# RECENT PROGRESS IN MULTIPROCESSOR THREAD SCHEDULING

*Daniel Shapiro*

Computer Architecture Research Group
School of Information Technology and Engineering
University of Ottawa
E-mail: dshap092@uottawa.ca

## ABSTRACT

In this paper we will discuss progress in the area of thread scheduling for multiprocessors, including systems which are Chip-MultiProcessors (CMP), can perform Simultaneous MultiThreading (SMT), and/or support multiple threads to execute in parallel. The reviewed papers approach thread scheduling from the aspects of resource utilization, thread priority, Operating System (OS) effects, and interrupts. The metrics used by the discussed papers will be summarized.

***Index Terms***— Scheduling, multiprocessor, threads, priority.

## 1. INTRODUCTION: COMMON ISSUES IN THREAD SCHEDULING

As multithreading multiprocessors become pervasive in consumer and industrial settings, a host of problems have been addressed recently in the literature that were not problematic for uniprocessors. In this paper we look at recent developments in the field, and examine some of the background and results of this body of work. Some problems of particular importance are controlling the priority of a thread at the hardware level, whereas classic threads were only controllable at the OS scheduling level. Scheduling in general for mutliprocessors becomes problematic because the list scheduling approach does not work anymore in the task to processor assignment. Resource utilization was expected to be an important metric, but as we will see it does not seem to express the true behavior of the multithreading multiprocessor. Instead we find that speedup and task throughput are better metrics for this application area. We can look at the multiprocessor as a black box and observe factors such as quality of service and response time as measures of system quality. We start with a review of the contributions of each paper, and then proceed to examine the primary metrics used to evaluate outcomes. The papers are categorized into clusters according to their relationships to each other and conclusions are drawn based upon the observed trends in the literature.

## 2. REVIEW OF RECENT PRIOR ART

In this section the major contributions are presented for 17 recent papers that are related to the scheduling of threads for multiprocessors. We begin with [1], which attempts to enforce priorities in a simultaneous multi-threading processing system. The trouble is that the fetch policy is not visible to the OS and so software priorities are not followed. This paper provides three new quality of service features to address this problem: execution monitoring that is exposed to the OS, priority-based resource management, and thread protection.

Preemption Threshold Scheduling (PTS) for memory limited hard real-time systems was examined in [2]. They found that when the objective is minimizing stack memory requirement, PTS is the optimal scheduling algorithm. They show a method for reducing the average worst-case response time, and a new framework for applying PTS to priority-based scheduling algorithms.

The work in [3] focuses on thread migration and thread to core affinity. The experiments were performed in simulation, and the results indicated that thread migration algorithms performed better than non-thread migration scheduling algorithms. Algorithms which considered core affinity and thread migration performed the best.

In [4] Cycles Per Instruction Spent on Memory ($CPI_{MEM}$) is used to measure the resource demands. Mix-Scheduling is used to even the $CPI_{MEM}$ across the cores by migrating threads.

Threads are meant to be parallel, but [5] notices that threads interference by memory access can serialize the schedule. Their solution is to have a parallelism-aware batch scheduler that preserves memory-level parallelism across scheduled threads. Furthermore, the scheduler prevents thread starvation, and promotes fairness by priority.

Dynamic scheduling for NoC is considered by [6], where Preemptive Virtual Clock (PVC) is proposed as the Quality of Service (QoS) solution. PVC provides QoS guarantees, reduces packet jitter, and allocates network bandwidth with a common buffer for each source node. PVC addresses priority inversion using packet preemption. There is a tradeoff between priority enforcement and throughput that is tunable

within the PVC approach. The strength of the QoS guarantees degrades as priority enforcement is traded for throughput. Another interesting feature in PVC is bandwidth allocation. Like WiMax, PVC allows the fixed allocation of bandwidth to applications, regardless of the number of threads in the application. The allocation can be tuned at runtime.

A fair scheduling strategy for SMT is proposed in [7] where thread priority is enforced. As mentioned earlier, the OS cannot access the hardware priorities, and this paper addresses that drawback in the thread-architecture interface. This approach combines the instructions' and processor's architectural state into the scheduling strategy.

Another paper that looks at priority enforcement for SMT is [8]. The problem of OS sending thread priority is addressed in real systems: Linux running on both multicore (core 2 duo) and SMT (Pentium 4) processors. They describe their framework for scheduling threads that is based upon on-line statistics collected by hardware performance counters.

Unlike the past few papers, [9] looks at a real-time embedded application with only 2 priority levels for tasks. The system is realized as a shared memory multiprocessor on a Xilinx FPGA. The cores are MicroBlaze, and the FPGA is a Virtex-II Pro. Runtime behavior is compared to the simulated performance. A hardware unit for scheduling is described where a large common global ready queue is maintained for periodic and aperiodic low priority tasks. High priority tasks are scheduled into core dedicated task queues.

The solution of [10] for the thread priority problem in SMT processing systems is to design a new processor and scheduler in hardware which understand thread priority. Their real-time processor is called Responsive MultiThreaded (RMT) and it is also designed to control the instructions per cycle of each thread, which speeds up or slows down the thread throughput. This method of observing and manipulating the IPC (instructions per cycle, not to be confused with interprocessor communication) to achieve the throughput corresponding to the thread priority is called IPC control. The RMT processor performs branch prediction and contains 6 million gates for the processor alone. It can run eight threads in parallel and allows for 256 priority levels. Thread access to all of the hardware is priority based, including the memory system and issuing logic. If less than 8 threads are scheduled, then they can be statically scheduled. Otherwise a software scheduler can be used.

The response time of the OS was studied in [11], and they found that static scheduling assumptions can be very different from the real behavior due to unresponsiveness of the OS. The schedule can be stretched out in quite undesirable ways when a task on the critical path of the schedule is waiting for the OS to respond. The problem is pronounced when many cores are depending on a centralized OS. The jitter in OS response time can have a variety of sources including "user space processes, kernel threads, interrupts, SMT interference and hypervisor activity". Co-scheduling, the idea of scheduling tasks simultaneously across the cores is one way to reduce the observed OS jitter. Another possibility is the use of microkernels at the various cores. Many complex sources of jitter are not easily avoided. For example a direct mapped cache shared between cores will cause interference that just cannot be programmed away. This interference can also happen between threads that are running together in a superscalar processor, and during process migration. By using the extra thread bandwidth in the system, and also the unused cores, [11] proposes that reducing jitter is possible. The reduction of the number of active kernel threads, intelligent interrupt handling, and thread priority tuning are also used to attack the problem of OS jitter. Their implementation was specific to the Power Architecture and the Linux OS.

Performance isolation is the opposite of thread interference, where co-runners disrupt the schedule in unanticipated ways due to hardware resource conflicts. A novel scheduling algorithm for the OS of a multiprocessor is described in [12]. Interestingly, this paper explains how co-runners can interfere in terms of memory accesses such as cache reservation. The described approach preserves priority enforcement by the OS, and provides QoS capability. The solution is software based and it is aware or the cache allocation policy. The cache allocation is not guaranteed to be fair, instead the application runs just as quickly as it would if there had been fair cache allocation. Similarly to the approach of bandwidth tuning in [6] and the IPC control in [10], [12] provides variable time slicing for the threads in order to tune the resource allocation among threads.

A resource-aware, low-power distributed real-time OS was implemented in [13] for a Wireless Sensor Network (WSN) application. Their system allows for threads and for events, which each have their own scheduling policy. Threads can be preempted but not regular tasks, and all of the scheduling is priority based. The system was deployed on an Atmel AT91SAM7S256-EK evaluation board.

Fairness, priority inversion, operating system priorities being lost at the hardware scheduler level, spatial and temporal scheduling support, heterogeneous multiprocessing, SMP, and other topics are discussed in [14] in the context of real-time Java. The Real-Time Specification for Java (RTSJ) is not up to date, and so it does not account for emerging aspects of parallel computing. The idea is to expose these various architecture features to the Java language in an abstract way.

Security for the system from malicious threads is the focus of [15]. The proposed system is called Composite and it is designed to be dependable and predictable, even though it executes untrusted code in threads. The system is be configurable to allow some threads more access than others to the architecture. Each thread can be associated with a set of services and policies. Like a server, the user time is restricted, and so threads do not become zombies. This creates good predictability for scheduling. One big problem faced by this system is interrupt handling. To ensure that interrupts are ser-

viced fast enough, Composite allows user-defined scheduling policies that combine interrupt and task management. Safe memory access issues and also safe interrupts are supported by allowing for hierarchical access policies to be created for the scheduler.

Threads are usually managed by the OS scheduler, and interrupt handlers are managed by the hardware. This means that in a real-time system a low priority interrupt handler can interrupt high-priority thread. For [16], the solution to the priority problem between the OS-managed threads, and the hardware running the threads is to rewrite threads as interrupts. Their system is called SLOTH and it is compatible with a wide variety of real-time systems already out there.

In [17] the thread as interrupt paradigm is discussed from another perspective. They discuss interrupt handlers that execute within threads. This allows the system to control the interference of interrupts with high priority threads. Using fixed priorities proves for these threads does not work well as a tradeoff between predictability and speed. The resource reservation abstraction is proposed in this paper in order to allow for scheduling analysis. Reservation-based scheduling with the CBS algorithm was used to interrupt threads, and the priority tweaking for the interrupt threads were validated with experiments on a real-time Linux kernel.

## 3. COMPARISON OF REVIEWED PAPERS

The metrics used when studying thread scheduling include resource utilization (ALU, memory), throughput, speedup, execution time, IPC, cache miss rates, and response time. These metrics are accompanied by units of measurement. In [2, 9, 10, 13], the units for resource utilization were memory bytes [2, 13], shared processor hardware [9], and instruction buffer [10]. This difference in focus indicates that the resource utilization although critical to scheduling, is not necessarily a good metric for performance when the whole system is shared by a group of software threads. One can imagine that keeping hardware units busy (e.g. in livelock) or keeping memory usage low (e.g. as the system crashes) does not necessarily translate into a higher throughput of threads, or a bigger speedup.

In [1, 4, 5, 6, 8, 10, 12], throughput is discussed in terms of the balance between instructions issued and fairness to priority. Metrics such as the error rate on IPC control give a measure of the impact on performance caused by balancing priority and throughput. IPC control allocates resources to real-time threads and the error rate on the allocation of resources (resource should be allocated according to priority controls but is unavailable) is the IPC error rate. Clearly, throughput maximization in a system with multiple threads is nominally good, but as we will see for the IPC metric, this number does not represent a full picture of the system performance. An alternative and common metric for system comparison is speedup. Because it is a ratio of the performance of two systems, speedup is not often accompanied by units. [12] used slowdown as a measure of performance, but then also had negative slowdown to represent speedup. Speedup was used as a metric in [5, 7, 8, 12, 11]. This metric is very commonly used in computer architecture papers for expressing the benefit of an approach. However, speedup may abstract the finer grained details of a multiple thread program, where situations such as priority inversion and thread starvation are often more important than the overall system speedup.

[2, 3, 5, 6, 9, 10, 11, 12] used seconds, microseconds, abstract units in a simulation (called time, latency), and cycles as units for execution time. While cycles and ticks in a simulation remove the clock frequency from consideration, real time units provide a much better understanding of the real-world implications of a given approach. The units for instructions per cycle are implicit, but [7, 12] normalized the collected samples, while [4] gave an average IPC, and [7, 10] presented raw IPC values. Maximizing IPC does not guarantee better throughput, as we have seen in the RISC versus CISC comparison. Specifically, RISC has a simpler instruction set and so even though the Cycles Per Instruction (CPI) is minimized compared to the CISC, the CISC can perform a broader range of operations, and can directly access the global memory in a single cycle. Having said this caveat, IPC is probably interesting for thread analysis because it gives an idea of the number of events happening in parallel.

When discussing execution time and scheduling for multiprocessors, it is helpful to define the critical path in a parallel program which limits the execution time to some minimum value. This number is called the makespan, and is composed of the critical path along the optimal schedule of the tasks which are in our case threads. Another term to keep in mind is Worst-Case Execution Time (WCET), which is the measure of the longest possible path resulting from a valid schedule of the tasks. A valid schedule is one with latency between the end of one task and the start of the next only when the scheduled task is waiting for the results of a currently executing task or obtaining data from a task that has completed on another processor.

Cache miss rates were touched on by [4, 12], where the number of misses was observed by [4] and the miss rate was observed by [12]. Scheduling is so sensitive to cache coherence policy, size, and structure that it should probably be reported on more often. Sometimes there is not enough data available on the inner workings of the cache at runtime, and so a simulator or hardware debug support is required. In some cases there is no cache or limited caching present in the hardware, such as the multiprocessor MicroBlaze system in [9] where there is a local memory for data and an L1 instruction cache.

Response time and/or the jitter in response time was noted by [6, 9, 11, 12] and was represented as a percentage of total execution time [11], and a time spent waiting. Jitter is typ-

**Table 1**. Metrics utilized in each paper discussed.

| Paper | Resource Utilization | Throughput | Speedup | Execution Time | Instructions Per Cycle | Cache Miss Rate | Response Time |
|---|---|---|---|---|---|---|---|
| [1] | | V | | | | | |
| [2] | Stack memory | | | Worst-case execution cycles | | | Average worst case |
| [3] | | | | Makespan | | | |
| [4] | | V | | | Average | L1/L2 | |
| [5] | | V | V | Thread turnaround time | | | |
| [6] | | V | | Average packet latency | | | Packet arrival jitter |
| [7] | | | V | | V | | |
| [8] | | V | Of high-priority threads | | | | |
| [9] | V | | | WCET | | | V |
| [10] | Instruction buffer writes | Error rate on IPC control | | V | V | | |
| [11] | | | V | Makespan, Jitter duration | | | Total Jitter % |
| [12] | | Transactions per second | V | Normalized | Normalized | Misses per 10K cyc | Jitter |
| [13] | Memory | | | | | | |

ically random a runtime effect, and so static scheduling will not be able to take such noise into account easily.

The benchmark used to obtain the stated metrics were IDCT in [10], MiBench in [9, 7], SPEC CPU2000 in [1, 12, 7, 4], Trace Collector in [11], PapaBench in [2], PARSEC in [6], netperf in [17], SPEC CPU2006 for [8], PCMark05 in the related work of [3]. It is proposed in [14] that a new benchmark suite is needed for real-time Java on parallel processors. This will only continue the trend one can see in performance evaluation where the benchmarks used are not the same even for a small cluster of papers in a subspecialty of computer architecture. In [15] the programs referred to as "microbenchmarks" were non-standard, looking at hardware measurements, Linux primitives, and the performance of Composite operating system primitives. It is worth noting that SPEC CPU and MiBench are broadly used.

The utilized thread scheduling metrics are summarized in Table 1. Once the data is aggregated this way, one can see that throughput is a popular metric, along with execution time.

We can organize the aforementioned papers into clusters according to topic. The identified clusters are scheduling to improve thread performance [1, 2, 3, 4, 5, 14], thread priority schemes [6, 7, 8, 9, 10, 14], OS impact on thread performance [11, 12, 13], and the relationship between interrupts and threads [15, 16, 17].

The approaches studied here have quantitatively moved forward the state of the art. For example, in [2] the stack size was provably to be minimized. In [1], 90% of the throughput obtained by a state of the art fetch policy was observed (10% lost due to priority enforcement). Their work showed that there is a tradeoff between throughput and priority in SMT processors. In [8], high-priority threads saw a speedup as high as 50%, with equal or better overall throughput. [15] used an example program to show that their thread halting was more than twice as fast as in a regular Linux OS. In [2] the task worst case response time improved by 15% to 50%. These are all big moves in performance according to the metrics that the authors set out to address.

## 4. CONCLUSION

A set of recent papers on the topic of multiprocessor thread scheduling was reviewed. These contributions were organized into clusters, and compared based upon their content and the metrics they used to define their problem. The identified clusters were scheduling to improve thread performance, thread priority schemes, the OS impact on thread performance, and the relationship between interrupts and threads. The reviewed papers provided good results for very interesting problems in a variety of domains. For example, some authors solve thread interference with rewriting of the software, some with interfaces into the hardware, and others designed custom hardware from the ground up just to address the problem. Tuning the scheduling at runtime was another common theme. In general the papers addressed problems with SMT processing systems such as priority inversion. The reviewed papers approached thread scheduling from the aspects of resource utilization, thread priority, Operating System (OS) effects, and interrupts.

## 5. REFERENCES


[1] F.J. Cazorla, P.M.W. Knijnenburg, R. Sakellariou, E. Fernandez, A. Ramirez, and M. Valero, "Predictable performance in SMT processors: synergy between the OS and SMTs," *Computers, IEEE Transactions on*, vol. 55, no. 7, pp. 785 – 799, july 2006.

[2] R. Ghattas and A.C. Dean, "Preemption threshold scheduling: Stack optimality, enhancements and analysis," in *Real Time and Embedded Technology and Applications Symposium, 2007. RTAS '07. 13th IEEE*, april 2007, pp. 147 –157.

[3] F.N. Sibai, "Performance effect of localized thread schedules in heterogeneous multi-core processors," in *Innovations in Information Technology, 2007. IIT '07. 4th International Conference on*, nov. 2007, pp. 292 – 296.

[4] Lichen Weng and Chen Liu, "On better performance from scheduling threads according to resource demands in MMMP," in *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, sept. 2010, pp. 339 –345.

[5] O. Mutlu and T. Moscibroda, "Parallelism-aware batch scheduling: Enabling high-performance and fair shared memory controllers," *Micro, IEEE*, vol. 29, no. 1, pp. 22 –32, jan.-feb. 2009.

[6] B. Grot, S.W. Keckler, and O. Mutlu, "Preemptive virtual clock: A flexible, efficient, and cost-effective QOS scheme for networks-on-chip," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, dec. 2009, pp. 268 –279.

[7] Cheng Lian and Yang Quansheng, "Thread priority sensitive simultaneous multi-threading fair scheduling strategy," in *Computational Intelligence and Software Engineering, 2009. CiSE 2009. International Conference on*, dec. 2009, pp. 1 –4.

[8] J.C. Saez, J.I. Gomez, and M. Prieto, "Improving priority enforcement via non-work-conserving scheduling," in *Parallel Processing, 2008. ICPP '08. 37th International Conference on*, sept. 2008, pp. 99 –106.

[9] A. Tumeo, M. Branca, L. Camerini, M. Ceriani, M. Monchiero, G. Palermo, F. Ferrandi, and D. Sciuto, "A dual-priority real-time multiprocessor system on FPGA for automotive applications," in *Design, Automation and Test in Europe, 2008. DATE '08*, march 2008, pp. 1039 –1044.

[10] N. Yamasaki, I. Magaki, and T. Itou, "Prioritized SMT architecture with IPC control method for real-time processing," in *Real Time and Embedded Technology and Applications Symposium, 2007. RTAS '07. 13th IEEE*, april 2007, pp. 12 –21.

[11] P. De, V. Mann, and U. Mittaly, "Handling OS jitter on multicore multithreaded systems," in *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, may 2009, pp. 1 –12.

[12] A. Fedorova and M. Seltzer, "Improving performance isolation on chip multiprocessors via an operating system scheduler," in *Parallel Architecture and Compilation Techniques, 2007. PACT 2007. 16th International Conference on*, sept. 2007, pp. 25 –38.

[13] Hai ying Zhou and Kun mean Hou, "LIMOS: A lightweight multi-threading operating system dedicated to wireless sensor networks," in *Wireless Communications, Networking and Mobile Computing, 2007. WiCom 2007. International Conference on*, sept. 2007, pp. 3051 –3054.

[14] V. Olaru, A. Hangan, G. Sebestyen-Pal, and G. Saplacan, "Real-time java and multi-core architectures," in *Intelligent Computer Communication and Processing, 2008. ICCP 2008. 4th International Conference on*, aug. 2008, pp. 215 –222.

[15] G. Parmer and R. West, "Predictable interrupt management and scheduling in the composite component-based system," in *Real-Time Systems Symposium, 2008*, dec. 2008, pp. 232 –243.

[16] W. Hofer, D. Lohmann, F. Scheler, and W. Schroder-Preikschat, "Sloth: Threads as interrupts," in *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, dec. 2009, pp. 204 –213.

[17] N. Manica, L. Abeni, and L. Palopoli, "Reservation-based interrupt scheduling," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2010 16th IEEE*, april 2010, pp. 46 –55.