Shared Memory and Shared Memory Consistency

Josip Popovic
Graduate Studies and Research, Systems and Computer Eng. Carleton University, OCIECE program

Abstract

Multiprocessors (MP) use is growing in standard desktop/server arena as well as in the embedded application space. Individual processors need to share data to be processed due to parallel programming requirements. Availability of data needs to be signaled between individual software threads. Sharing of processing data and thread synchronization is supported by a shared memory system. Shared memory systems are often accompanied with local caches. The interaction between local caches and shared memories is a complicated one. Preferably inner workings of a shared memory system should be hidden from a programmer, in an ideal case the programmer would be presented with a sequential memory consistency model.

1. Introduction

Parallel MP memory subsystems are built so local and shared data is available to all processors. Data processing intensive program segments can be executed in parallel. Application data can be split in pages and re-allocated among processors using existing uniprocessor hardware (Memory Management Unit (MMU), Translation Look-aside Buffer (TLB), caches etc) or kernel features. These MP enablers prompted major processor companies to invest in designing and building MP systems. Designing shared memory subsystems in an efficient manner proved to be the key challenge. What is expected from these subsystems is to be consistent, fast and scalable.

Scalable MP systems need to hide shared memory system data latencies. A number of relevant techniques have been proposed and accepted in industry:

- Coherent Caches (CC) – data latency hidden by caching data close to a processor
- Relaxed Memory Consistency – reordering of memory accesses and buffering/pipelining some of memory accesses
- Data Pre-fetching – long latency memory accesses issued long before needed
- Multithreading – program threads context switched if a long memory access takes place

In most of contemporary MP systems more than one of the above techniques is used. For example cache coherent multithreaded system with context (thread) switching in the case of read cache miss.

It this paper we focus on how parallel programs use shared data and why the memory consistency is required, data synchronization in different shared memory consistency models and their implication on parallel programming. Software and hardware (or a combination of the two) based memory consistency models are presented. What memory consistency model is the most attractive depends on the multiprocessing system architecture, targeted performance and costs. Therefore MPSoC (Multiprocessor System on a Chip) or distributed MP system made from individual networked computers provide memory consistency differently.

2. Shared Memory and Relaxed Consistency

Parallel processor architectures use shared memory as a way of exchanging data among individual processors to:

- Avoid redundant copies conserving memory space by directing memory accesses to a single data instance
- Enable Inter-Processor Communication (IPC), for example thread/process synchronization events

Most of contemporary multiprocessor systems provide complex cache coherence protocols to ensure all processors' caches are up to date with the most recent data. However having a cache coherent system does not guarantee correct MP system operation (due to memory access bypass etc). Therefore in a parallel programming a memory consistency is required (cache coherence is less restrictive than memory consistency – if shared memory is consistent, its cache is coherent). While cache coherence is a HW based protocol, memory

consistency protocols are implemented in software (SW) or combination of software and hardware (HW).

Shared memory could be organized as a centralized memory or as a distributed memory. Processors and memories could be connected with a shared bus (not a scalable solution) or with a NoC.

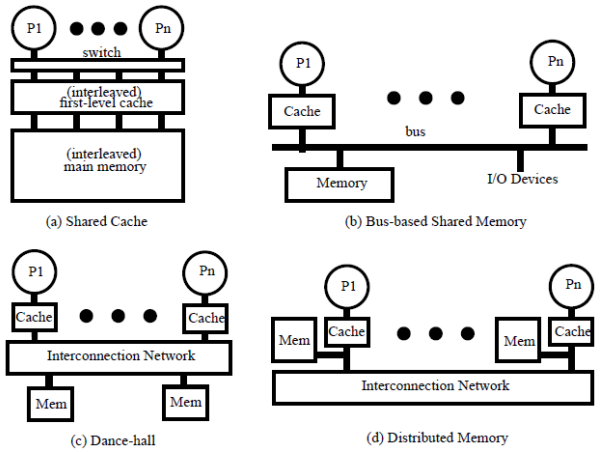

Figure 1. Fundamental multiprocessor shared memory topologies, [1]

Table 1. Cache and memory system topologies

Architecture	Pro	Con
Shared cache	Cache updates Low latency	Cache bandwidth – bus conflicts
Private cache	Cache conflicts	Scalability, coherence
Dane Hall	Not used much	
DSM	Used now and in future	

Distributed Shared Memory (DSM) can be added to a system as a symmetrical structure, meaning all processors are uniformly spaced away from it or with individual memories attached to their respective processors. The last configuration, shown on Figure 1 above (d), is not symmetrical since processor access time to its local attached memory is different versus a memory attached to a different processor.

One example of contemporary architecture with non-symmetrical distributed shared memory architecture is Intel NUMA (Non-Uniformed Memory Access). In this architecture memory is distributed as to minimize penalties when more than one processor requires access to the memory. At the same time all memory instances are shared by all processors. For a short list of DSM pros and cons see Table 2 below.

Table 2. Distributed Shared Memory pros/cons

Distributed Memory Pros	Distributed Memory Cons
Better memory utilization if a processor(s) idle (load balancing)	Different memory latencies (local shared versus distant shared)
Better memory bandwidth due to multiport memory accesses	Memory consistency complicated
Speed of access to local memory	Cache coherence complicated

Correctly utilizing multiprocessors with shared memory and writing efficient programs for them requires a memory consistency model [1].

From Merriam Webster on line dictionary: "*consistency - agreement or harmony of parts or features to one another or a whole*". In the computer systems memory consistency has a more specific meaning. In this paper the following definition is used [1]:

"A *memory consistency model* for a shared address space specifies constraints on the order in which memory operations must appear to be performed (i.e. to become visible to the processors) with respect to one another. This includes operations to the same locations or to different locations, and by the same process or different processes, so memory consistency subsumes coherence."

In other words memory consistency models define how shared memory accesses from one processor are presented to other processors in a MP system. Therefore if one processor performs memory writes we want all processors reading these same memory locations to read data in a sequence required by the parallel program being run. If this condition is met, the program execution is guaranteed to be the same regardless of the system it is run on.

In either of the two examples that follow [1] a wrong assumption about write operations on the processor 1 is made.

```
        P1                              P2
   /* Assume initial value of A and flag is 0 */
A = 1;                          while (flag == 0);
Flag = 1;                       print A;
```

Figure 2. Write Operation [1]

Although processors are cache coherent, the actual sequence of writes to different locations of shared memory is not guaranteed. A simple reason could be that the MMU may reorder writes to optimize memory bandwidth (for example bank conflict avoidance, see the Figure 3, below).

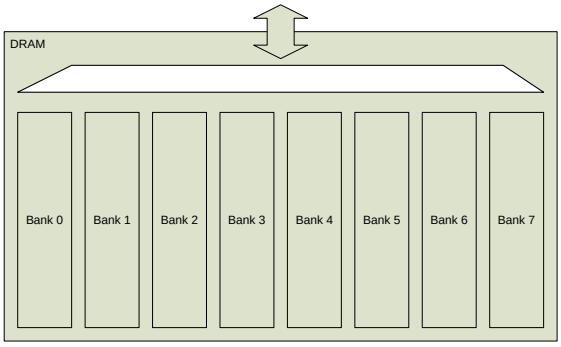

Figure 3. DRAM are organized in banks

Another unknown variable in the memory consistency equation is the use of barriers. If barriers are done purely in SW by using shared variables we are in the same situation as the example with flags. If the barrier is supported by HW we still do not know if writes are completed and non-atomic to other processors.

```
        P1                              P2
   /* Assume initial value of A and flag is 0 */

     A = 1;                             ……
  BARRIER(b1)
                                  print A;
```

Figure 4. Use of Barriers [1]

The problem of maintaining memory consistency in multiprocessor systems becomes apparent. Considered the following Sequential Consistency (SC) memory model [2]:

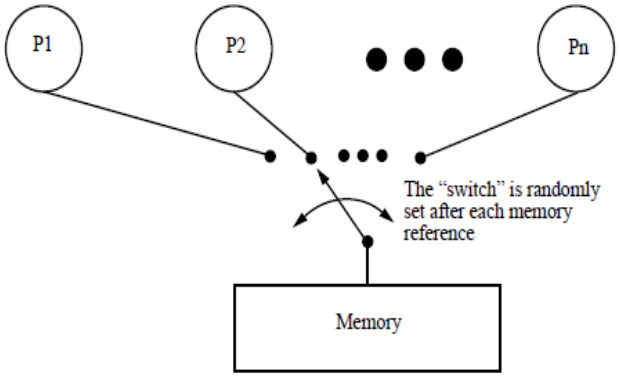

Figure 5. Programmers view of Sequential Consistency [1]

In this model the result of execution of program instructions is the same as if individual processors executed their threads in the program sequential order to their full completion. As a result all operations appear to be executed in an atomic order. Therefore the memory system is consistent.

In order to guarantee SC a multiprocessing system needs to meet the following three requirements [13]: (a) memory operations from the same processor can not be re-ordered, (b) processor issuing a write operation needs to wait till its completion and (c) memory reads can not be completed till all copies of the relevant variable are updated throughout the complete memory hierarchy.

As it will be shown in the following example SC significantly reduces processor instruction execution optimization solution space. These optimizations are regularly used on uniprocessors and compilers.

In essence processor write instructions are posted operations (processor does not need an ack, it can be assumed that they took place). The write buffers are used to prevent processor stalls if a memory address is unavailable. However if reads that follow the posted writes (buffered writes) are not for the same memory address, reads can bypass the buffered writes, effectively improving the processor performance.

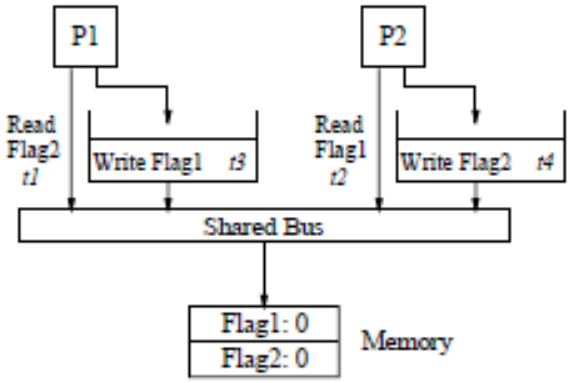

Figure 6. Write buffer and buffer bypass [1]

However due to the read bypass the SC model is violated as shown in the following Dekker's algorithm [1]:

Assume Flag1=Flag2=0 at init time:

P1:	P2:
Flag1 = 1;	Flag2 = 1;
if (Flag2 == 0)	if (Flag1 == 0)
critical section	*critical section*

Figure 7. Dekker's Algorithm [1]

In the above example flags are used by respective processors to determine if the other processor is in the critical section, therefore both flags should never be 0. However once both processors come to the same code due to the reads bypassing writes (for example P1 writes Flag1 while reading Flag2, 2 different addresses) they will both see Flag1-Flag2 as it is in the shared memory. As a consequence both processors end up being in the critical code section.

In Figure 8 it is shown how two consecutive write operations to different memory modules (a) and non blocking reads (b) violate SC. Non-blocking reads allow a read to be executed followed by another read without waiting for the first read data to come. This case also applies to a non-blocking read to write operation.

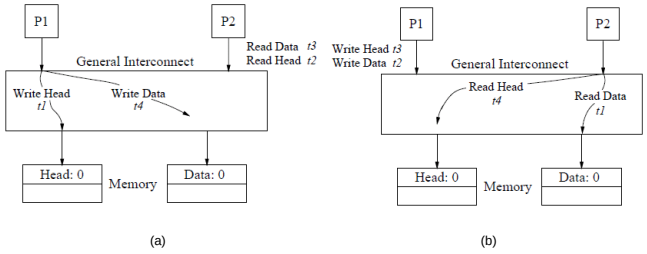

Figure 8. (a) Overlapping write operations, (b) Non-blocking reads [2]

The following sequence of instruction could break SC model due to the overlapping writes or non-blocking reads:

P1:	P2:
Data = 2000;	While (Head == 0) {;}
Head = 1	Data1 = Data;

Figure 9. Invalidation of SC Model [1]

In [1] authors show other examples where processor optimization hardware can violate SC including:

- use of caches can violate SC even if a write acknowledge is used,
- SC affected by the compiler instruction optimizations etc

Considering the above SC imposed hardware optimization limitations we can conclude that the SC model is too restrictive and other memory consistency models need to be considered. A number of relaxed memory consistency models have been documented in research communities and/or implemented in commercial products. Relaxed memory consistency can be implemented in SW and HW or a combination.

Suggested and researched relaxed consistency models could address each of the above SC limitations individually or all of them together [1] [2] [3]. If all program order requirements are relaxed weak consistency ordering (WC) and release consistency (RC) models have been defined and explained in the text that follows.

3. Distributer Shared Memory

Distributed Shared Memory (DSM) is a concept where a cluster of memory instances are presented to SW running on a multiprocessor system as a single shared memory. There are two distinctive ways of achieving memory consistency, HW based or SW based. Off-course it is possible to combine HW and SW methods.

Here as an example of HW based DSM technology we use Intel CC-NUMA (Cache Coherent Non-Uniform Memory Access) architecture [14]. Intel CC-NUMA is a product of Intel processor evolution. It is a scalable distributed shared memory as documented in [19].

Figure 10 below shows Intel's path to CC-NUMA architecture supported by QuickPath Interconnect (QPI) used in modern systems.

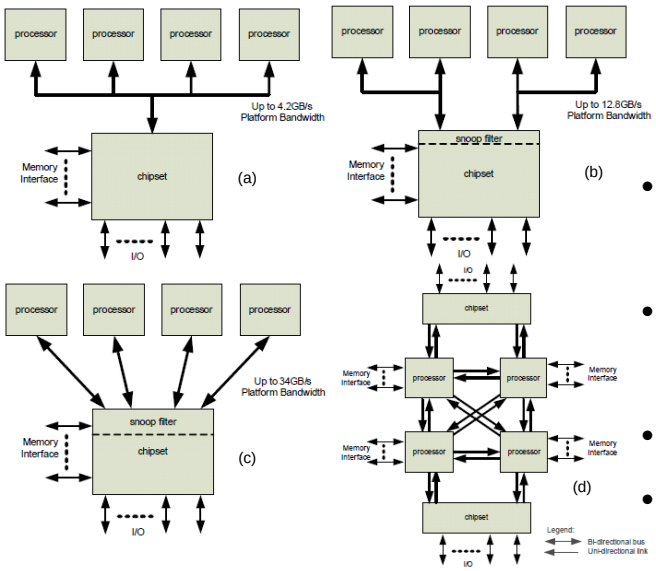

Figure 10. Intel Processor Interconnect Evolution [14], (a)-Shared Front-side Bus, 2004, (b)-Dual Independent Buses, circa 2005, (c)-Dedicated High-speed Interconnects, 2007 and (d)-QPI

QPI together with use of multi-channel DSM memory topology provides CC-NUMA with high-throughput memory transfer features. QPI links are uni-directional, differential signaling serial links and packet based. Packets are CRC protected.

QPI is based on OSI modeling and supports 5 layers see Figure 11 below.

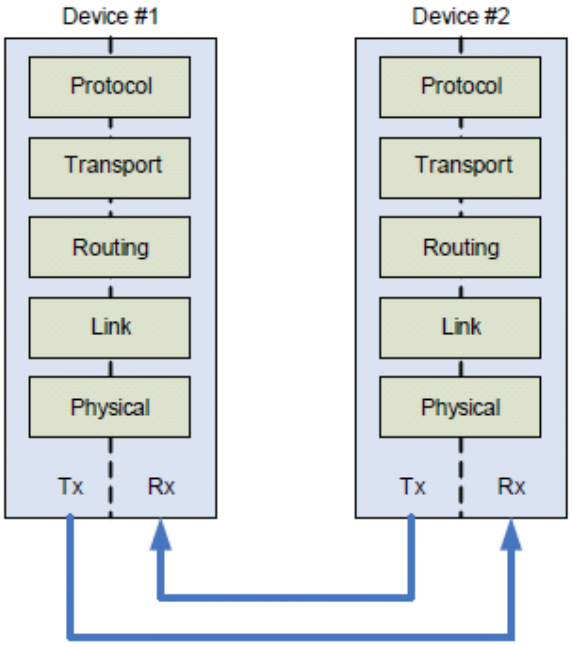

Figure 11. QPI OSI Model Layers [14]

- The Physical layer: unit of transfer at the Physical layer is 20 bits, which is called a Phit (for Physical unit).
- Link layer: reliable transmission and flow control. The unit of transfer is an 80-bit Flit (for Flow control unit).
- Routing layer: directing packets through the fabric.
- Transport layer: architecturally defined layer advanced routing capability for reliable end-to-end transmission.
- Protocol layer: high-level set of rules for exchanging packets of data between devices. A packet is comprised of an Integral number of Flits.

Intel CC-NUMA uses modified MESI or MESIF (Modified, Exclusive, Shared, Invalid and Forward) cache coherence protocol. Cache related message passing has been designed to be limited to only 3 hoops versus 4 in previous systems.

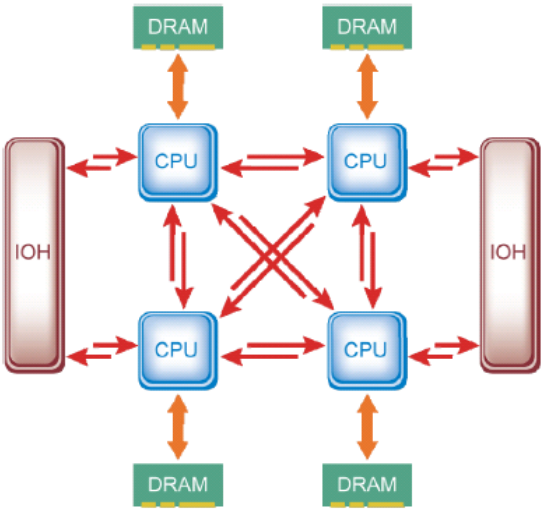

Figure 12. Intel QPI based NUMA architecture [14]

It is important to note that the CC-NUMA architecture is not memory consistent therefore a memory consistency protocol is still required.

4. Page-based shared virtual memory

Contrary to Intel QPI CC-NUMA architecture where memory consistency is provided by the hardware based cache protocols and synchronization instructions added to a parallel program, the software implementation of DSM consistency requires adding a software layer between the operating system and the application.

In SW based memory consistency model processor memory management unit, concept of paging and virtual memory are utilized. All of these concepts are in use in processors therefore it is advantageous to reuse them. Required memory consistency functionality can be just added to the existing page fault handler HW/SW support.

This model is called a *page-based shared virtual memory* or SVM [1] [15] [16]. From Figure 13, in a case of page fault, a remote page access right is given to the processor that incurred the page fault.

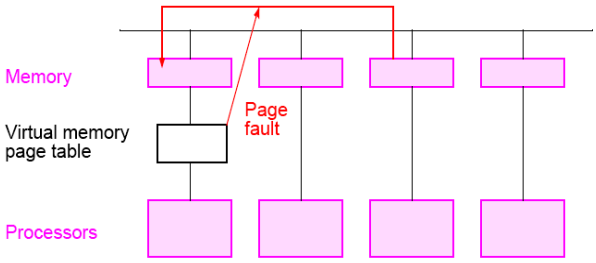

Figure 13. Page-Based Distributed Shared Virtual Memory [15]

Advantage of this method is low HW support required since it is built on top of the existing infrastructure. However due to the page size, probability of false data sharing is higher (different processor may have the same virtual address although their physical addresses are different). Figure 14 below displays interaction between processors and their local memories with SVM SW library (running on the main processor) and the main memory (could be hard disk drive). P0 obtains a page from the main memory, virtual address P0.V1. Processor P1 needs a page from the main memory with a virtual address P1.V1. If P0.V1 is the same as P1.V1 SVM SW layer will copy read-only P0.V1 into P1 memory. It is possible that although these virtual addresses from P0 and P1 are the same that underlying physical addresses are different. In the next cycle P0 tries to write to read-only page which causes a page fault so P1 has its copy invalidated. In the last cycle P0 fetches page from P0.V1, this time with read-write access. P1 when it tries to read its copy will get a page fault (the page was invalidated) and will get a new copy from P0.

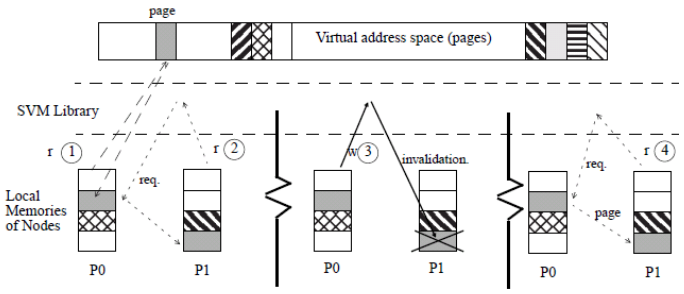

Figure 14. Page-Based Distributed Shared Virtual Memory [1]

This higher probability of false page sharing is expressed in a more frequent inter-processor communication overhead. Therefore SC needs to be replaced with a release relaxed memory consistency such as release consistency. In RC model the page invalidation is performed when all processors are at a synchronization point, see Figure 15.

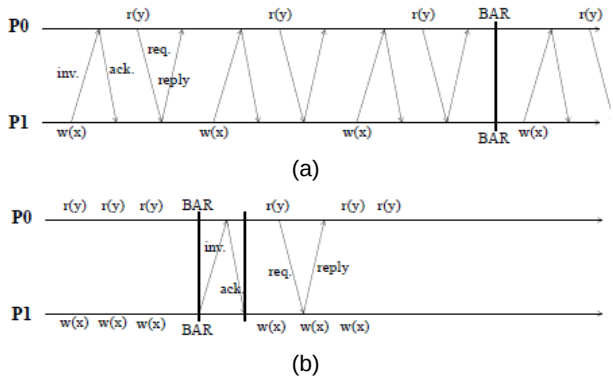

Figure 15. SC (a) and RC (b) shared virtual memory consistency [1]

From Figure 15 P0 reads variable y while P1 writes variable x that are in the same virtual page but not the same variables – therefore this represents a false sharing case. In the case (a) we have SC in the case (b) RC.

What we would prefer would be RC model where inter-processor communication takes place only during processor synchronization (a SW barrier is hit).

In the case when memory consistency is HW supported invalidation messages are sent whenever writes take place therefore RC shown in Figure 15 (b) is not really real. Actual invalidation messages are created all the time not only during synchronization stage. Our intention is minimizing invalidation messages. The following two approaches are suggested [1]: eager release consistency (ERC) and lazy release consistency (LRC) SVM models.

In the eager case invalidation messages are created only during the page release moments (Figure 16 (a)). P0 after acquiring a page writes to it sending an invalidation message during the page release to all processors. Therefore remaining processor may go ahead and read the invalidated page due to the false sharing (see write by P0 to variable x and reads by P1 and P2 of variable Y).

In addition this method is conservative since P0 sends invalidations to processors without knowing when they will acquire the relevant page.

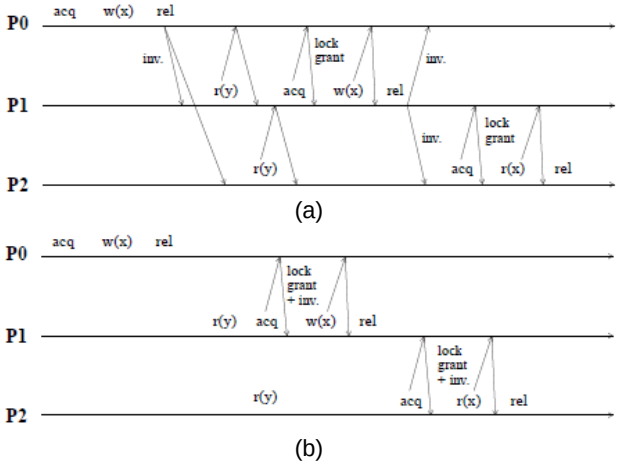

Figure 16. Eager (a) and Lazy (b) release relaxed SVM consistencies [1]

To avoid the above ERC issues LRC was proposed as a more promising release consistency model [5] [6] (see Figure 16 (b)). LRC sends invalidation message not at the page release (a release that follows write(s)) but rather on the next acquire by a relevant processor. If a processor acquires a page it gets all grant/invalidation messages since its last acquire.

It was shown that LRC gives better results that ERC. However designing in LRC is more difficult.

Claims have been made that NCC-NUMA (Non Cache Coherent NUMA) systems that use SW based SVM consistency can approach performance of fully HW coherent multiprocessors [16]. Some examples of SW based SVM consistency tools are CACHMERe, JIAJIA, Treadmarks [5] etc.

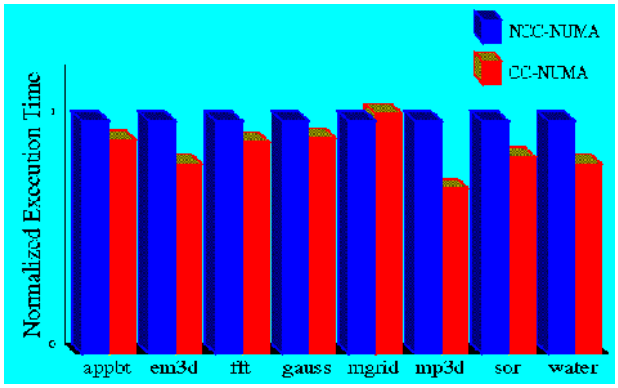

Figure 17. NCC-NUMA with SW DSM consistency versus CC-NUMA [16] – Cashmere, Rochester University

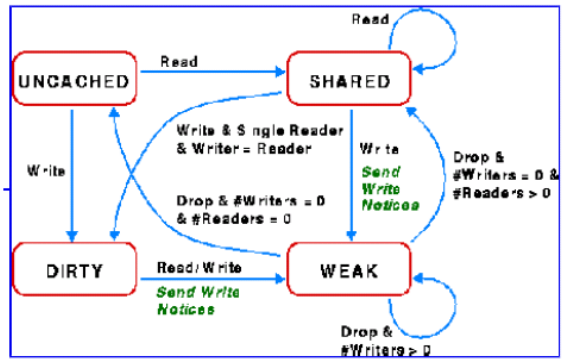

Figure 18. Lazy Cache Coherence Protocol [9]

4.1 TreadMarks

TreadMarks is a DSM SW platform created at Rice University. It supports running parallel programs on a network of computers. It provides global shared address space across different computers (versus message passing architecture) allowing programmers to focus on application rather than on message passing. Data exchange between computers is based on UDP/IP. It uses lazy release memory consistency model as well use of barrier and lock synchronization, critical sections etc. Some of available functions:

```
void Tmk_barrier(unsignedid);
void Tmk_lock_acquire(unsignedid);
void Tmk_lock_release(unsignedid);
char *Tmk_malloc(unsignedsize);
void Tmk_free(char*ptr);
```

4.2 Lazy Release Consistency for HW Coherent Processors

In [9] authors suggest using a cache coherence protocol based on the lazy release consistency model. However this protocol is not analyzed here, this method is just mentioned as an example how eager/lazy release consistency can be implemented in SW or HW.

5. Relaxed memory consistency models for parallel software

Shared address space memory consistency models specify the order of memory operations. Less reordering is allowed more understandable programming models is but less performance optimizations can take place. We need a memory model that gives us a good balance between programming complexity and performance.

The sequential consistency (SC) model is the simplest for a programmer to understand. Programming order within a process is maintained while individual processes are interleaved. Neither compiler nor HW can do any memory re-ordering. However this model is not efficient since it prevents optimization techniques from being used.

Therefore changing the sequential memory model to a somehow relaxed one that would allow the compiler and HW to do some memory access re-ordering is needed. The compiler can reorder some accesses before they are used by HW. The HW is also allowed to do some reordering to hide memory access latencies. This model is referred as to the relaxed memory ordering (RC).

In [1] there are shown three parts forming a complete solution for RC

- The system specification: what program orders related to memory accesses are guaranteed to be preserved and if not preserved what are the system rules provided to the programmer to enforce ordering. There are 3 sets of relaxation models: (a) in the first one only a read is allowed to bypass previous incomplete writes, (b) in the second we allow the above plus writes can pass previous

writes, (c) in the third we allow all of the above plus reads or writes to bypass previous reads

- The programmers interface: to make parallel programming easier we need to define a simple set of directions that programmer needs to follow to enforce ordering. For example if a variable *flag* is used to synchronize threads programmer would use pragma style compiler instructions.
- The translation interface

5.1 Write to Read Re-Ordering Relaxation

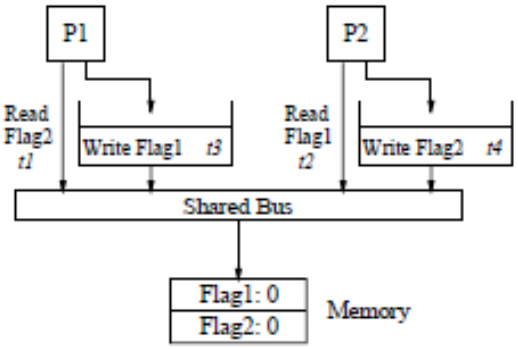

Figure 19. Write to Read Re-Ordering Relaxation [1]

In the case we had a write miss a potentially long memory latency to fetch relevant cache line from the system memory would noticeable slow down the processor. Therefore we allow reads to go ahead of this write. Write is stored in a write FIFO while reads with cache hit bypass writes (read misses also can go ahead but there could be only one read bypass instruction).
The following code executes correctly on a system supporting Write to Read Re-Ordering:

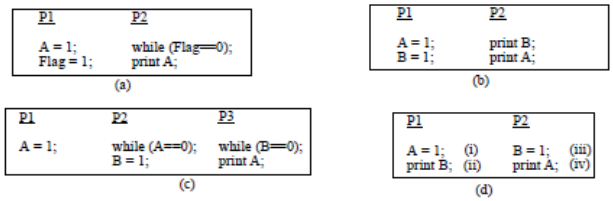

Figure 20. Write to Read Re-ordering [1]

Examples (a), (b) and (c) are understandable and it is clear what the programmer needs to do. In the example (d) variable A and B cannot be both 0 and that is true if P1 or P2 complete both instructions (in any order) before the other processor (SC provided).

5.2 Write-to-Read and Write-to-Write Program Orders Relaxation

What is gained with this relaxation is that multiple writes with cache misses can be merged in the write buffer (they are likely to be in the same cache line). In return this will noticeable minimize processor stalls due to write cache miss latencies. In cases when a critical write-to-write re-ordering is not allowed a programmer needs to use special instructions.

5.3 All Program order Relaxation

In this case reads can complete out of order, be bypassed by latter writes or there could be multiple reads outstanding. Re-ordering can be done by the compiler or processor HW.

5.3.1 Weak Ordering consistency (WC)

In WC parallel programs use synchronization instructions when necessary, otherwise full compiler/HW optimization is allowed. Flags and lock/unlock operations are used to maintain this synchronization. Code example:

```
      P1                          P2
TOP: while(flag2==0);     TOP: while(flag1==0);
     A = 1;                     x = A;
     u = B;                     y = D;
     v = C;                     B = 3;
     D = B * C;                 C = D / B;
     flag2 = 0;                 flag1 = 0;
     flag1 = 1;                 flag2 = 1;
     goto TOP;                  goto TOP;
```

Figure 21. WC code example [1]

Notice use of flags in *while* loops and clearing of the "opposite" flags. Therefore we can summarize that WC relaxes all compiler and HW memory access ordering except when synchronization operations are used.

In WC all memory operations are defined as data or synchronization. Data operations can be executed with processor HW optimizations (reordering). Synchronization operations need to be HW supported so they provide a safety net to enforce the program order.

One example of WC synchronization safety net is a fence operation [18]. It could be implemented with a memory operation counter per processor. The counter is incremented when processor issues an operation and decremented when an issued operation is completed.

None of the processors can issue a synchronization operation till all of the counters are at 0. Consequently no further operations are issued till the synchronization operation is completed.

The above or similar HW mechanisms are used where HW and SW are in a contractual agreement [3]. In the contract SW is obligated to follow a set of constraints while HW provides an illusion of sequential consistency.

The original definition for WC was given in [4] where storage accesses are weakly ordered if (1) global synchronizing variable accesses are strongly ordered, (2) no access to a synchronizing variable is issued by a processor before all previous global data accesses have been globally performed, and if (3) no access to global data is issued by a processor before a previous access to a synchronizing variable has been globally performed.

A number of papers proposed further WC relaxation techniques targeting even better MP performance [3].

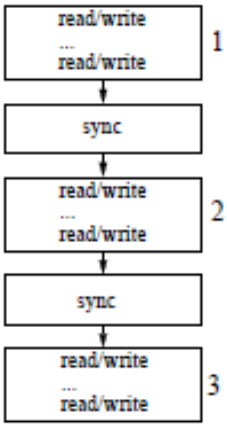

Figure 22. Weak Ordering Consistency [1]

There are 3 basic fence operations: lfence (load fence), sfence (store fence) and mfence (memory fence). In the case of Intel 86 architecture, mfence operation is defined as a serialization operation on all memory accesses before the mfence. Therefore instructions that follow mfence are guaranteed to see all previous memory accesses in order.

In summary a system is WC memory consistent if it is a cache coherent system, it is running programs with mfence (or similar operation that make certain all memory operations on processor are executed) and it is using shared variable synchronization instructions.

5.3.2 Release Consistency (RC)

RC extends WC by dividing synchronization operations into acquire and release.
Acquire operations are read operations used to gain access to a set of variables. For example accessing a flag within a while operation, see Figure 21, above.

Release operations are write operations that grant access to a variable by another processor. For example setting the flag, see Figure 21, above.

Use of acquire/release pair allows a potential extra degree of parallelism. In the example shown on Figure 23 below notice that the program segment 1 does not share memory accesses with the acquire operation therefore they can run in parallel – acquire operation can be re-ordered. Same applies to the release operation; memory accesses after it do not have to wait for the release to complete. However program segment 3 needs to wait till program segment 1 is completed.

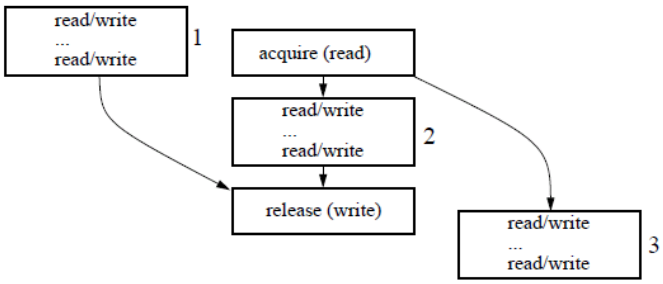

Figure 23. Release Consistency [1]

5.3.3 Digital Alpha, Sparc V9 relaxed memory ordering, IBM PowerPC

These systems provide HW hooks that are used by compilers to insert memory barrier operations when instructed by the programmer. All paralleled program threads wait on a barrier till all other threads arrive. All threads live barrier together therefore memory synchronization is enforced.

Different system provide different flavors' of memory barriers: memory barrier, write memory barrier, memory barrier with 4 extra bits for read-to-read, read-to-write, write-to-read, and write-to write orderings, etc.

In [7] a multithreaded MP cache coherent system accompanied with different shared memory consistency models was simulated. It was assumed that thread context switching takes place on a cache miss.

Authors have developed a method of measuring performance of the following 4 shared memory consistency models: Weak Consistency (WC- Dubois et al. 1986), the Processor Consistency (PC - model by Goodman 1989), and the Release Consistency (RC - Gharachorloo et al. 1990). Table that follow summarizes memory events under SC, WC and RC memory consistency model:

Table 3. Multithreaded Processor Definition for SC, WC and RC, based on [7]

For Each Thread	*SC*	*WC*	*RC*
Read	Processor issues a read access and waits for the read to perform. Context switch on a cache miss.	Processor stalls for a pending release to perform. Processor issues a read access and waits for the read to perform. Context switch on a cache miss.	Processor issues a read access and waits for the read to perform. Context switch on a cache miss.
Write	Processor issues a write access and waits for the write to perform. Context switch on a cache miss.	Processor sends a write to the write buffer, stall if the buffer is full. Writes are pipelined.	Processor sends a write to the write buffer, stall if the buffer is full. Writes are pipelined
Acquire	Same as *Read.*	Processor stalls for pending writes or releases to perform. Processor sends acquire and wait for acquire to perform.	Processor sends acquire and wait for acquire to perform. Context switch on a cache miss
		Context switch on a cache miss.	
Release	Same as *Write*	Processor sends a release to the write buffer, stall if the buffer is full. The release request is retired from the buffer only after all previous writes are performed.	Processor sends a release to the write buffer, stall if the buffer is full. The release request is retired from the buffer only after all previous writes and releases are performed.

The following results were observed:

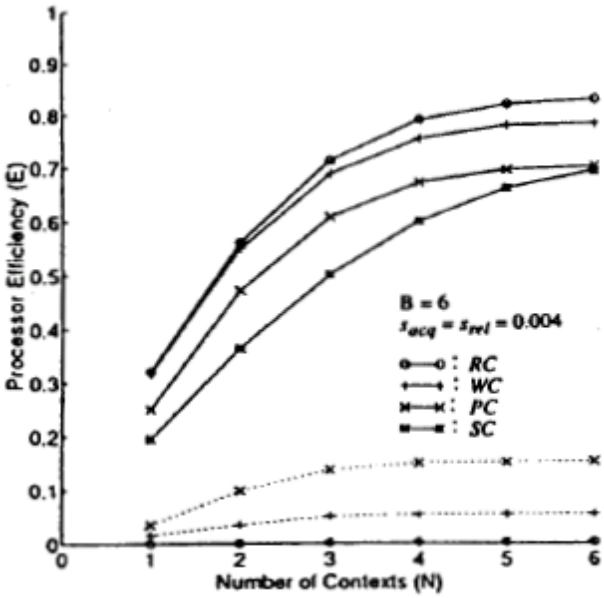

Figure 24. Compare RC, WC and SC Shared Memory Models [7]

RC in all tests gave the best results, SC the worst. WC was very close to RC in all but one test. RC model performs the best by hiding memory write latency with a small write buffer size. WC is somehow limited by the application code synchronization instructions. If this synchronization rate was low WC and RC performance was almost identical and both were able to hide write

latencies. If synchronization rate was high WC would lag RC due to the processor synchronization stalls.

It was found that multithreading benefits were more than 50% of the total performance improvement while memory consistency models contribution would vary between 20% and 40%.

6. Contemporary relaxed consistency used in shared MPSoC and multicore

6.1 Embedded Processors

6.1.1 ARM

ARM [12] for their embedded MPSoC provides a memory consistency model that is between TSO (Total Store Ordering) and PSO (Partial Store Ordering). TSO allows a read to bypass an earlier write, while PSO in addition to TSO allows write to write reordering. ARMs' model is supported by SW synchronizations and barriers.

6.1.2 Xtensa

Xtensa [11] configurable and extensible processors from Tensilica are used in MPSoC. In order to synchronize individual processors and enable data sharing via a shared memory Xtensa architecture provides support for relevant synchronization instructions. This model is not classified by Xtensa however it looks as Weak Ordering.

Mutex (a lock function) protects shared data from parallel modifications of critical code sections. If mutex is unlocked no threads/cores owns the critical code, if it is locked the critical section is owned by one thread/core.

Barrier in an instruction that requires a number of threads/cores to wait at the barrier for any of relevant threads/cores can proceed with program execution.

S32C1I is a read-conditional-write instruction; write is done only if read value is equal to the expected value.

If a shared memory that holds the mutex variable is cacheable it is required to maintain the cache coherence. Therefore updates to the mutex variable have to be made available to all processors as mutex variable reads cannot come from the cache. Instruction S32C1I automatically manages this coherence.

Xtensa HW as well as Xtensa compiler can reorder memory related accesses. To preserve memory consistency both of those need to be told not to reorder memory accesses by implementing *flush_memory* pragma (translated to MEMW instruction). There is also option for some of Xtensa cores to separately instruct compiler or HW not to reorder memory instructions (compiler - no_reorder_memory pragma, HW - S32RI instruction). The former approach gives better overall processor performance.

Figure 25 shows an Xtensa based multiprocessor system with private data/instruction memories and a small shared memory that constrains synchronization data.

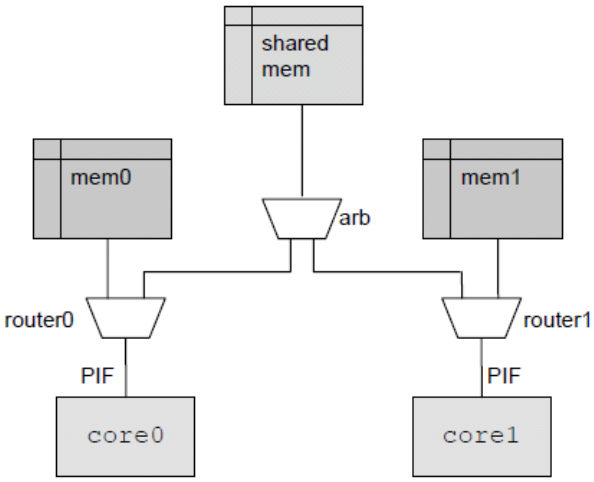

Figure 25. Xtensa Shared Memory with Synchronization Data [11]

Also note that programs written for specific platform lack portability however in the case of embedded systems this is almost given.

6.2 Processors

In [17] a memory consistency models are grouped and linked to industry solution:

- Strong: requires Write atomicity and does not allow local bypassing

- Weak: requires Write atomicity and allows local bypassing
- Weakest: does not require Write atomicity and allows local bypassing
- Hybrid: supports weak load and store instructions that come under Weakest memory model class and also support strong load and store instructions that come under Strong or Weak memory model class

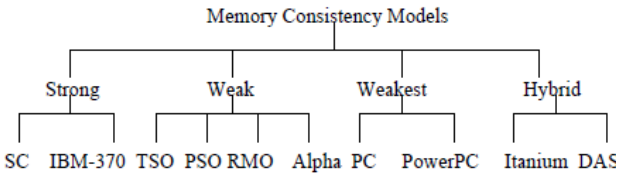

Figure 26. Processor Memory Consistency Models

7. Conclusion

In this paper benefits and requirements for correct use of shared memory in multiprocessor systems are presented. It was shown that even the ultra high speed cache coherent architectures (Intel QPI based CC-NUMA) or much slower Ethernet network based distributed multiprocessing systems still require memory consistency.

Memory consistency is a well researched topic resulting in a number of consistency models accepted in the industry. All of these consistency models are implementable in software or hardware with different levels of complexity and benefits to a system performance. However in most of the industry solutions memory consistency is achieved based on the hardware provided cache coherence algorithms and software synchronization instructions supported by simple hardware hooks. In all of the solutions a parallel software programmer still needs to be aware of instructions that require synchronization and establish memory consistency with fence-like instructions before parallel program threads can safely exchange data.

8. Appendix

Why is OSI model used in packet networks?

OSI model is developed and used to insure interoperability in systems that use multi-vendor developed devices. Software drivers, protocol stacks, network layers from different vendors can be used together.

Why is concept of Shared Memory used?

Shared Memory is used to enable exchange of data among individual parallel processors. By using Shared Memory we avoid data copy benefiting in speed and memory capacity as well as we enable processors to pass synchronizations events to each other.

What are alternatives to Shared Memory?

If multiprocessors use respective local memories data exchange would require one processor (A) to stream data to other processors' (B) local memory. After it is done the requesting processor (A) would create an event to the second processor (B) to let it know that new data is available. This approach still requires processors to share memory as well as use of events.

Why Intel CC-NUMA does not guarantee memory consistency even though QPI bus provides extreme interconnect speed?

Memory consistency issues are created by the processor itself due to write buffers, reads bypassing writes etc. Therefore the system interconnects speed if nothing else can make memory consistency issues even more frequent.

What is the main advantage and the main disadvantage of the page based shared memory?

The main advantage is that neither new hardware nor the modifications to the existing hardware is required. The main disadvantage is high probability of the false data sharing therefore page copy is often required which itself is problematic due to the page size. As a consequence the memory consistency software algorithms used in these systems need to be efficient.

9. References

[1] David Culler, Jaswinder Pal Singh, Anoop Gupta , *Parallel Computer Architecture A Hardware / Software Approach*, Morgan Kaufmann, ISBN 13: 9781558603431

[2] Sarita V. Adve, Kourosh Gharachorloo , "Shared Memory Consistency Model, A Tutorial", *Digital Western Research Laboratory*, Research Report 95/7

[3] Sarita V. Adve, Mark D. Hill, "Weak Ordering - A New Definition", Computer Sciences Department University of Wisconsin Madison, Wisconsin 53706

[4] M. Dubois, C. Scheurich and F. A. Briggs, "Memory Access Buffering in Multiprocessors", *Proc. Thirteenth Annual International Symposium on Computer Architecture* 14. 2 (June 1986),434-442.

[5] C. Amza, "Tread Marks: Shared memory computing on networks of workstations", *IEEE Computer,vol.29*, February 1996, pp.18-28.

[6] P. Keleher, "Lazy Release Consistency for Distributed Shared Memory", *Ph.D. Thesis*, Dept. of Computer Science, Rice University, 1995.

[7] Yong-Kim Chong and Kai Hwang, "Performance Analysis of Four Memory Consistency Models for Multithreaded Multiprocessors", *IEEE Transactions on Parallel and Distributed System*, Volume 6 Issue 10, October 1995

[8] Leslie Lamport "How to make a multiprocessor computer that correctly executes multiprocess programs" *IEEE Transactions on Computers*, September 1979

[9] Leonidas I. Kontothanassis, Michael L. Scott, Ricardo Bianchini, "Lazy Release Consistency for Hardware-Coherent Multiprocessors", Department of Computer Science, University of Rochester, December 1994

[10] Kunle Olukotun, Lance Hammond, and James Laudon , *Chip Multiprocessor Architecture: Techniques to Improve Throughput and Latency*, Morgan & Claypool, 2007

[11] Xtensa Inc, "Implementing a Memory-Based Mutex and Barrier Synchronization" *Application Note*, July, 2007

[12] John Goodacre, "MPSoC – System Architecture, ARM Multiprocessing" http://www.mpsoc-forum.org/2003/slides/MPSoC_ARM_MP_Architecture.pdf, , ARM Ltd. U.K.

[13] J.H.G.M. Vrijnsen, "A NUMA Architecture Simulator", Section of Information and Communication Systems (ICS), Faculty of Electrical Engineering, ICS/ES 815, Practical Training Report.

[14] Intel, http://www.intel.com/technology/quickpath/

[15] UNC Charlotte: http://coitweb.uncc.edu/~abw/parallel/par_prog/resources.htm

[16] University of California Irvin, http://www.ics.uci.edu/~javid/dsm.html

[17] Prosenjit Chatterjee, "Formal Specification and Verification of Memory Consistency Models of Shared Memory Multiprocessors" *Master of Science Thesis*, The University of Utah, 2009

[18] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons,Ahoop Gupta, and John Hennessy, "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors", Computer Systems Laboratory, Stanford University, CA

[19] John L. Hennessy and David A. Patterson *Computer Architecture, A Quantitative Approach*, Morgan Kaufmann, ISBN: 978-0-12-370490-0