An Infrastructure for Efficient Synchronization of Asymmetric Threads on Hyper-Threaded Processors

Nikos Anastopoulos, Nectarios Koziris[1]

Computing Systems Laboratory, School of Electrical and Computer Engineering,
National Technical University of Athens, Zografou Campus, Zografou 15780, Greece

ABSTRACT

So far, the privileged instructions MONITOR and MWAIT introduced with Intel Prescott core, have been used mostly for inter-thread synchronization in operating systems code. In a hyper-threaded processor, these instructions offer a "performance-optimized" way for threads involved in synchronization events to wait on a condition. In this work, we explore the potential of using these instructions for synchronizing application threads on hyper-threaded processors which are characterized by workload imbalance.

Initially, we propose a framework through which one can use MONITOR/MWAIT to build condition wait and notification primitives, with minimal kernel involvement. Then, we evaluate the efficiency of these primitives in an incremental manner: at first, we quantify certain performance aspects of the primitives that reflect the considered execution model, such as resource consumption and responsiveness. As a further step, we use our primitives to build synchronization barriers. We examine the same performance issues as before, and we evaluate also the efficiency of our implementation for fine-grained inter-thread synchronization. Finally, we test our barriers in a real-world scenario, and specifically, in applying thread-level Speculative Precomputation. For the application model under consideration, our proposed primitives is the option that best balances low resource consumption with high responsiveness and reduced call overhead.

1 Introduction

Simultaneous Multithreading (SMT) [TEL95] allows a superscalar processor to issue instructions from multiple independent threads to its functional units, in a single cycle. Hyper-threading technology [MBH$^+$02] is Intel's two-threaded, low-end approach to SMT. In a hyper-threaded processor, almost all resources are shared, and only the architectural state, along with any control-flow related structures, are replicated for each thread.

In such an "all-shared" environment, implementation of synchronization is a key factor for multithreaded performance. Synchronization primitives based on spin-wait loops have been commonplace in traditional multi-processor systems, but can incur significant performance penalty in an SMT environment, especially when one or more threads wait

[1]E-mail: {anastop,nkoziris}@cslab.ece.ntua.gr

on synchronization events for long periods. Although hardware extensions have been proposed for SMT to support low-latency, resource-friendly synchronization [TLEL99], unfortunately hyper-threading does not provide similar mechanisms. There are other techniques that could be employed in order to mitigate the excessive resource consumption of spin loops, which, however, lack high responsiveness.

A first option would be primitives that resort to OS mediation to handle long condition-wait periods. The waiting thread would yield its logical processor and release all its resources, but its notification and resumption would be expensive in terms of cycles, due to the invocation of the scheduler. A second solution would be to loosen the spinning by embedding the PAUSE instruction in the spin-wait loop. Although in this way the waiting thread is prevented from aggressively consuming dynamically shared resources, the cost of spinning is not entirely eliminated because the thread still holds its share of entries in the statically partitioned queues. A third approach would be to use the HALT instruction, through which a thread can go to sleeping mode making all its statically and dynamically shared resources available to the peer context. However, kernel privileges are required to execute HALT and to notify the sleeping thread, which translates into system call overhead.

Intel's Prescott core introduced a new pair of instructions, MONITOR and MWAIT, which implement a condition-wait as close as possible to the hardware level: MWAIT enables a logical processor to enter into a "performance-optimized" state while waiting for a single store to the address range set up by MONITOR. MONITOR/MWAIT are similar with HALT in that they release all shared and partitioned resources of a hyper-thread, and require kernel privileges. However, they obviate the need for expensive IPI delivery to exit the sleeping state, since they require just a single memory update for this purpose. In this work we explore the potential of using MONITOR/MWAIT to synchronize application-level threads on hyper-threaded processors, with imbalanced workloads.

2 Framework for Implementing Synchronization Primitives

In their initial implementation, MONITOR and MWAIT are available at privilege level 0 only. Therefore, in order to be able to implement a condition-wait primitive to be used at the application level, a system call just to access these instructions is unavoidable. What comes next in terms of extra cost, is the way that condition-wait (occurring always at kernel-space) and notification primitives communicate each other the contents of the triggering address.

In order to establish the fastest possible communication between kernel and user-space, we allocate the monitored memory region (a cache-line sized region, actually) in the context of a special character device in kernel-space, and then map the device to user-space. The memory region can be then directly accessed both from kernel and user-space, without any redundant copy operations or additional system calls.

After that, the implementation of a system call that uses MONITOR and MWAIT is straightforward. We extended the Linux kernel with the `mwmon_mmap_sleep` system call, which implements the condition-wait primitive. A typical program should make the following steps in order to use our proposed primitives: at initialization phase, the program should open the special device for read and write, and then mmap it in its address space. A thread that wishes to wait on a condition, calls `mwmon_mmap_sleep`. A thread that wishes to notify the waiting thread, sets a random byte within the monitored memory region to a pre-defined value. At finalization, the program unmaps and then closes the device.

3 Performance Evaluation

Throughout our work, we have considered an application model where two asymmetric threads are executing on the two contexts of a hyper-threaded processor. One of the two threads is the *heavyweight* (or *worker*), performing computations throughout its entire execution, and the other is the *lightweight* (or *helper*), whose execution alternates between short periods of useful work and long idle periods. When in idle mode, the lightweight thread waits until it is notified by the heavyweight before proceeding. An example of this execution scenario is depicted in Fig. 3, where threads are synchronized with barriers.

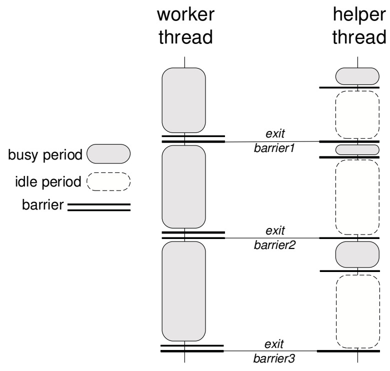

Figure 1: Application model under consideration.

Three requirements must be met in order synchronization to be effective for this execution model: first, the helper thread must not introduce significant impediment to the worker thread while waiting, by excessively consuming shared resources. Second, the helper thread must resume as fast as possible each time it is notified by the worker, in order its actions to be timely and accurate. Finally, the time that the main thread needs to invoke a synchronization primitive in order to notify the helper, must be as little as possible. Since these requirements are normally conflicting, we look for an option that best balances them.

We compared our MONITOR/MWAIT-based synchronization mechanisms (*mwmon*) with implementations based on spin-wait loops with the PAUSE instruction (*spin-loops*), spin-wait loops with the HALT instruction (*spin-loops-halt*), and synchronization primitives offered by the NPTL library (*pthreads*). We experimented on an Intel Xeon processor, one of the first mainstream chips to encompass low-end SMT capabilities.

The evaluation process followed a bottom-up approach: First, we measured raw performance of condition-wait and notification primitives by measuring performance aspects such as resource consumption and responsiveness. While *spin-loops* version provided best response times, as expected, it decelerated the worker thread significantly. Among the rest implementations, which were the most resource-conserving ones, *mwmon* was the option with the lowest wakeup times and call overhead.

As a second step, we used our primitives to build synchronization barriers. Again, *mwmon* offered the best combination of low resource consumption, high responsiveness and low call overhead. With respect to *spin-loops*, the waiting thread in *mwmon* introduced 24% less interference to the main thread. Compared to *pthreads*, which was the best option among the three most resource-friendly versions, it provided almost 4 times lower wakeup latency and 3.46 times reduced call overhead. In a fine-grained synchronization scenario, where threads with small, yet varying workloads were synchronized in a loop, the *mwmon* implementation outperformed all other versions in terms of throughput, for all levels of threads asymmetry. On average, it yielded 12% and 26% better throughput compared to *pthreads* and *spin-loops*, respectively.

Finally, we evaluated the various barrier implementations in applying *Speculative Precomputation* (SPR) on a series of real-world applications. In SPR, a helper runs ahead and prefetches data that are going to be used by the worker in its near future. Whenever it has prefetched a certain amount of data, it is throttled, so that it is prevented from running too far ahead and polluting the cache. This synchronization can be implemented with barriers, reflecting the execution scheme of Fig. 3. The reader is referred to [AAKK08] for a more detailed discussion on our implementation of SPR. SPR using *mwmon* barriers outperformed all other cases and offered speedups between 1.07 and 1.35. On average, it yielded 15% better execution times compared to *spin-loops*, and 3.6% compared to *pthreads*.

4 Future Work

As a future work, we intend to extend our framework in order to support MONITOR/MWAIT-based synchronization in multi-SMT systems, following hierarchical schemes (e.g. tree barriers). Additionally, we intend to evaluate our primitives on parallel programs with requirements for fine-grained synchronization, and on other application models that reflect our considered execution scheme, such as MPI programs or I/O bound multithreaded applications. We argue that, with the advent of hybrid architectures that encompass multitude of hardware contexts within a single chip, architecture-aware hierarchical synchronization schemes will play a significant role in parallel application performance and thus seem to be worthwhile to investigate.

References

[AAKK08] E. Athanasaki, N. Anastopoulos, K. Kourtis, and N. Koziris. Exploring the performance limits of simultaneous multithreading for memory intensive applications. *J. Supercomput.*, 44(1):64–97, 2008.

[MBH$^+$02] D. Marr, F. Binns, D. Hill, G. Hinton, D. Koufaty, J. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, Feb 2002.

[TEL95] D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *ISCA '95: Proceedings of the 22th annual international symposium on Computer architecture*, pages 392–403, 1995.

[TLEL99] D. Tullsen, J. Lo, S. Eggers, and H. Levy. Supporting fine-grained synchronization on a simultaneous multithreading processor. In *HPCA '99: Proceedings of the IEEE 5th International Symposium on High Performance Computer Architecture*, page 54, 1999.