizes that the flow time of a task is the time at which its execution is completed and the flow time of a schedule is the sum of all the flow times of all tasks in the schedule.

The survey paper then discusses novelties from the lecture: scheduling techniques are based on two conditions: there are no loops and there are no decision nodes in the task graph.

Regarding the single-processor schedule, all tasks are executed at the same time. The characteristics of the tasks stay constants during the tasks duration.

The survey paper concludes by emphasizing the fact that future works should be on the study of heuristics.

Paper 2 Scheduling Problems for Parallel and Distributed Systems

by Olga Rusanova and Alexandr Korochkin [158]

This paper discusses scheduling algorithms for parallel and distributed computer systems. The authors reveal that today, parallel and distributed systems have increasingly developed, resulting in a high improvement in performance. Thus, scheduling is one of the main factors of that improvement in performance. That is why the authors concentrated their study on that important aspect. They also specify the two evaluations of scheduling. Those are the scheduling performance and the scheduling efficiency.

Scheduling performance represents the minimal total completion time of a parallel program and scheduling efficiency is a time complexity of the scheduling. In fact, the paper clarify that time complexity is a measure for dynamic methods which execute the schedule during the runtime of parallel programs and time complexity is not important for static scheduling where the execution is just at compile time.

The paper thus focuses on static scheduling and the authors emphasize that today static scheduling algorithms are based on heuristic scheduling algorithms. Heuristic algorithms are divided in three categories: genetic, clustering and list. In genetic, scheduling of distributed systems and parallel systems with a distributed memory take into account tasks and the system topology.

In clustering, there exist two approaches. In the first one, many tasks are assigned to a number of processors and tasks are then merged and scheduled on a limited number of processors. In the second one, one processor do the sequential computation. If there are more processors available, some tasks from the first processor are distributed among the other processors to execute the computation in parallel, thus reducing the time to do the entire computation.

In list scheduling technique, the order of tasks is by priorities. Then the tasks are assigned to the processors. The assignment can be done with or without considering the communication delay. The authors specify that the best results give the scheduling heuristic in which the topology and the communication time between tasks are considered.

The authors mention that list scheduling is the best scheduling technique for parallel and distributed systems.
The paper is relevant to the lecture in many ways. Firstly, as stated above, the concept of heuristic algorithm is similar to the one covered in the lecture. In addition, the assignment of tasks to processors by priorities was also discussed in the lecture.

The main problem in this paper is identical to the one presented in the lecture, mainly, the scheduling of parallel programs on parallel or distributed systems to minimize the total completion time of the execution of the program. Parallel systems consist of homogenous processors while distributed systems consist of heterogeneous processors.

The lecture discussed about task graph. That is also the case in this paper where the authors mention that parallel program can be represented by a directed acyclic graph (DAG) also called task graph. The graph is composed of a set of nodes that represent tasks and a set of edges that are used for communication between each node. The authors emphasized that each node has a computation cost and each edge has a communication cost. Before starting to execute, a task must receive all its input data. Then it executes until it finishes. In fact, the task graph is assumed static. This means that during the entire execution, there is no change in the graph.

Processors of parallel or distributed systems are connected in a given topology where each processor has one or more links for communications. One thing that was not mentioned in the lecture was the processor graph which represents a parallel system topology with undirected unweighted graph. The graph consists of a set of nodes (processors) and a set of edges (communication). In contrast, a distributed system topology can be pictured as an undirected weighted graph where the different weights are: the weight of the nodes and the weight of the edges. The weight of the nodes represents the processor performance and the weight of the edges the channel capacity.

Virtual processors graph are also discussed in the paper and was not in the lecture.

Lecture covered the concept of Gantt chart. In this paper, that concept is not omitted and the article specify that a schedule of a task graph can be illustrated as a Gantt chart, where we can visualize the start and finish times of all tasks. More precisely stated in the paper, a Gantt chart consist of a list of all processors and, for each processor, a list of all tasks allocated to that processor ordered by their execution time, including the start and finish time of the tasks.

Another concept outlined in the paper that was covered during the lecture is the two key components that contribute to the total completion time of a program. Those are the execution time and the communication delay. In fact it is important to note that during scheduling those two components are minimized as much as possible. Recall that in list scheduling, there are two stages: formation in order (with priorities) and allocation of tasks to processors. Those two stages have different goals. The goal of the first stage is to minimize the execution time and the goal of the second stage is to minimize the communication delay.

In addition, the paper recall that today, there are three main communication models: model-A, model-B, and model-C. In model-A, each processor cannot execute a task and communicate with another processor at the same time. Here, the communication cost is equal to the sum of the edges weights in a given task graph. In model-B, it is stated that if a task has several successors, and some of them are allocated to the same processor, their communication cost is counted only one (in model-A, it was counted multiple times).In model-C, the paper states that an input-output processor is assumed to exist and that a processor can execute a task and communicate with another processor or processors(if there are multiple links) at the same time. Another important thing that was mentioned in the lecture is that communication delay between tasks allocated to the same processor is negligible. The communication delay between two tasks allocated to different processors is a function of the size of the message, the route, and the communication speed.

The authors also specify that in the list scheduling where tasks are considered by priorities, the priority of the tasks is determined by some characteristics that are taken into account: the node weight, the earliest and the latest execution time and the sum weight of entering edges. Those characteristics correspond also to the ones of the task graph.

The authors suggest that to minimize the communication delay, it is better to use model-C or model-D.

To conclude, the paper summarizes the static list scheduling approach which was chosen to be the best scheduling technique for parallel and distributed systems. It also points out some future experiments to solve. As an example, a creation of a hybrid scheduling approach (static & dynamic).

Papers Analysis of Lecture #20a: Vector Processing and Vector Architectures

Paper 1Scalable Vector Processors for Embedded Systems [164]

This paper demonstrates an alternative approach to embedded processing that provides high performance for critical tasks and lowers power consumption while also reducing design complexity. An important part of vector processing is recognizing the possibility of executing a vector instruction. This is usually the compiler's job. The paper provides several benchmarks for the vectorization of multimedia application code. The tests show that 9 out of 10 benchmarks were able to vectorize over 90% of the instructions for execution on the vector processor [164]. This will in turn reduce the number of instructions executed and increase performance.

Continuing to enforce the benefits of Vector processors, the paper explains the advantage of using multiple parallel lanes (functional units) in order to achieve greater performance and scalability, as well as reduced design complexity. By introducing the concept of parallel lanes, it becomes possible to add or remove lanes easily in order to increase performance.

The paper also discusses features such as adding On-Chip main memory and clustering the memory to reduce power loss and enhance performance, respectively. The paper concludes by stating that Vector processor architectures demonstrate a great potential for use in embedded systems because they can be made to exploit vector techniques for data level parallelism as well as multithreading techniques for task-level parallelism – the two major types of parallelism in embedded applications [164].

The lecture on Vector processors was an introduction to the subject, and this paper is relevant as it reinforces what was discussed in the lecture, as well as it explores some of the possible applications of Vector processors. This journal also encapsulates previous lectures about System-On-Chip by introducing the processor as an embedded system with On-Chip memory. Many of the concepts explained in the lecture about Vector processors were applied when the authors wrote the journal. The authors have added their own modifications to the general architecture of the Vector processor in order to accomplish the task of enabling the processor to perform in an embedded system environment.

The paper also performs software analysis. As in the lecture, the instructions (assembly instructions) that run on a Vector processor were discussed. If some instructions can not be vectorized, then it is not possible to take full advantage of the

Vector hardware; therefore, a scalar processor may be more efficient. This highlights the importance of a good compiler when vectorizing code for multimedia applications. The more vectorizeable that the produced code is, the greater the performance increase, as shown in the test benches.

This journal was similar to the lecture in many ways. The general architecture of the Vector processor was exactly as discussed in class. A Vector processor is an SIMD architecture. One instruction is used to control all the functional units (parallel lanes) that operate on different data simultaneously. This makes for a very large register file and datapath with a corresponding simple control [164].

In the lecture, it was shown that Vector processors deal directly with memory when loading instructions or operands. The paper explains that no data caches are used because Vector processors deal mostly with applications that have limited temporal locality (multimedia applications) [164]. The processor is interconnected with memory and other functional units through the use of crossbar networks. The size of the crossbar network depends on the number of inputs and outputs to the register file, as well as the number of functional units and control signals.

One of the differences between the paper and the lecture is that the Vector processor discussed in the journal is a co-processor to a MIPS architecture. The MIPS processor is responsible for generating the vector instructions for the co-processor. A special compiler called a vectorizing compiler is used to generate vector processor instructions from the provided code of the multimedia application [164].

A novelty of this paper is clustering the vector register file. In the lecture, a large register file was shown with 8 write inputs (1 per block) and 16 outputs (2 per block). This is similar to the register file shown in Figure 8a. The register file is seen by the processor as one unit with many inputs and outputs. The paper suggests clustering the register file into several smaller files to increase performance. This new clustered architecture is also scalable since a new register file can easily be added along with a new lane to further increase performance [164].

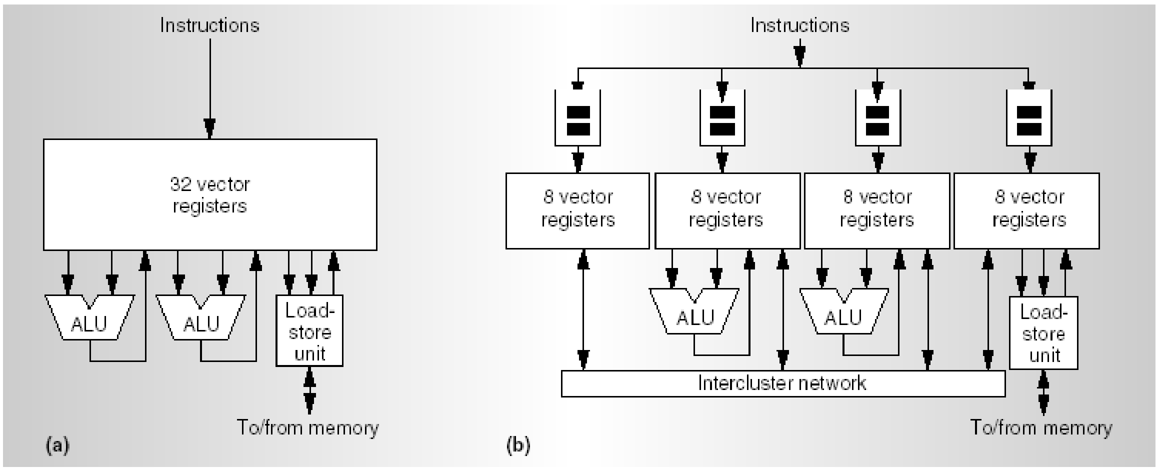


Figure 8: Memory Clustering [164].

Paper 2 Simple Vector Processors for Multimedia Applications [165]

This paper examines the emergence of multimedia applications and the influence they have had on the designers of microprocessors. The paper suggests that adding vector hardware to a state-of-the-art superscalar processor increases its complexity by making it difficult to implement. Therefore, this paper puts forward a design similar to traditional vector computers with simple control logic [165]. As an argument for implementing a Vector processor, the paper discusses a new design where the bulk of transistors and die area is used for datapath and registers, reducing the time to design, implement, and verify the controlpath [165]. The paper then explains why Vector processors are faster at processing multimedia applications than Scalar processors, even with a larger CPI. In conclusion, this paper states that a simple, highly parallel Vector processor represents an alternative for executing multimedia applications [165].

During the lecture, it was explained that many scientific applications deal with vectors. Vector processors are more efficient at processing multimedia data than scalar processors because multimedia data also deals large sets of data. This paper applies the advantages of Vector processors discussed in class to common computing problems such as processing multimedia applications. As technology advances, there exists a reliance on computers, cell phones, PDAs, etc., to provide multimedia applications. Many of these applications are real-time, or almost real-time. This demonstrates the need for a processor that can process multimedia applications in an efficient manner (i.e. Vector processors).

A portion of the lecture was dedicated to comparing the number of instructions required to implement the same program on a Scalar processor against a Vector processor. As shown in the lecture, a Vector processor requires many fewer instructions to implement the common loop of a Scalar processor. This paper expands on this observation by examining the reasons why a Vector processor has a higher cycles-per-instruction (CPI) than a scalar processor, and yet it is much faster. The journal also dissects the cycles-per-operation (CPO) in order to explain that the Vector processor is using the instruction-level-parallelism more effectively than the Scalar processor.

The paper deduces that even though the Vector processor has a higher CPI than the scalar processor, the number of instructions is significantly lower on a Vector processor. This results in lowering the CPO of the Vector processor, thus making it faster in comparison to the Scalar processor.

This paper exhibits similarities to the lecture as it suggests developing a processor more like the traditional Vector processor as studied in order to improve the processing of multimedia applications. This includes a simple design and control, multiple parallel functional units, and a vector register file (VRF) [165].

The Vector processor is a processor built for applications that contain vector data. Vector chaining was studied in the context of performance improvement in Vector processors.

This paper also explains how the Vector processor hardware automatically detects vector dependencies and can perform chaining when these dependencies are present.

In Figure 9 below, one observes the different benchmarks of Vector processing on vector data as compared to a Superscalar processor. The different benchmarks were all performed on multimedia type programs [165]. A benchmark was analyzed during the lecture. Benchmarks can be an important tool in differentiating strengths and weaknesses of different architectures.

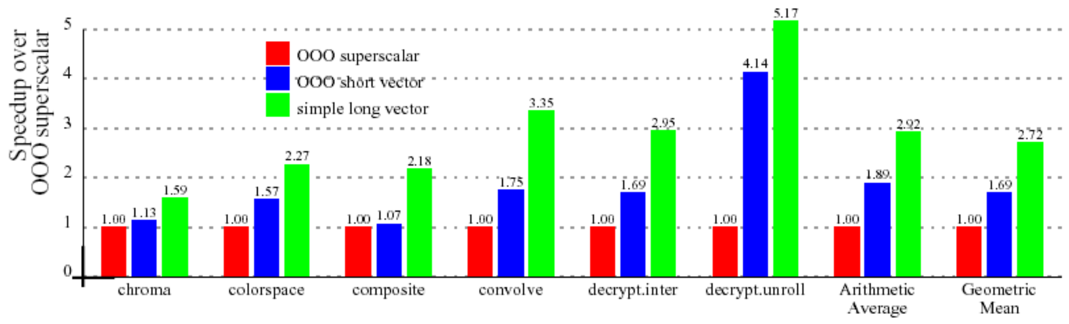


Figure 9: Speedup of Vector processors over Superscalar processors [165].

This paper discusses many aspects of Vector processors in terms of their size on a die. Although we did not cover this topic in the lecture, it is an interesting subject to cover. The journal compares the uses of the die between a scalar processor and a Vector processor. A Vector processor is determined to utilize most of the area for datapath and VRF in contrast to the Scalar processor which uses most of its area for control of instructions [165]. The Vector processor is able to reduce complex control for the larger datapath because it utilizes instructions that compactly encode parallel functions. This, in effect, transfers much of the functionality for detecting parallelism to the software [165].

Papers Analysis of Lecture #20b: Vector Processors

Paper 1 Scalable Vector Processors For Embedded Systems [170]

This paper examines a new area of deployment for Vector Processors (VPs); that of embedded, multimedia applications with data-level parallelism (DLP). These systems are traditionally designed using modified superscalar and very large instruction word (VLIW) processors that are adjusted to work in embedded systems. These systems are, however, very cumbersome to scale as they require significant hardware redesign (superscalar) or instruction-set redefinition (VLIW); since instruction-level parallelism (ILP) is detected dynamically by hardware in superscalar designs and during compilation by the compiler in VLIW designs. The authors present an alternative design using VPs, more specifically the Vector IRAM (VIRAM) processor developed at UC Berkley, that is configured to work with DLP and that provides high performance for embedded tasks without increasing power consumption and complexity dramatically when scaled. The processor is evaluated against superscalar and VLIW architectures using benchmarks from the Embedded Microprocessor Benchmark Consortium (EEMBC). The authors' simulations and prototype prove to support their thesis that VPs can outperform superscalar and VLIW processors while using the same amount of power and retaining simplicity of design. They have concluded that their prototype outperforms other high-end systems by 1.5x-100x for media tasks.

This article is relevant to the content of the lecture in that it deals with an application of VPs in embedded systems. It uses the inherent features such as compiler vectorization of code, ability to perform parallel arithmetic operations and load/stores to increase efficiency and speed of multimedia applications. The VIRAM processor block has four Lanes in which it operates in, thus the architecture is similar to that of conventional VPs; this architecture provides a simple mechanism for scaling performance, area and power consumption. In the case of VPs, scaling produces noticeable speedup. The scaling from a one lane VP to a four lane VP produces a linear speedup (except when running JPEG benchmark). This linear speedup is not the case, for example, when scaling a four-way superscalar to an eight-way.

There are some differences between this design and what we have discussed in class. Most noticeably is the use of IRAM [167], developed at UC Berkley, which promises "considerably reduced latency and dramatically increased bandwidth to main memory, reduced power and energy consumption, and reduced space and weight for embedded, portable, desktop, and parallel computer systems" [167]. Another big difference is that this design does not use any SRAM, instead only the IRAM in conjunction with 8 DRAM blocks is used. The authors designed the system in this way so as to be able to have a slow 200 MHz clock that drives the unit, which in turn reduces power consumption. The DRAM latency is hidden due to the nature of the deep pipelines associated with load-store and arithmetic vector operations. The next noticeable difference is the authors' suggestion for reducing the complexity of Vector Register Files (VRF). For embedded systems applications a high number of these units are needed to be able to handle short and medium-length vectors that are associated with embedded applications; therefore, since adding multiple lanes does not help with the performance of the VRFs a clustered system is proposed that will alleviate the problem. The VRF is split up into four different components of each of equal size of VRF/4. These components are connected together using an Intercluster Network and are controlled by the main controller. A renaming table is used to identify the cluster that stores the source and destination registers for a vector instruction and indicates whether a transfer between clusters is necessary. This architecture is similar to that of multicluster superscalar processors; however, it is simpler because maintaining an issue rate of one vector instruction per cycle is sufficient.

This application is unique as it tries to utilize a system that is not the norm for embedded system applications. The authors present a very unique approach using VPs that looks like it has some merit for industrial applications. This paper is a very good example of how VPs can be used in the industrial field for commercial applications.

Paper 2 Vector Architectures: Past, Present and Future ***[167]***

This paper discusses the history of Vector Processors (VPs) as well as where VPs are today, and what applications could it be useful for in the future. The authors discuss the gradual fall of VPs since the introduction of "killer micros" in 1991; however, they also argue that there are many features inherent in VPs that should be considered when designing next generation computers, such as the vector instruction sets.
The authors go on to discuss the earliest supercomputers developed such as the Control Data 6600 and 7600, which were designed before the term supercomputer was coined, and also the TI-ASC and STAR100 developed in the early 1970s. The CRAY-1, which was the first commercially successful VPs, is mentioned and its architecture discussed as well.
The demise of VPs began with the advancement in CMOS VLSI technology that enabled designers to fit more transistors onto a single die, and clock the circuits at higher frequencies. No longer was it necessary to use expensive ECL technology to break the 100 MHz barrier. The rise of this fast, inexpensive and easily scalable processor quickly took over the supercomputer market.
VPs still remain much more powerful than massive parallel systems; however, the cost of a VP is what eventually led to its downfall, its price tag could be anywhere from 5 to 30 million dollars (this is in the 1970-80s), and the performance of MPPs can reach speeds close to that of VPs (especially in the case of cache-friendly applications). The reason as to why VPs are very expensive is mainly due to its large amount of very fast memory (usually SRAM/SSRAM) that is needed to provide the high performance inherent in VPs. The authors however present a case as to why vector ISAs, not physical architectures, should still be considered in future computer implementation. These are primarily due to: semantics, explicit parallelism in instructions, and that the combination of regularity in each vector instruction and explicit parallelism allows for aggressive design techniques.

The content of the paper is very relevant to the lecture in that it discusses many different types of VP architectures. The authors go into some detail about the structure of some of the commercially available VPs that were designed, from the CRAY-1 (1976) all the way to the NEC SX-4 (1996). Units such as Cycle time, Load/Store paths, number of Vector Registers, number of Elements/Register, and number of lanes (pipes) are all listed in Table 1 of the document. Concepts such as memory bandwidth and MFLOPS are also discussed and listed by the authors. It is interesting to note that VPs can perform 2 to 16 floating point operations per cycle, whereas microprocessors can perform 1 or 2 per cycle. The concept of inherent parallelism in vector instructions is also discussed and the authors propose that this instruction set architecture should be incorporated into future parallel architectures as it has many pros, such as greatly reduced instruction fetch bandwidth, which in turn can lead to simpler control units and a more aggressive clock. The concept of the stride in accessing memory is also discussed and its merits are explained. The stride value enables the processor to directly know the memory access pattern, thus speeding up the design.

There are some new concepts discussed in this paper that were not covered in class; Concepts such as power consumption and real-time performance. Since VPs have inherent localization of instructions once an instruction is executed only the functional unit, and register buses feeding it need to be powered. Other units, such as the instruction fetch unit, the reorder buffer and other power consuming blocks can be powered off until the end of the instruction’s cycle. The authors also propose future development paths for VPs. Certain scientific applications (such as meteorology) need VPs since MPP with microprocessors do not satisfy their requirements; however, since this is a small subset of the scientific applications there might not be a big enough market to support the development of these costly machines. The authors suggests that a possible path of development would be to mix all the high performance techniques together: vector processing, superscalar processing, and multithreaded processing. The ultimate goal would be to fully integrate vector and scalar processors to be able to handle large multimedia tasks, since in the next couple of years computers will be dominated mostly by multimedia tasks. This would allow computers to be able to handle the smaller vector sizes of multimedia tasks while having the throughput and simplicity of a vector design for parallel processing. Incorporating the ISA into this model would also simplify greatly the programming of these units for parallel applications. The authors believe that if VPs remain the way they are today (without incorporating CMOS design and commodity components) they will quickly disappear; However, if these incorporations are made and a hybrid system is developed incorporating the best of both VP and microprocessor architectures then these systems, in the future, will come to the fore for their ability to extract more parallelism from programs.

Papers Analysis of Lecture #21a: VLIW Processors

Paper 1: Very Long Instruction Word Architectures and the ELI - 512 [187]

Description of the paper
The focus of this paper was to introduce the term VLIW and the VLIW architecture. The paper also focused in explaining Trace Scheduling algorithms which is also called region scheduling and exposing the main problems faced in simulations. This paper also introduces ELI – 512 (Enourmously Longword Instructions – 512 bit) which was the machine that implemented parallelism. The main objective was to find alternatives to the problems related to speed up program codes using VLIW Processor. The paper also compares VLIW with other architectures such as Vector Machines and RISC architectures and points out main differences, advantages and disadvantages of some of these architectures
Relevance to the lecture
This paper is relevant to the lecture for several reasons. First, it introduced the concepts of VLIW architecture and discussed Trace scheduling algorithms. This paper is relevant to the second part of the lecture which had as main topic the discussion and main features of VLIW architecture and also comparison with Superscalar architecture. The second part of this part relates to Trace Scheduling algorithms which was the main topic of the very last part of the lecture.
Similarities and novelties with respect to the lecture
This paper was written at the beginning of the development of VLIW and does not present many new ideas that were not covered in the lecture. However it is interesting to refer to it because it was written by the person who first developed a VLIW processor. With respect to the main features of VLIW presented, it had the following properties:

"There is a control unit issuing a single long instruction per clock cycle.". This is basically true as far as VLIWs are concerned. VLIW does issue a single instruction containing many independent operations to be executed in parallel.
"Each operation required a statically predicable number of clock cycles to execute." This is true. In the lecture, it was discussed that VLIW processors define the dependencies before sending the code to be executed and during that stage it is also possible to predict what number of clock cycles the entire program code execution will take are.

The paper also makes comparisons regarding degree of parallelism of VLIW and Vector Machines. In this case, the paper suggested that Vector machines were a good alternative for VLIW in terms of offering good degree of parallelism. The main problem lies on the difficulty of programming in such architecture. There was also the problem with the code

in Vector Machines not being able to provide a satisfactory speed-up in parts of the code and this seems to be a great drawback with respect to VLIW performance.
Trace scheduling was dealt as an alternative to provide parallelism for a set of instructions. The paper explains that trace scheduling is based on processing instruction in advance such that long streams (traces) of code could be scheduled appropriated. It was presented an example of a code that contained no loops but rather an implementation of guessing the trace to choose. After this step is done, he introduces to notions of replacement and compensation code which is used to recover the code in case the trace taken was the one not predicted by the compiler. These were briefly discussed in the lecture and are further explained section 5.3 of the scribing and [176].

The paper also describes how the ELI – 512 bit works. It is basically a processor that contained 16 clusters, each with an ALU. Each ALU would perform a different operation. It also contained 32 register accesses and used pipeline implementation for memory reference instructions.

The main problems pointed out were putting tests on each instruction without making the machine too big (a problem with Superscalar implementation) and how to put enough memory reference in each instruction without making the machine too show(a problem shown in CISC architectures).

The paper concludes with alternatives to make the machine ELI – 512 able to access memory faster. A suggested implementation was using pipeline and was under discussion at the time. As for experimental results, it was attempted to obtain a speed-up of 10-30 over normal 2-3 of RISC architectures. The algorithm used was trace scheduling for locating and specifying parallelism

Paper 2: “Dynamically Scheduled VLIW Processors” [187]

Description of the paper
The focus of this paper is to present new means of solving the problem of incompatibility of VLIW Processors with other applications. For that purpose, it is introduced new techniques implemented in Superscalar processors such as delayed split-issue instructions and dynamic scheduling hardware. This paper starts by going over the basic definitions of VLIW, Superscalar, concepts related to MultiOp (multiple operations) and UniOp (one instruction), some properties of dynamically scheduled Superscalar features and introduces new terminologies used throughout the document such as UAL(unit assumed latencies) and NUAL(non-unit assumed latencies).
Relevance to the lecture
This paper is relevant to the lecture for many reasons. The first part of the lecture, listed many properties related to ILP implementations such as Superscalar and VLIW processors. This paper emphasises the main differences between VLIW and Superscalar and suggests approaches to make VLIW have more Superscalar features without losing its properties. Another point discussed was suggestion of a way of solving the problem of compatibility issues in VLIW processors. This is one of the main reasons why VLIW are not very well accepted or implemented nowadays.
Similarities and novelties with respect to the lecture
Rau’s paper introduces some new ideas regarding implantation of Dynamic Scheduled VLIW processor that is compatible to multiple hardware systems. This has certainly shown some promise but this would require some changes in the compilation system of the VLIW. Rau introduces that VLIW architectures are horizontal machines and that MultiOp machines consists of many operations whereas UniOp consists of a single operation. In this context, VLIW computer programs are scheduled with necessary knowledge of functional unit latencies. These computer programs are called NUAL when it can recognise latency and UAL when unit latencies are assumed for all functional units in VLIW design.

Rau also introduces the term split – issue which is the mean of correction of a NUAL program when the actual latencies do not agree with the theoretical or assumed latencies.

Rau concludes the paper explaining that if compatibility is to be implemented in VLIW that the main tool used for enabling VLIW to be dynamic scheduled is to use split- issue algorithms. Compatibility would be solved if MultiOp and NUAL were implemented together. Even though there is an introduction of new algorithms, a basic implementation for compatibility is to make scheduling in hardware.

Paper Analysis of Lecture #21b: Very Large Instruction Word (VLIW) Processors

Paper 1 Optimizing Loop Performance for Clustered VLIW Architectures [195]

This paper describes a new method to accurately predict the inter-cluster communication cost of a loop. Also this paper presents integer-optimized versions of loop unrolling and unroll-and-jam for clustered VLIW architectures.

DSP applications are demanding higher performance from embedded VLIW processors. But the embedded VLIW are constrained by power consumption and chip cost. To meet the demand VLIW processors have to increase instruction level parallelism without significantly raising power consumption and chip cost. Increasing the ILP of a VLIW processor will increase the size of the register bank. A size increase for the register bank will lead to higher chip cost and might also decrease performance. To increase ILP without significantly increasing the register bank a cluster VLIW processor can be used, like the TMS320C6000 processor. A clustered VLIW use several small register banks instead of one large one. One or more functional units will access one register bank directly. If a functional unit needs data from another register bank, the value has to be copied over to the local register bank. With clustered VLIW the compiler has to expose parallelism for maximal functional unit use and also keep the inter-cluster communication cost low.

Since DSP application spend a lot of time in loops, performance can be increased by using better loop transformation techniques. This paper presented a systematic way to unroll loops for clustered VLIW while taking into account the inter-communication cost. Unrolling VLIW loops is easier than unrolling clustered VLIW loops because VLIW compilers do not have to worry about the inter-communication cost

The clustered VLIW processor that was used in this paper was the same one that was covered in class, Texas Instruments TMS320C6000. The unrolling technique that was discussed in class was for VLIW processors, not for clustered VLIW. The paper focused on cluster VLIW processors specifically the Texas Instruments TMS320C6x family of processors. The unrolling technique that was discussed in class was fairly simple and there was no formal way to figure out the maximum number of iterations that can be unrolled. This paper formally introduced clustered VLIW processors and explained why they are used. Clustered VLIW were not really discussed in the lecture.

The idea that the register bank would be a limiting factor for the size of VLIW was discussed in the lecture notes. The idea of decentralizing the register bank was also discussed. The reason why loops are unrolled is the same for both clustered VLIW and VLIW.

Paper 2 A Code Generation Framework for VLIW Architectures with Partitioned Register Files [196]

This paper discusses an approach to partitioning values among several register banks. As computers take increased advantage of instruction level parallelisms it becomes more difficult to maintain a single register bank and a high clock rate. The keep the clock rate high the register bank can be partitioned among several different register banks. Each register bank can only be accessed by a subset of the functional units. If a functional unit wants data from another register bank, it will have to obtain a copy of the data and save it in its local register bank. The job of the compiler becomes more difficult because of the partitioned register bank. The processor has to maximize parallelism and limit the inter-register bank copies to a minimum. Partitioning the register file may reduce instruction level parallelism, if the data resides in a register bank that the functional unit cannot access.

The ideas of increasing ILP and partitioning the register banks have opposite goals. To achieve high parallelism the work should be equally distributed over all functional units. But to reduce communication costs the compiler might favor a group of functional units that are connected to the same register bank. The authors' try to solve this problem by building a graph called the register component graph. The graphs nodes represent the register and the edges represent the two registers that appear in the same instruction. After the register component graph is created the unconnected values in the graph would be good candidates to assign to separate register banks. Also, values with low costs can also be assigned to a separate register bank. The big advantage of register component graphs is that it abstracts machine dependent details into costs.

The paper also talked about an experiment where the authors implemented the algorithm discussed in this paper on a C compiler for FORTRAN. The results showed that the algorithm did a good job limiting inter-bank copies of values, 10% more than the benchmark. The execution cycle results were not as good. This is something the authors did not expect. The authors hypothesized a low inter-register bank copies would lead to a low execution cycle.

The subject that this paper discussed was not covered in class. But this paper is an extension to what was discussed in class. The lectures discussed how the register bank limits the scalability of the VLIW processor. The lecture also discussed that large register banks are expensive and lead to lower performance. Using a partitioned register bank can solve these problems.

Paper Analysis of Lecture #22 Superscalar Processors

PAPER 1 SUPERSCALAR VS. SUPERPIPELINED MACHINES.

NORMAN P. JOUPPI

Both superscalar and superpipelined machines try to exploit instruction-level parallelism. The first one by issuing several instructions per clock cycle and the second one by issuing only one instruction per clock cycle but using shorter ones.

Defining a base machine with the parameters below will help us to compare performances in instruction-level parallelism. The base machine is defined like this:
• Instructions issued per cycle = 1
• Simple operation latency measured in cycles = 1
• Simple operation latency measured in instructions = 1

A superscalar machine of degree *n* can issue *n* instructions per cycle. So considering the set of instructions below and *n*>=3, we can affirm this:

```
|  Load Cl <- 23 (R2)  |
|   Add R3 <- R3 + 1   |
| FPAdd C4 <- C4 +d C3 |
0                      1 cycle
```

• Instructions issued per cycle = n
• Simple operation latency measured in cycles = 1
• Simple operation latency measured in instructions = n

A superpipelined machine of degree *m* has a cycle time of 1/*m*. Considering again the instructions above we can say this:

```
|  Load Cl <- 23 (R2)  |
        |   Add R3 <- R3 + 1 |
                | FPAdd C4 <- C4 +d C3   |
0                      1 base cycle time
0       1       2      3 superpipelined cycles, m = 3
```

• Instructions issued per cycle = 1, but the cycle time is *1/m* of the base machine
• Simple operation latency measured in cycles = m
• Simple operation latency measured in instructions = m

When comparing superscalar and superpipelined machines of equal degree, we have to know that the performances of both will be similar.
One of the differences between these architectures is that in superscalar scheme resources conflicts can happen. This is not a problem in superpipelined machines since the functional units are pipelined.
Another difference is the complexity of the hardware when we want to transform a base machine into a superpipelined or superscalar machine. In the first case, simplifying the whole procedure, we can say that it is necessary to put m-1 latches between every pair of latches in

the base machine. In the second case it is necessary to duplicate at least the register file ports, bypasses, busses and instruction decode logic. In summary, the superscalar machine requires much more hardware that increases the cycle time decreasing the performance and making the superpipelined machine more tempting.

To conclude, we want to introduce a scheme mix of the ones studied. Due to the cycle time and the number of instructions issued per cycle are theoretically orthogonal, we could have a superpipelined superscalar machine which would be able to perform several instructions at the same time each fraction of clock cycle.

This document is an extension of some topics studied in the lecture 24. The concepts are the superscalar machines and pipelined machines. This appendix helps us to understand differences between these two schemes.

PAPER 2 THE EFFECT OF EMPLOYING ADVANCED BRANCHING MECHANISMS IN SUPERSCALAR PROCESSORS

Yen-Jen Oyang, Chun-Hung Wen, Yu-Fen Chen, and Shu-May Lin

In superscalar processors an important amount of clock cycles are wasted due to dependencies among the programs instructions. One of these dependencies is the one due to branch operations. Reducing this kind of operations improve the superscalar processors performance. This paper describes and compares branching mechanisms to achieve this goal.

The first mechanism consists in adding multiple condition registers to the registers that already exist in the processor. When there are several conditions to check followed by one branch instruction depending on them we need to check the conditions in order to know which action should be done after. Having several condition registers a set of the conditions (or even all of them) can be checked at the same time, determining in one step what should be done as opposite to having only one condition register, where all the conditions have to be checked one by one.

The second one is a multi-way branching. It can be seen as a concurrent execution of traditional conditional branch instructions. Several conditional branch instructions are grouped and executed in parallel. First of all, instructions are dispatched to certain functional units to evaluate the conditions concurrently. The condition satisfied is selected and the destination address is loaded in the program counter. The conditions grouped must be mutual exclusive in order to ensure that only one of them will satisfy the condition. But a drawback of this scheme is that for a maximum effectiveness the processor should have multiple condition registers to apply this multi-way branching to all the conditions.

To study the effectiveness of these schemes we will study the performance of three different superscalar processor models with theses characteristics: model 1 does not employ multiple condition registers nor multi-way branching mechanism; model 2 employs multiple condition registers but not the multi-way branching mechanism; model 3 employs multiple condition registers and the multi-way branching mechanism.

Some of the results observed were that incorporating multiple condition registers increase the superscalar processor performance by a factor between 16% to 20% while the proposed multi-way branching mechanism improve the superscalar processor performance by 18%, 37% and 47% for machines with 2, 3, and 4 execution units respectively.

There is an interesting result that can be observed after the tests. The performance improvement achieved when using the branching mechanisms explained is bigger in machines with more execution units. This is due to the fact that a machine with several execution units able to exploit the instruction-level parallelism effectively and to the fact that the reduction of the number of branch operations has more impact on performance of superscalar processors with more execution units.

The main conclusion, is that the advanced branching mechanisms produce a more important improvement for machines configurated with more execution units. The first reason is, clearly, that a machine with, for example, 2 execution units, cannot execute more than two instructions at the same time, so the improvement in the branch prediction is not really that important.

Besides, these advanced mechanisms for branching decrease the grade of dependency in the program.

Paper Analysis of Lecture #23: Superscalar Processors – Review processors

Paper 1: The Microarchitecture of Superscalar Processors

by JAMES E. SMITH and GURINDAR S. SOHI

The paper "The Microarchitecture of Superscalar Processors" presents an overview of Superscalar processors in 1990's in the following areas: (1) introduction to the concepts and techniques for superscalar processing; (2) the general problems could be solved using superscalar processors; (3) specific techniques were used in typical superscalar microprocessors; (4) three recent superscalar processors in 1990's: MIPS RlOOOO, DEC Alpha 21 164, and AMD K5; (5) future directions for instruction level parallelism.

1. Introduction to superscalar processing

This section first introduces the work flow for superscalar processor and the most important features in superscalar processing. Superscalar processing can initiate multiple instructions in the same clock cycle and schedule instruction dynamically. A typical work flow for superscalar processor: fetches and decodes the several instructions at a time, predicts the outcomes of conditional branch instructions in advance, analyzes for data dependences, distributes instructions to functional units, initiates the instruction for execution in parallel based on the availability of operand data, and re-sequences instruction results on completion. Then it reviews the history of Instruction level parallelism in the form of pipelining. Following this part, it discusses the traditional Instruction Processing Model: sequential execution model and Instruction Processing Model for superscalar processing. Finally it covers the necessary elements of high performance processing: (1) Instruction fetch strategies; (2) Methods for determining true dependences; (3) Methods for initiating, or issuing, multiple instructions in parallel; (4) Resources for parallel execution of many instructions; (5) Methods for communicating data values through memory via load and store instructions and memory interfaces; (6) Methods for committing the process state in correct order.

2. The general problems solved using superscalar processors

In this section, an example was used to show how the superscalar processor executes this program and how the superscalar processor resolves control dependence. Figure 6 [200] shows the program in C language and assembly language. Figure 7 shows the dependences existing in this program. Figure 8 shows the sequence of program execution and the window of executions. When author analyzed the code, he introduced several important concepts covered in the class: true dependence, anti dependence, and output dependence. True dependences can cause a read after write (RAW) data hazard. Anti-dependences can cause write after read (WAR) data hazard. Output dependence can cause write after write (WAW) data hazard. RAW occurs when a consuming instruction reads a value before the producing instruction writes it. WAR occurs when an instruction writes a value before a preceding instruction reads it. WAW occurs when multiple instructions update the same storage location but not in the proper order.

```
for (i=0; i<last; i++) {
  if (a[i] > a[i+1]) {
    temp = a[i];
    a[i] = a[i+1];
    a[i+1] = temp;
    change++;
  }
}
```

(a)

```
L2:
    move    r3,r7       #r3->a[i]
    lw      r8,(r3)     #load a[i]
    add     r3,r3,4     #r3->a[i+1]
    lw      r9,(r3)     #load a[i+1]
    ble     r8,r9,L3    #branch a[i]>a[i+1]

    move    r3,r7       #r3->a[i]
    sw      r9,(r3)     #store a[i]
    add     r3,r3,4     #r3->a[i+1]
    sw      r8,(r3)     #store a[i+1]
    add     r5,r5,1     #change++
L3:
    add     r6,r6,1     #i++
    add     r7,r7,4     #r4->a[i]
    blt     r6,r4,L2    #branch i<last
```

(b)

Fig. 6. (a) A high level language program written in C; (b) its compilation into a static assembly language program (unoptimized). This code is a part of a sort routine; adjacent values in array $a[\,]$ are compared and switched if $a[i] > a[i+1]$. The variable change keeps track of the number of switches (if change $= 0$ at the end of a pass through the array, then the array is sorted.) **[1]**

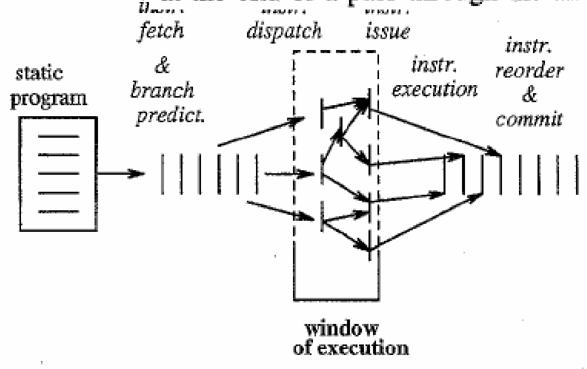
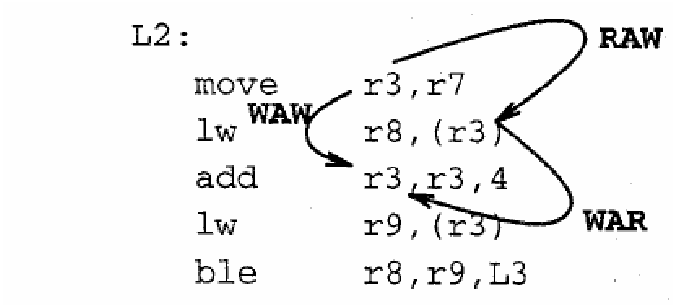


Fig. 7. A conceptual figure of superscalar execution. Processing phases are listed across the top of the figure. **[1]**



```
L2:
    move    r3,r7
    lw      r8,(r3)
    add     r3,r3,4
    lw      r9,(r3)
    ble     r8,r9,L3
```

Fig. 8. Example of data hazards involving registers. **[1]**

3. Typical superscalar microprocessor architecture

This section introduces the architecture and hardware components and their functions for superscalar processor architecture. The typical superscalar processor architecture, shown in Figure 9 [200], was used in this section as an example for specifying superscalar processor hardware components and their functions. From Figure 9, we can see that the major hardware components for superscalar processor include: pre-decode unit, instruction cache unit, instruction buffer unit, decode, rename and dispatch unit, register files units for float and integer, instruction buffers units for float, integer and address, multiple function units, re-order and commit unit, and memory interface unit. This section covers all the procedures and most important techniques used in execution of instruction in superscalar microprocessor. The procedures are: instructions fetch and decode, branch prediction, data dependence analysis, instruction issue and execution, memory operation, instruction reorder and commit. The techniques include: control dependence, register renaming, single queue, multiple queue and Reservation Stations for issues the instruction in-order/out-order.

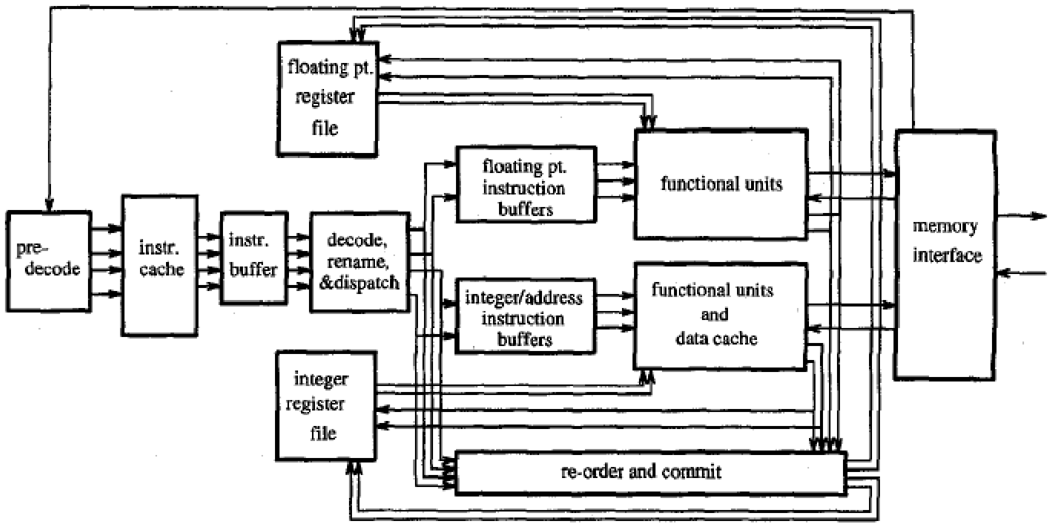

floating pt. register file
floating pt. instruction buffers
functional units
pre-decode
instr. cache
instr. buffer
decode, rename, &dispatch
memory interface
integer/address instruction buffers
functional units and data cache
integer register file
re-order and commit

Fig. 9. Organization of a superscalar processor. Multiple paths connecting units are used to illustrate a typical level of parallelism. For example, four instructions can be fetched in parallel, two integer/address instruction scan issue in parallel, two floating point instructions can complete in parallel, etc. [1]

4. Three superscalar processors and future for superscalar processor

In the last two sections of this paper, author described three superscalar processors using architecture and the techniques reviewed in the previous section and future for superscalar processor. MIPS RlOOOO superscalar processor uses the very similar framework in this paper and it uses the extensive dynamic scheduling. DEC Alpha 21 164 is a simplified superscalar processors. It supports high clock rate instead of using extensive dynamic scheduling. AMD K5 uses the complex instruction set which was originally not used in the pipeline.

5. Relevance to the lecture

This paper is related to this lecture in the following aspects. The first section of this paper introduced the typical work flow for superscalar processor: instruction fetches, decodes, executes, accesses memory, and writes back. It also pointed that the most important feature of superscalar processor is instruction level paralleling. The second section of this paper explained how the instruction level paralleling works in superscalar processor works by an example of analysis of data dependence, execution of windows and the techniques and algorithm for parallel execution schedule. The third section of this paper investigated superscalar processor architecture and hardware components. The hardware components are: pre-decode unit, instruction cache unit, instruction buffer unit, decode, rename and dispatch unit, register files units for float and integer, instruction buffers units for float, integer and address, multiple function units, re-order and commit unit, and memory interface unit. The new points in this paper is that it introduced the parallel execution schedule with different approach: single queue, multiple queue and Reservation Stations for issues the instruction in-order/out-order.

Paper 2: Parallelism exploitation in superscalar Multiprocessing

by N.P. Lu and C-P. Chung

Description of paper

This research paper proposes simulator models for evaluating superscalar multiprocessor systems where it can exploit instruction level parallelism, and task level parallelism. The simulator model collected in various configurations to investigate the parallelism exploitation of superscalar multiprocessor systems. The simulator model results that by using a moderate degree of superscalar processing and a high degree of multiprocessing to exploit the instruction level and task-level parallelism. The superscalar multiprocessor simulator used in this research paper was SMINT which is superscalar multiprocessor simulator based on MINT. Figure 5 shows the SMINT features such as superscalar execution, dynamic instruction scheduling, register renaming and dynamic branch prediction and speculative execution.

Relevancy to lecture

The paper talks about instruction parallelism and task parallelism. We learned in instruction parallelism in superscalar lecture class and task parallelism in scheduling lecture class. Even though this paper uses the simulator model to exploit the parallelism, it remains connected to our core concept of superscalar processor. From the paper models, we show how prediction affecting instruction level parallelism. As we learned from the class notes that larger the instruction window, the more exploitable instruction-level parallelism.

One of the models shows how instruction window size affecting instruction-level parallelism. We know from the lecture notes that larger instruction window size, the more exploitable instruction-level parallelism. We have trade-off between the size of instruction level window and the instruction level parallelism since its circuit complexity grows with the size of instruction window. Another fact revealed from this model result that increased exploitable instruction level parallelism is very insignificant after the instruction window size grows above certain threshold value.

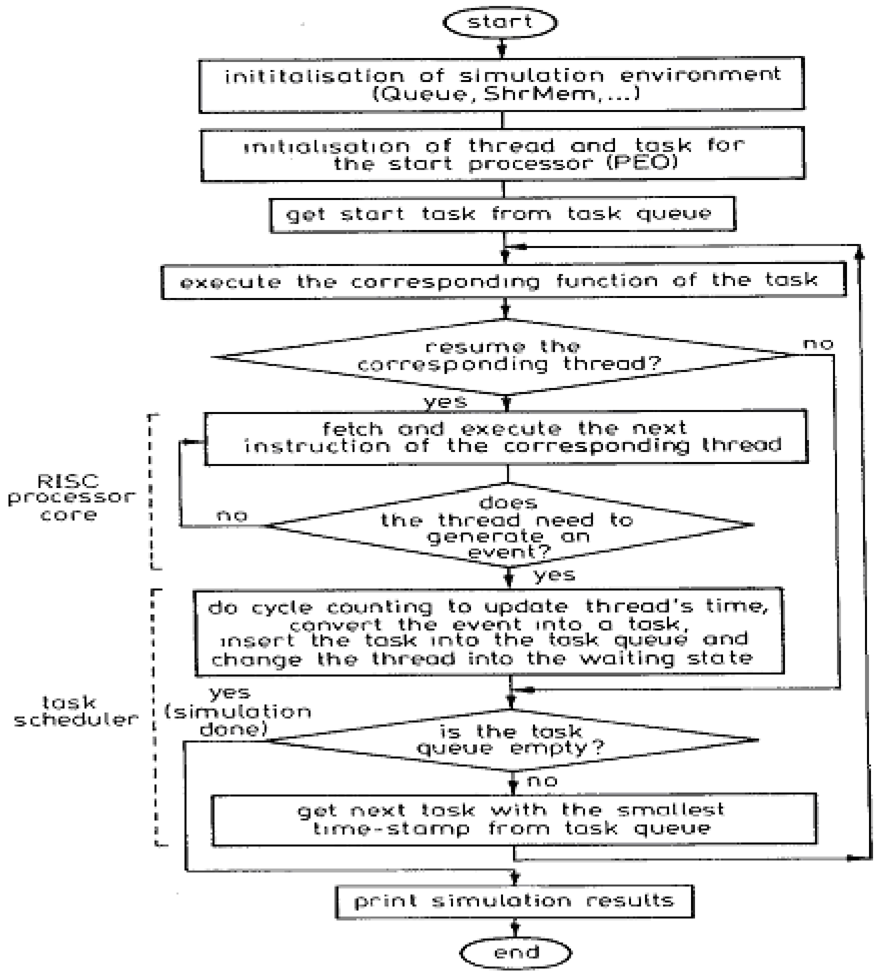

start
inititalisation of simulation environment (Queue, ShrMem,...)
initialisation of thread and task for the start processor (PE0)
get start task from task queue
execute the corresponding function of the task
resume the corresponding thread?
no
yes
fetch and execute the next instruction of the corresponding thread
RISC processor core
does the thread need to generate an event?
no
yes
do cycle counting to update thread's time, convert the event into a task, insert the task into the task queue and change the thread into the waiting state
task scheduler
yes (simulation done)
is the task queue empty?
no
get next task with the smallest time-stamp from task queue
print simulation results
end

Figure 5 Simulation flow of MINT [203]

References

[1] Altera Corporation. "Straftix Device Family Overview". http://www.altera.com/products/devices/stratix/overview/stx-overview.html.

[2] Altera Corporation. "TriMatrix Memory in Stratix Devices. http://www.altera.com/products/devices/stratix/features/stx-trimatrix.html

[3] Altera Corp., Stratix Module 2: Logic Structure & MultiTrack Interconnect, 2004.

[4] Altera Corp., **Nios II Software Developer's Handbook**, 2005.

[5] Field-programmable gate array, http://en.wikipedia.org/wiki/FPGA

[6] NOIS Architecture CcEG4131 lecture notes

[7] R. Saleh, S. Wilton, S. Mirabbasi, A. Hu, M. Greenstreet, G. Lemieux, P. Pande, C. Grecu, A. Ivanov, "System-on-Chip: Reuse and Integration", Proceedings of the IEEE, Vol. 94, No. 6, June 2006, pp. 1050-1069. [pdf]

[8] System-on-chip: reuse and integration Saleh, R. Wilton, S. Mirabbasi, S. Hu, A. Greenstreet, M. Lemieux, G. Pande, P.P. Grecu, C. Ivanov, A. Dept. of Electr. & Comput. Eng., British Columbia Univ., Canada This paper appears in: Proceedings of the IEEE Publication Date: June 2006 Volume: 94 , Issue: 6 On page(s): 1050 - 1069 ISSN: 0018-9219 INSPEC Accession Number:8985242 Digital Object Identifier: 10.1109/JPROC.2006.873611 Posted online: 2006-07-10 09:48:32.0 http://ieeexplore.ieee.org/search/srchabstract.jsp?arnumber=1652898&isnumber=34642&punumber=5&k2dockey=1652898@ieeejrns&query=%28+soft+core%3Cin%3Emetadata+%29&pos=3

[9] Design considerations for soft embedded programmable logic cores Wilton, S.J.E. Kafafi, N. Wu, J.C.H. Bozman, K.A. Aken'Ova, V.O. Saleh, R. Dept. of Electr. & Comput. Eng., Univ. of British Columbia, Vancouver, BC, Canada This paper appears in: Solid-State Circuits, IEEE Journal of Publication Date: Feb. 2005 Volume: 40 , Issue: 2 On page(s): 485 - 497 ISSN: 0018-9200 INSPEC Accession Number:8276592 Digital Object Identifier: 10.1109/JSSC.2004.841038 Posted online: 2005-01-31 08:28:33.0 http://ieeexplore.ieee.org/search/srchabstract.jsp?arnumber=1388638&isnumber=30218&punumber=4&k2dockey=1388638@ieeejrns&query=%28+soft+core%3Cin%3Emetadata+%29&pos=9

[10] Nios development board, Stratix edition, reference manual: http://www.altera.com/literature/hb/nios2/n2sw_nii5v2.pdf

[11] Nios II Processor Architecture and Programming: CEG4131 course notes by Miodrag Bolic, http://www.site.uottawa.ca/~mbolic/ceg4131/ceg4131_nios2.ppt

[12] TriMatrix Memory in Stratix Device, Altera, http://www.altera.com/products/devices/stratix2/features/memory/st2-memory.html NiosII processor reference handbook http://www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf Hard IP core: http://www.mips.com/content/Products/Cores/HardIPCores/content_html Various definitions: http://www.wikipedia.org

[13] Reconfigurable computing: architectures and design methods: by T.J. Todman, G.A. Constantinides, S.J.E. Wilton, O. Mencer, W. Luk and P.Y.K. Cheung. It's published in IEE Proceedings - Computers and Digital Techniques, Volume 152, Issue 2, page261 to 272, in Mar 2005 Network-on-chip architectures and design methods: by L. Benini and D. Bertozzi. It's published on IEE Proceedings - Computers and Digital Techniques, Volume 152, Issue 2, page193 to 207, in Mar 2005

[14] Ruud van der Pas. (2002, November) *Memory Hierarchy in Cache baced systems* [Online] . avaliable: http://www.sun.com/blueprints/1102/817-0742.pdf#search=%22memory%20hierarchy%22

[15] Stephen J. Bigelow, "PC Hardware Desk Reference". Osborne: McGraw-Hill, 2003 pp. 959

[16] David A. Patterson, John L.Hennessy, "Computer Organization and Design" 3rd edition, Morgan Kaufmann Publishing, 2005

[17] Miodrag Bolic (2006, September) *Revision of cache memories* [Online], available: http://www.site.uottawa.ca/~mbolic/ceg4131/index.shtml

[18] [Online] available: http://wearcam.org/ece385/lecturecaches1/

[19] Abraham Silberschatz, Peter Baer Galvin, "Operating System concepts" Greg Gagne, John Wiley and Sons inc. 2005

[20] Altera Corporation (2006, May) NOIS II Processor Reference Handbook [Online] http://www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf

[21] Nihar R. Mahapatra and Balakrishna Venkatrao The Processor-Memory Bottleneck: Problems and Solutions http://www.acm.org/crossroads/xrds5-3/pmgap.html

[22] James D. Fix, Cache Performance Analysis of Algorithms, 2002 http://portal.acm.org/citation.cfm?coll=GUIDE&dl=GUIDE&id=935223

[23] Ruth E. Anderson, Thu D. Nguyen, and John Zahorjan. (1999, April). Cascaded Execution: Speeding Up Unparallelized Execution on Shared-Memory Multiprocessors. Seattle, Washington. [Online]. Available: http://ieeexplore.ieee.org/iel4/6155/16457/00760554.pdf?tp=&arnumber=760554&isnumber=16457

[24] John L. Gustafson. (1992, January). The Consequences of Fixed Time Performance Measurement. Ames, Iowa. [Online] Available: http://ieeexplore.ieee.org/iel2/378/4717/00183285.pdf?tp=&arnumber=183285&isnumber=4717

[25] T. N. Venkatesh, V. R. Sarasamma', Rajalakshmyl S., Kirti Chandra Sahu' and Rama Govindarajan. (2005, February). Super-linear speed-up of a parallel multigrid Navier-Stokes solver on Flosolver. Current Science (Vol. 88). [Online]. pp. 589-593. Available: http://nal-ir.nal.res.in/785/01/curr25feba.pdf

[26] Various authors. Speedup. [Online]. Available: http://en.wikipedia.org/wiki/Speedup

[27] Various authors. Performance Analysis. [Online]. Available: http://en.wikipedia.org/wiki/Performance_Analysis

[28] Various authors. Parallel Computing. [Online]. Available: http://en.wikipedia.org/wiki/Parallel_computing

[29] Various authors. Standard Performance Evaluation Corporation. [Online]. Available:http://en.wikipedia.org/wiki/Standard_Performance_Evaluation_Corporation

[30] Various authors. EEMBC. [Online]. Available: http://en.wikipedia.org/wiki/EEMBC

[31] Miodrag Bolic. Performance Analysis of Multiprocessor Architectures. [Online]. Available: http://www.site.uottawa.ca/~mbolic/ceg4131/ceg4131_performance.ppt

[32] Hesham El-Rewini, Speedup. [Online]. Available: http://www.site.uottawa.ca/~mbolic/ceg4131/ceg4131_speedup.ppt

[33] EEMBC. GrinderBench. [Online] Available: http://www.grinderbench.com/

[34] H. El-Rewini, M. Abd-El-Barr, *Advanced Computer Architercture and Parallel Processing*. Hoboken, NJ: John Wiley & Sons, 2005, pp.1-15

[35] Wikipedia, "Flynn's Taxonomy". http://en.wikipedia.org/wiki/Flynn's_taxonomy, last accessed September 24, 2006

[36] M. Bolic "Parallel Computer Models" (CEG4131 Computer Architecture III Lecture Slides) http://www.site.uottawa.ca/~mbolic/ceg4131/index.shtml, last accessed September 22, 2006

[37] H. El-Rewini, M. Abd-El-Barr, *Advanced Computer Architercture and Parallel Processing*. Hoboken, NJ: John Wiley & Sons, 2005, pp.103-105

[38] Wikipedia, "Very Long Instruction Word". http://en.wikipedia.org/wiki/VLIW, last accessed September 24, 2006

[39] H. El-Rewini and M. Abd-El-Barr, *Advanced Computer Architecture and Parallel Processing*. Hoboken, New Jersey: John Wiley and Sons, 2005.

[40] M. Bolic. CEG4131. Class Lecture, Topic: "Dynamic Interconnection Networks Buses." CBY D207, Faculty of Engineering, University of Ottawa, Ottawa, Canada, Sept. 27, 2006.

[41] Altera Corp., "Avalon Interface Specification", 2005; available at http://www.altera.com/literature/manual/mnl_avalon_spec.pdf (last accessed October 15, 2006).

[42] T. Dysart, "Interfacing processors, memory and peripherals Part 2", April 3, 2006; available at http://www.cse.nd.edu/courses/cse30322/www/Notes/IO2_2up.pdf (last accessed October 15, 2006).

[43] I. Sander, "Buses", November 1, 2004; available at http://www.imit.kth.se/courses/2B1447/Lectures/2B1447_L4_Buses.pdf (last accessed October 15, 2006).

[44] Altera Corp., "Nios II Processor Family Questions & Answers", April 26, 2006; available at http://www.altera.com/products/ip/processors/nios2/benefits/ni2-q-and-a.html (last accessed October 1, 2006).

[45] F. E. Guibaly, "Design and Analysis of Arbitration Protocols," *IEEE Trans. Comput.*, vol. 38, pp. 161–171, Feb. 1989.

[46] S. Winegarden, “Bus Architecture of a System on a Chip with User-Configurable System Logic,” IEEE Journal of Solid-State Circuits, vol. 35, pp. 425–433, Mar. 2000.

[47] H. El-Rewini and M. Abd-El-Barr, “Advanced Computer Architecture and Parallel Processing”, John Wiley and Sons, 2005, pp. 19-24

[48] Altera, (September 29, 2006). Avalon Interface Specification [Online] Available: http://www.altera.com, http://www.altera.com/literature/manual/mnl_avalon_spec.pdf#search=%22Avalon%20Bus%22

[49] L.N. Bhuyan, Y. Qing, D.P. Agrawal, (1989 February). “Performance of multiprocessor interconnection networks”, in Computer, volume 22, 1989, pp. 25-37 Available: www.ieee.org

[50] S.M. Mahmud, (1994 July). “Performance Analysis of Multilevel Bus Networks for Hierarchical Multiprocessors”, in IEEE TRANSACTIONS ON COMPUTERS, volume 43, 1994, pp. 789-805 Available: www.ieee.org

[51] K. Hwang, *Advanced Computer Architecture Parallelism, Scalability, Programmability*. McGraw-Hill, 1993.

[52] Sven Brehmer, PolyCore Software, Inc.. Foster City, CA, USA, “On-Chip Interconnects for Multi-Core Chips: A Software Perspective”, May 8, 2006; available at http://www.soccentral.com/results.asp?CatID=488&EntryID=18919 (last accessed October 29, 2006)

[53] University of Illinois at Urbana-Champaign Department of Computer Science, Urbana, IL, USA, “ILLIAC 6 MACHINE ORGANIZATION”, August 25, 2005; available at: http://illiac6.cs.uiuc.edu/hardware.html#topology (last accessed October 29, 2006)

[54] Jonathan Turner and Naoaki Yamanaka, *Architectural Choices in Large Scale ATM Switches*, May 1, 1997; available at: http://citeseer.ist.psu.edu/cache/papers/cs/521/http:zSzzSzwww.cs.wustl.eduzSzcszSztechreportszSz1997zSzwucs-97-21.pdf/turner98architectural.pdf (last accessed October 29, 2006)

[55] Goyal, A. and Agerwala, “T. Performance Analysis of Future Shared Storage Systems”, *IBM Journal of Research and Development*, pp. 95-107, January, 1984; available at: http://www.research.ibm.com/journal/rd/281/ibmrd2801J.pdf (last accessed October 29, 2006)

[56] H. El-Rewini, M. Abd-El-Barr, *Advanced Computer Architecture and Parallel Processing*. Hoboken, NJ: John Wiley & Sons, 2005, pp.29-33

[57] M. Bolic. (September 2006). *Dynamic Interconnection Networks*. [Online]. Available: http://www.site.uottawa.ca/~mbolic/ceg4131/ceg4131_dynamic_networks.ppt

[58] H. El-Rewini and M. Abd-El-Barr, *Advanced Computer Architecture and Parallel Processing*, Hoboken, NJ: John Wiley & Sons, Inc., 2005, pp. 19-33.

[59] K. Hwang, *Advanced Computer Architecture: Parallelism, Scalability, and Programmability,* New York, NY: McGraw-Hill Inc., 1993.

[60] M. T. Goodrich and R. Tamassia, *Data Structures and Algorithms in Java*, 2nd ed., John Wiley & Sons, Inc., 2001, pp 585-596.

[61] PMC-Sierra Inc., *PMC-Sierra Samples RM9000x2 64-Bit MIPS-based Multiprocessor Running at 1.0 Gigahertz with Integrated Memory and I/O Interfaces*. [Online]. Available: http://investor.pmc-sierra.com/phoenix.zhtml?c=74533&p=irol-newsArticle&ID=343893&highlight=

[62] V. Chandramouli and C.S. Raghayendra, *"Nonblocking Properties of Interconnection Networks"*, IEEE Trans. Communication, vol 43, pp 1793-1799, Feb-Mar-Apr 1995.

[63] Hongbing Fan and Yu-Liang Wu, *"Crossbar based design schemes for switch boxes and programmable interconnection networks"*, Design Automation Conference, 2005, vol 2, pp 910-915, 18-21 Jan 2005.

[64] *Two-Dimensional Array of Processing Elements for Emulating a Multi-Dimensional Network;* www.freepatentsonline.com; US PAT. 5,058,001

[65] *Interconnection Networks (Course Notes);* Benjamin Macey; University of Western Australia; http://www.ee.uwa.edu.au/~maceyb/aca319-2003/index.html; 2002

[66] El-Rewini, Adb-El-Barr; *Advanced Computer Architecture and Parallel Processing*; Wiley Interscience; Hoboken, New Jersey, 2005

[67] Laxmi N. Bhuyan, (University of Southwestern Louisiana), Qing Yang, (University of Rhode Island), Dharma P. Agrawal, (North Carolina State University); *Performance of Multiprocessor Interconnection Networks,* Pages; 25-37, can be found at IEEE archives; http://ieeexplore.ieee.org/iel1/2/783/00019830.pdf

[68] Dowd, P.W.; Dowd, M.; Jabbour, K., *Static interconnection network extensibility based on marginal performance/cost analysis,* Pages; 9 - 15 can be found at IEEE archives, http://ieeexplore.ieee.org/iel5/2192/461/00009087.pdf

[69] J. Kowalczyk "Multiprocessor Systems Xilinx" [Online document], 2003, [cited 2006 Oct 14], Available HTTP: http://www.xilinx.com/bvdocs/whitepapers/wp162.pdf

[70] D. Culler, J. P. Singh, "Parallel Computer Architectures: A Hardware/Software Approach", San Francisco, Unites States of America: Morgan Kaufmann Publishers, 1998

[71] M. Bolic "CEG4131 - Shared-Memory Architectures" [Online document], 2006, [cited 2006 Oct 14], Available HTTP: http://www.site.uottawa.ca/~mbolic/ceg4131/index.shtml

[72] Altera Corp. "Creating Multiprocessor NIOS II System Tutorial" [Online document], 2005, [cited 2006 Oct 14], Available HTTP: http://www.altera.com/literature/tt/tt_nios2_multiprocessor_tutorial.pdf,

[73] ATI tech. "Cross Fire, multiprocessor GPU technology" [Online document], 2006, [cited 2006 Oct 14], Available HTTP: http://www.ati.com/technology/crossfire/index.html

[74] Wikipedia. *OpenMP* , last accessed October 20, 2006 *[Online].Available:* http://en.wikipedia.org/wiki/OpenMP

[75] Blaise Barney. (2006, August 8). *OpenMP*. [Online]. Available: http://www.llnl.gov/computing/tutorials/openMP/ .

[76] Feng Liu and Vipin Chaudhary. *A Practical OpenMP Compiler for System on Chips.* WOMPAT 2003, LNCS 2716, pp. 54-68, 2003.

[77] Tim Mattson and Rudolf Eigenmann. *OpenMP: An API for Writing Portable SMP Application Software.*

[78] Yoshihiko Hotta, Mitsuhisa Sato, Yoshihiro Nakajima, Yoshinori Ojima. *OpenMP Implementation and Performance on Embedded Renesas M32R Chip Multiprocessor*, Japan, 2002 .

[79] Veljko Milutinovic, “Some Solutions for Critical Problems In The Theory and Practice of Distributed Shared Memory: New Ideas to Analyse” (http://tab.computer.org/tcca/news/sept96/dsmideas.pdf).

[80] Gary Graunke and ShreeKant Thakkar, “Synchronization Algorithms for Shared-Memory Mulitprocessors”http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=55501

[81] B. Barney, Summary of design for Parallelism, available at: http://www.llnl.gov/computing/tutorials/parallel_comp/

[82] H. El-Rewini and M. Abd-El-Barr, Advanced Computer Architecture and Parallel Processing. Hoboken, New Jersey: John Wiley and Sons, 2005.

[83] P. Paulin,C. Pilkington, M.Langevin, E. Bensoudane, D. Lyonnard, O, Benny, B. Lavigueur, D. Lo, G. Beltrame, V. Gagné, and G. Nicolescu, Parallel Programming Models for a Multiprocessor

[84] SoC Platform Applied to Networking and Multimedia, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 14, No. 7, July 2006.

[85] L. Hammond, B. Hubbert, M. Siu, M. Prabhu, M. Chen, K. Olukotun, The Stanford Hydra CMP, Stanford University, IEEE Micro 2000.

[86] Wikipedia , “ThreadsModel”, Wikipedia available at http://en.wikipedia.org/wiki/threadsModel (last accessed November 10,2006).

[87] Models, “Introduction to Parallel Computing”, llnl, available at http://www.llnl.gov/computing/tutorials/parallel_comp/#Models (last accessed November 8, 2006).

[88] Definition, “Shared Memory”, searchsmb, available at http://searchsmb.techtarget.com/sDefinition/0,290660,sid44_gci212976,00.html (last accessed November7, 2006).

[89] Shared memory, “Shared Memory Introduction”, kohala, available at http://www.kohala.com/start/unpv22e/unpv22e.chap12.pdf (last accessed November 10, 2006).

[90] SPMD, “Single Program Multiple Data Stream”, llnl.gov, available at

[91] http://www.llnl.gov/casc/Overture/henshaw/documentation/App/manual/node36.html, (last accessed November 11, 2006).

[92] Claudia Leopold, “Parallel and Distributed Computing”, John Wiley and Sons, 2001. pp.45-54

[93] H. El-Rewini and M. Abd-El-Barr, *"Advanced Computer Architecture and Parallel Processing"*. Hoboken, New Jersey: John Wiley and Sons, 2005, pp.84-95.

[94] Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, "Operating System Concepts", New Jersey: John Wiley and Sons, 2005, pp.54-63.

[95] Qing Yang *Member, IEEE*, George Thangadurai, and Laxmi N. Bhuyan, *Senior Member, IEEE*. "*Design of an Adaptive Cache Coherence Protocol for Large Scale Multiprocessor*" IEEE TRANSACTIONS ON PARRALLEL AND DISTRUBUTED SYSTEMS, VOL 3, NO. 3, MAY 1992.

[96] Patterson, David A. and Hennessy, John L. Computer Organization And Design. New-Delhi, India: Nutech Photolithographers, 2005.

[97] Adb-El-Barr, Mostafa and El-Rewini, Hesham. Advanced Computer Architecture and Parallel Processing. Hoboken, NJ: John Wiley & Sons, Inc., 2005.

[98] Bolic, Miodrag. (2006). [Online]. CEG4131: Cache Coherence, Additional Materials. Available: www.site.uottawa.ca/~mbolic/CEG4131 November 3, 2006. [date accessed]

[99] Michel Dubois, *Member, IEEE*, and Faye A. Briggs *Member, IEEE*, "*Effects of Cache Coherency in Multiprocessors*", IEEE TRANSACTIONS ON COMPUTERS, VOL. C-3 1, NO. 1 1, NOVEMBER 1982

[100] Wikipedia, "Cache Coherence", 2006, available at http://en.wikipedia.org/wiki/Cache_coherence, (last accessed October 30, 2006)

[101] M. R. Zargham, *Computer Architecture Single and Parallel Systems*. Upper Saddle River, New Jersey: Prentice-Hall, Inc,1996, pp.252-257

[102] H. El-Rewini, M. Abd-El-Barr, *Advanced Computer Architercture and Parallel Processing*. Hoboken, NJ: John Wiley & Sons, 2005, pp.89-96

[103] M. Bolic. CEG4131. Class Lecture, Topic: "Cache Coherence." CBY D207, Faculty of Engineering, University of Ottawa, Ottawa, Canada, Oct 27, 2006.

[104] M. Thapar, B. Delagi, *Proceedings of the second annual ACM symposium on Parallel algorithms and architectures*, New York, NY: ACM Press, 1990, pp.155-160

[105] A. Agarwal, R. Simoni, J. Hennessy, M. Horowitz, *An Evaluation of Directory Schemes for Cache Coherence*, Standford University, CA: ACM Press, 1988, pp. 353 - 362

[106] [1] M. Bolic. CEG4131. Class Lecture, Topic: "Cache coherence." CBY D207, Faculty of Engineering, University of Ottawa, Ottawa, Canada, Oct 27, 2006.

[107] [2] H. El-Rewini and M. Abd-El-Barr, *Advanced Computer Architecture and Parallel Processing*. Hoboken, New Jersey: John Wiley and Sons, 2005.

[108] [3]Charles M. Kozierok,Cache write policy and the dirty bit [Online] Available,http://www.pcguide.com/ref/mbsys/cache/funcWrite-c.html

[109] [4] Badrish Chandramouli, and Sita Iyer, A performance study of snoopy and directory based cache-coherence protocols [Online], Available http://www.cs.duke.edu/~badrish/papers/cps221_paper.pdf

[110] [5] Lehrstuhl für Rechnerarchitektur, Shared memory with caching [Online] Available: http://www.ra.informatik.uni-mannheim.de/lsra/lectures/ws03_04/vl_ra2/script_pdf/dbcc.pdf

[111] [6] Wikipedia, Directory-based coherence protocols [Online] Available: http://en.wikipedia.org/wiki/Directory-based_coherence_protocols

[112] J. Kowalczyk, "Multiprocessor Systems" White Paper, Virtex-II Series, 2003

[113] H. El-Rewini, M. Abd-El-Barr, "ADVANCED COMPUTER ARCHITECTURE AND PARALLEL PROCESSING", ISBN: 0-471-46740-5, January 2005.

[114] B. Wilson, "INTRODUCTION TO PARALLEL PROGRAMMING USING MESSAGE-PASSING", Journal of Computing Sciences in Colleges, Volume 21, Issue 1, October 2005.

[115] Message Passing Interface (MPI), Available at: http://www.llnl.gov/computing/tutorials/mpi/#Environment_Management_Routines, accessed December, 10, 2006**.**

[116] M. Bane, R. Keller, M. Pettipher, I. Smith, "A Comparison of MPI and OpenMP Implementations of a Finite Element Analysis Code", Cray User Group, 2000.

[117] D. Culler, J. P. Singh, and A. Gupta, "Parallel Computer Architectures, A Hardware/Software Approach", ISBN 1-55860-343-3, August 1998.

[118] D. Sima, T. Fountain and P. Kascuk, “Advanced Computer Architectures – A Design Space Approach”, Addison Wesley, ISBN: 0201422913, 1997.

[119] The MPI: A Message Passing Interface (MPI) Forum.. In Proceedings of Supercomputing '93, pages 878--883, November 1993; available at: http://citeseer.ist.psu.edu/cache/papers/cs/1168/ftp:zSzzSzcse.ogi.eduzSzpubzSzogipvmzSzpaperszSzMPI-Super93.pdf/forum93mpi.pdf, accessed December 10, 2006.

[120] J. Kowalczyk, *Multiprocessor Systems*, Xilinx, 2003.

[121] D. Culler, J. P. Singh, *Parallel Computer Architectures: A Hardware/Software Approach*, Morgan Kaufman, 1999.

[122] Pittsburgh Super Computing Center (PSC), “Parallel Programming Techniques: MPI Basics,” December 2000, http://www.sc-2.psc.edu/workshop/jan01/Message_Passing/MPI_Basics.html#start_t3e .

[123] EPCC Training and Education Center, “Writing Message Passing Parallel Programs with MPI”, December 1995. http://www.epcc.ed.ac.uk/epic/mpi/notes/mpi-course-epic.book_4.html.

[124] D. Sima, T. Fountain and P. Kascuk, *Advanced Computer Architectures – A Design Space Approach*, Pearson, 1997.

[125] LLNL: Lawrence Livermore National Laboratory, “MPI: Message Passing Interface”, July 2006, http://www.llnl.gov/computing/tutorials/mpi/.

[126] E.D. Demaine, I Foster, C. Kesselman and M. Snir, “Generalized Communicators in the Message Passing Interface,” in *IEEE Transactions on Parallel and Distributed Systems*, June 2001, Volume 12, Issue 6, pp 610-61.

[127] K.T. Lim, “Mega-Molecular Dynamics on Highly Parallel Computers: Methods and Applications” PhD thesis, California Institute of Technology, Pasadena, California, USA, 1995.

[128] Abd-El-Barr and El-Rewini, *Advanced Computer Architecture and Parallel Processing*, John Wiley & Sons, 2005.

[129] Bolic, “Dynamic Interconnection Networks Buses” (CEG4131 Lecture Slides) http://www.site.uottawa.ca/~mbolic/ceg4131/index.shtml, last accessed November 5, 2006

[130] Bolic, “Message Passing Architectures and Routing” (CEG4131 Lecture Slides) http://www.site.uottawa.ca/~mbolic/ceg4131/index.shtml, last accessed November 5, 2006

[131] Pinkston and Duato, "Interconnection Networks". http://ceng.usc.edu/smart/slides/appendixE.html, last accessed November 5, 2006.

[132] Stallings, *Computer Networking with Internet Protocols and Technology*, Pearson, 2004.

[133] Wikipedia, "Packet switching". http://en.wikipedia.org/wiki/Packet_switching, last accessed November 5, 2006.

[134] Vaidya, Sivasubramaniam and Das, "Impact of Virtual Channes and Adaptive Routing on Application Performance", *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, pp. 223-237, February 2001.

[135] Al-Tawil, Abd-El-Barr and Ashraf, "A Survey and Comparison of Wormhole Routing Techniques in Mesh Networks", *IEEE Network*, pp. 38-45, March/April 1997.

a. W. Dally, B. Towles, Principles and Practices of Interconnection Networks, Morgan Kaufmann, 2004. pp. 160-165, 173, 189-192.

b. Timothy Mark Pinkston and Jose Duato. Computer Architecture: A Quantitative Approach, 4th Ed, Elsevier Publishers, 2006. Appendix E. [Online] Available: http://ceng.usc.edu/smart/slides/appendixE.html

c. Nicola Asuni. Network Switching Tutorial. Tecnick.com S.r.l. [Online]. Available: http://www.technick.net/public/code/cp_dpage.php?aiocp_dp=guide_networking_switching

d. S.Sartzetakis, C.Tziouvaras, and Leonidas Georgiadis. (2002, July) Adaptive Routing Algorithm for Lambda Switching Networks. *SPIE OPTICAL NETWORKS Magazine*. [Online]. Available: http://www.ics.forth.gr/netlab/publications/ONMpaper.pdf

e. A.L. Narasimha Reddy and R. Freitas. Fault Tolerance of Adaptive Routing Algorithms in Multicomputers. IBM Almaden Research Center, CA. [Online]. Available: http://ieeexplore.ieee.org/iel2/448/6240/00242750.pdf?tp=&arnumber=242750&isnumber=6240

f. M. Bolic, Dynamic Interconnection Networks Buses (CEG4131 Lecture Slides), University of Ottawa [Online]. Available:

http://www.site.uottawa.ca/~mbolic/ceg4131/index.shtml (last accessed November 8, 2006)

[136] M. Bolic. CEG4131. Class Lecture, Topic: “Flow Control and Deadlock.” CBY D207, Faculty of Engineering, University of Ottawa, Ottawa, Canada, Nov. 8, 2006.

[137] W. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. New York: Morgan Kaufmann, 2003.

[138] National Chung-Hsing University, “Buffered Flow Control”, 2005; available at http://www.cs.nchu.edu.tw/lab/syslab/2005Spring%20PDF/Chap13%20Buffered%20Flow%20Control.pdf (last accessed November 12, 2006).

[139] Università Modena Reggio Emilia, “Architetture MIMD a memoria distribuita”, 2005; available at http://weblab.ing.unimo.it/dispense_imp_03-04/d-architetture-mem-distribuita.pdf (last accessed November 12, 2006).

[140] T. C. Lethbridge and R. Laganière, *Object-Oriented Software Engineering*, London: McGraw-Hill, 2001.

[141] A. Silberschatz, G. Galvin, and P. Gagne, *Operating System Concepts* (7th ed.), New York: Wiley, 2005.

[142] H. El-Rewini and M. Abd-El-Barr, *Advanced Computer Architecture and Parallel Processing*. Hoboken, New Jersey: John Wiley and Sons, 2005.

[143] J. Rubio, P. Lopez, and D. Duato, "FC3D: Flow Control-Based Distributed Deadlock Detection Mechanism for True Fully Adaptive Routing in Wormhole Networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 8, pp. 765-779, Aug. 2003.

[144] Li-Shiuan Peh, William J. Dally, "Flit-Reservation Flow Control", *High-Performance Computer Architecture (HPCA) 2000*, pp. 73-84, Jan. 2000.

[145] G. Reeves, “What really happened on Mars,” 1997; available at http://research.microsoft.com/~mbj/Mars_Pathfinder/Authoritative_Account.html (last accessed November 12, 2006).

[146] Top500.org, “On TOP 10 Sites for June 2006”, November 14, 2006; available at http://www.top500.org/lists/2006/06 (last accessed November 14th, 2006)

[147] IBM Corporation, “IBM eServer Blue Gene Solution”, August, 2005; available at ftp://ftp.software.ibm.com/common/ssi/rep_sp/n/DCS00764USEN/DCS00764USEN.PDF (last accessed November 14th, 2006)

[148] HPC Wire, “On &A With IBM's Blue Gene/L Chief Architect”, June 2, 2006; available at http://www.hpcwire.com/hpc/677493.html (last accessed November 14th, 2006)

[149] Kumar, S, " A Network on Chip Architecture and Design Methodology," VLSI, 2002. Proceedings. IEEE Computer Society Annual Symposium on, pp. 105 - 112 , 25-26 April 2002 .

[150] L. Benini and G. De Micheli, “Networks on Chips: A New SoC Paradigm,” Computer, vol. 35, no. 1, Jan. 2002, pp. 70-78.

[151] S. Tota and M. R. Casu Sergio Tota and Mario R. Casu, “Networks-on-Chip,” presentation. www.tlc.polito.it/~nordio/seminars/2006_05_05_Casu.ppt

[152] D. Harris CMOS VLSI Design, International ed., Ed. Pearson Education Inc., Boston, USA: Addison Wesley, 2005, p.299.

[153] Kumar, S, " A Network on Chip Architecture and Design Methodology," VLSI, 2002. Proceedings. IEEE Computer Society Annual Symposium on, pp. 105 - 112 , 25-26 April 2002 .

[154] P. Pratim and A. Ivanov, " Performance Evaluation and Design Trade-Offs for Network-on-Chip Interconnect Architectures," IEEE Transaction on Computers, pp. 1025-1040 , August 2005 .

[155] Abd-El-Barr and El-Rewini, *Advanced Computer Architecture and Parallel Processing*. John Wiley & Sons, 2005, pp. 235-256.

[156] El-Rewini, “Scheduling and Dependence (CEG 4131 Lecture Slides),” November 20, 2006, http://www.site.uottawa.ca/~mbolic/ceg4131/index.shtml

[157] Mario J. Gonzalez, Jr., *Deterministic Processor Scheduling*, Division of Mathematics, Computer Sciences and System Design, The University of Texas at San Antonio, San Antonio, Texas 78285

[158] Olga Rusanova and Alexandr Korochkin, *Scheduling Problems for Parallel and Distributed Systems*, National Technical University of Ukraine – “Kiev Polytechnical Institute", Prospect Peremogy 37, 252056,Kiev, Ukraine

[159] Hammerstrom, “Computer Architecture - Vector Processing. Week 8, Monday”.http://web.cecs.pdx.edu/~strom/ece587_web/lectures.htm, last accessed November 18, 2006.

[160] Wikipedia, “Vector processor”.http://en.wikipedia.org/wiki/Vector_processor, last accessed November 18, 2006.

[161] Hammerstrom, "Computer Architecture - Vector Processing. Week 8, Wednesday".http://web.cecs.pdx.edu/~strom/ece587_web/lectures.htm, last accessed November 18, 2006.

[162] IBM Whitepaper, "Cell Broadband Engine Architecture".http://www-306.ibm.com/chips/techlib/techlib.nsf/products/Cell_Broadband_Engine, last accessed November 18, 2006.

[163] J. Hennessy, D. Patterson, *Computer Architecture, A Quantitative Approach*, Morgan Kaufmann, 2002, Appendix G.

[164] Kozyrakis, C.E, Patterson, D.A. "Scalable, vector processors for embedded systems", Micro, IEEE, vol. 23, Issue 6, pp. 36-45, Nov.-Dec. 2003.

[165] Lee, C.G., Stoodley, M.G., "Simple vector microprocessors for multimedia applications", IEEE ACM Journal, pp. 25-36, 30 Nov.-2 Dec. 1998.

[166] Wikipedia, "Vector Processors"; available at http://en.wikipedia.org/wiki/Vector_processor (last accessed December 12th, 2006).

[167] Roger Espasa, Mateo Valero, and James E. Smith, *"Vector Architectures: Past, Present, and Future"*. ICS '98, Proceedings of the 1998 International Conference on Supercomputing, July 13-17, 1998, Melbourne, Australia. ACM, 1998.

[168] Dan Hammerston. Advanced Computer Architecture 1.Class Lecture, Topic: "Vector Processors". Portland State University, available at http://www.site.uottawa.ca/~mbolic/ceg4131/week8.pdf (last accessed December 12th, 2006).

[169] J. Hennessy, D. Patterson, *Computer Architecture, A Quantitative Approach*, Morgan Kaufmann, 2002, Appendix G: Vector Processors

[170] Christoforos E. Kozyrakis, David A. Patterson, "*Scalable Vector Processors for Embedded Systems"*. Nov-Dec 2003, Micro IEEE, pp. 36-45.

[171] UC Berkley. The Berkley IRAM Project. *University of California, Berkley*, available at http://iram.cs.berkeley.edu/ (last accessed November 20th 2006).

[172] IBM. Cell Broadband Engine Architecture, available at http://www-306.ibm.com/chips/techlib/techlib.nsf/techdocs/1AEEE1270EA2776387257060006E61BA/$file/CBEA_v1.01_3Oct2006.pdf (last accessed December 12th, 2006).

[173] M. Smotherman, “Understanding EPIC Architectures and Implementations”; Department of Computer Science, Clemson University; Clemson, SC, 29634 USA; available at http://www.cs.clemson.edu/~mark/464/acmse_epic.pdf (last accessed 25 November, 2006).

[174] P. Navaux, “Arquiteturas Superescalares” , (In Portuguese); Course Slides, Departamento de Informatica, Universidade Federal do Rio Grande do Sul; Porto Alegre, RS, Brasil; available at http://www.inf.ufrgs.br/procpar/disc/cmp134/prog/ippd.html (last accessed 25 November, 2006).

[175] M. Bolic, CEG4131. Class Lecture, “Superscalar and VLIW Architectures”; Faculty of Engineering, University of Ottawa, Ottawa, ON, Canada; 25 November, 2006.

[176] P. Eles, “VLIW Processors”, Lecture Slides pp. 1-4, Department of Computer and Information Science, Linköpings Universitet; available at http://www.ida.liu.se/~TDTS51/lectures/lectures9-10.pdf (last accessed 25 November, 2006).

[177] Wikipedia, “VLIW Processor”, “Superscalar”, “EPIC”, “CISC”, “RISC”; available at http://en.wikipedia.org/wiki/VLIW, http://en.wikipedia.org/wiki/Supersacalar , http://en.wikipedia.org/wiki/EPIC , http://en.wikipedia.org/wiki/CISC and http://en.wikipedia.org/wiki/RISC (last accessed 25 November, 2006).

[178] B. Matthew, “VLIW Processors and Trace Scheduling”; available at http://www.siliconintelligence.com/people/binu/pubs/vliw/old/vliw.pdf (last accessed 25 November, 2006).

[179] Philips Semiconductors, “An Introduction to Very-Long Instruction Word (VLIW) Computer Architecture”; available at http://www.nxp.com/acrobat_download/other/vliw-wp.pdf (last accessed 25 November, 2006).

[180] M. Sami, “Come aumentare le prestazioni: cenni alle architetture avanzate” (In Italian) Lecture Slides, Dipartimento di Elettronica e Informazione, Politecnico di Milano ; Milano, Italia; available at http://www.elet.polimi.it/upload/sami/organizzazione_dei_calcolatori/organizzazione_arch_avanzate.pdf (last accessed 24 November, 2006).

[181] Berkeley Design Technology, “Motorola/IBM PowerPC 604/604e”; available at http://www.bdti.com/products/reports_gpp604.htm (last accessed 25 November, 2006)

[182] T. Thompson and B. Ryan, "PowerPC 620 Soars"; November 2004; available at http://www.byte.com/art/9411/sec8/art5.htm (last accessed 25 November, 2006)

[183] Wikipedia, "Pentium"; available at http://en.wikipedia.org/wiki/Pentium (last accessed 25 November, 2006)

[184] J. Fisher, P. Faraboschi, C. Young; "Embedded Computing: A VLIW Approach to Architecture, Compilers, and Tools", Morgan Kaufmann, 2005

[185] J. Hennessy, D. Patterson; "Computer Architecture – A Quantitative Approach", Morgan Kaufmann, 3rd Edition 2002

[186] W. Stallings, "Computer Organization and Architecture", Prentice Hall, 6th Edition, 2002

[187] J. A. Fisher, "Very long instruction word architectures and the ELI-512," inProc. 10th Annu. Int. Symp. Comput. Architecture, ACM-SIGARCH and the IEEE Computer Society, June 1983, pp. 140-150. available at http://delivery.acm.org/10.1145/290000/285985/p263-fisher.pdf?key1=285985&key2=1753954611&coll=&dl=ACM&CFID=15151515&CFTOKEN=6184618 (last accessed 25 November, 2006)

[188] B. Rau, "Dynamically Scheduled Processors", Hewlett-Packard Laboratories; available at www.hpl.hp.com/research/itc/car/papers/../papers/dynamically.pdf (last accessed 25 November, 2006)

[189] M. Bolic. CEG4131. Class Lecture, Topic: "Flow Control and Deadlock." CBY D207, Faculty of Engineering, University of Ottawa, Ottawa, Canada, Nov. 22, 2006.

[190] Mark Smotherman, "Understanding EPIC Architectures and Implementations",; available at http://www.cs.clemson.edu/~mark/464/acmse_epic.pdf (last accessed November 24, 2006).

[191] Nicholas Carter, "Data Hazards and Forwarding", October 16, 2006; available at http://courses.ece.uiuc.edu/ece411/**lecture**s/**lecture**12.pdf (last accessed November 24, 2006).

[192] Philips Semiconductors, "An Introduction To Very-Long Instruction Word (VLIW) Computer Architecture",; available at http://whitepapers.silicon.com/0,39024759,60012157p-39000381q,00.htm(last accessed November 24, 2006).

[193] Petru Eles, “Very Large Intruction Word (VLIW) Processors”,; available at http://www.ida.liu.se/~TDTS51/lectures/lectures9-10.pdf(last accessed November 24, 2006).

[194] Brian L. Evans, “Introfuction to the TMS320C6x VLIW DSP”,; available at http://www.ece.utexas.edu/~bevans/hp-dsp-seminar/02_IntroC6x.ppt(last accessed November 25, 2006).

[195] Y. Qian, S. Carr and P. Sweany. "Optimizing Loop Performance for Clustered VLIW Architectures", In *Proceedings of the Eleventh IEEE International Conference on Parallel Architectures and Compiler Techniques (PACT-2002)*, Charlottesville, Virginia, September 22-25, 2002.

[196] S. Jang, S. Carr, P. Sweany, and D. Kuras, ``A Code Generation Framework for VLIW Architectures with Partitioned Register Files". In *Proceedings of the Third International Conference on Massively Parallel Computing Systems*, Colorado Springs, Colorado, April 1998.

[197] Miodrag Bolic (2006, November 25th) Computer Architecture III. “Superscalar Processors” slides.

[198] W. Stallings, *Computer Organization and Architecture*, 6th Edition, Prentice Hall, 2003, pp 505- 536

[199] J. P. Shen, M. H. Lipasti, *Modern Processor Design: Fundamentals of Superscalar Processors*, Beta Edition, McGraw Hill, 2003, pp 107-132

[200] H. El-Rewini and M. Abd-El-Barr, “Advanced Computer Architecture and Parallel Processing”, A John Wiley and Sons, 2005.

[201] Nicholas P. Carter, “Computer Architecture”, McGraw-Hill Professional, 2002.

[202] M. Bolic. CEG4131. Class Lecture Topic: “Superscalar Processor” and “Superscalar Processor – Review”, CBY D207, Faculty of Engineering, University of Ottawa, Ottawa, Canada, Nov. 29, 2006.

[203] N.P.Lu, C-P.Chung, “Parallelism exploitation in Superscalar multiprocessing” IEE Pro-Comput. Digit Tech., Vol. 145, No.4, July 1998. Available at: http://ieeexplore.ieee.org/iel4/2192/15240/00705689.pdf?tp=&arnumber=705689&isnumber=15240, (last accessed 29 November, 2006).

[204] James E.Smith, “The Microarchitecture of superscalar processors”, Proceeding of the IEEE, vol 83., no. 12, Dec 1995. Available at: http://ieeexplore.ieee.org/iel1/5/10194/00476078.pdf?tp=&arnumber=476078&isnumber=10194, (last accessed 29 November, 2006).