Facilitating Efficient Synchronization of Asymmetric Threads on Hyper-Threaded Processors *

Nikos Anastopoulos and Nectarios Koziris

National Technical University of Athens
School of Electrical and Computer Engineering
Computing Systems Laboratory
{anastop,nkoziris}@cslab.ece.ntua.gr

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 that execute on hyper-threaded processors, and are characterized by workload asymmetry. 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 a bottom-up manner: at first, we quantify certain performance aspects of the primitives that reflect the execution model under consideration, such as resource consumption and responsiveness, and we compare them against other commonly used implementations. As a further step, we use our primitives to build synchronization barriers. Again, we examine the same performance issues as before, and using a pseudo-benchmark we evaluate the efficiency of our implementation for fine-grained inter-thread synchronization. In terms of throughput, our barriers yielded 12% better performance on average compared to Pthreads, and 26% compared to a spin-loops-based implementation, for varying levels of threads asymmetry. Finally, we test our barriers in a real-world scenario, and specifically, in applying thread-level Speculative Precomputation on four applications. For this multi-threaded execution scheme, our implementation provided up to 7% better performance compared to Pthreads, and up to 40% compared to spin-loops-based barriers.

1 Introduction

Simultaneous Multithreading (SMT) [8] allows a superscalar processor to issue instructions from multiple independent threads to its functional units, in a single cycle. The motivation behind this technique is to maximize the utilization of processor resources by exploiting the thread-level parallelism that can be extracted from an application. Hyper-threading technology [6] 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, due to their simple implementation and the high responsiveness, as a result of their operation entirely at user-space. In an SMT environment, such primitives can incur significant performance penalty, especially when application execution entails situations where one or more threads should wait on synchronization events for a long period. In modern out-of-order processors, a spin-loop, which typically comprises of a memory value test and a subsequent branch, is dynamically unrolled multiple times by the scheduling units of the processor, because the branch can be easily predicted and no data dependences exist. In this way, the spinning thread, even though not performing any useful work, inserts a significant number of instructions in the pipeline that compete with the peer thread for execution resources. Additionally, upon update of the spin-loop variable, all speculatively issued instructions of the spinning thread that have not yet committed, must be discarded, which incurs a costly pipeline flush penalty.

Regarding SMT, Tullsen et al have proposed hardware extensions to support fine-grained synchronization, while addressing the aforementioned performance issues [9]. In their simulated SMT model, the processor provides directly explicit acquire and release primitives. It is extended with a small structure, the *lock-box*, with one entry per context, containing the address of the lock, the address of the locking instruction and a valid bit. When a thread fails to acquire a lock, the address of the lock and the address of the locking instruction are stored in that thread's entry. The thread then blocks and

*This research is supported by the PENED 2003 Project (EPAN), co-funded by the European Social Fund (75%) and National Resources (25%).

is flushed from the processor. When a thread releases a lock, hardware performs an associative comparison of the released lock against all lock-box entries. On finding a thread blocked on that lock, it is awakened and restarted from the locking instruction. This mechanism enables transfer of lock state between threads in very few cycles, and furthermore eliminates resource waste.

A hyper-threaded processor does not provide such an explicit mechanism for low-latency, resource-friendly synchronization directly at the hardware level. There are other techniques that could be employed in order to mitigate the excessive resource consumption of spin loops, which, however, cannot approach their high responsiveness. First, we could use primitives that resort to OS mediation to handle long condition-wait periods. In this way, the waiting thread would yield its logical processor and release all its resources (transition from Multi-Threaded to Single-Threaded mode), but its notification and resumption upon condition satisfaction would be expensive in terms of cycles, due to the invocation of the scheduler.

A second solution would be to loosen the spinning. The rationale behind that is that the waiting thread actually spins much faster than required, i.e. than the time needed by the memory bus to perform a single memory update. Intel recommends the use of PAUSE instruction for this purpose, which introduces a slight delay in the loop and de-pipelines its execution, preventing it from aggressively consuming processor resources. These are resources that are shared *dynamically* between hyper-threads (e.g., execution units, caches, fetch-decode-schedule-retirement logic). Although in this way the cost of spinning is reduced, it is not entirely eliminated, because some resources designed to be *statically* partitioned between the two hyper-threads (e.g., micro-op queues, load-store queues, re-order buffers), are not released. These are mostly intermediate buffering queues that guarantee independent flow of the instructions of the two threads through the pipeline. In Multi-Threaded mode, these queues are split so that each thread can use at most half of their entries. When a thread executes a PAUSE, it continues to occupy its share of entries. Thus, the spinning thread still holds a portion of resources that could be valuable for the peer thread to execute faster.

By using the privileged HALT instruction, a logical processor can relinquish all of its statically partitioned (and shared) resources, make them fully available to the other logical processor and stop its execution, going into a sleeping state. Later, as soon as it receives an inter-processor interrupt (IPI) from the active processor, it resumes its execution and the resources are partitioned again. Apart from requiring kernel privileges to enter the halt state and cause the processor to exit from it, which translate into system call overhead, the transitions themselves into and out of the halt state incur extra overhead in terms of processor cycles, as well.

Intel's Prescott core introduced a new pair of instructions, MONITOR and MWAIT [4]. MWAIT enables a logical processor to enter into an "implementation-dependent performance-optimized" state while waiting for a single store to the address range set up by MONITOR. In a hyper-threaded environment, all shared and partitioned resources of a logical processor are released with MWAIT, as with HALT. Like HALT, MONITOR/MWAIT must be executed at kernel-space, as well. Unlike HALT, however, which requires an expensive IPI delivery to awaken an idle context, MONITOR/MWAIT just require a single memory store for the same purpose.

So far, MONITOR/MWAIT instructions have been used mostly for inter-thread synchronization in operating systems code. In particular, they are used to implement the idle loop of the scheduler, where a processor waits on a memory location (the data structure that corresponds to its work queue) until it is notified that there is some task that needs to be scheduled. In this work, we explore the potential of using MONITOR/MWAIT instructions to synchronize application-level threads, which execute on hyper-threaded processors and are characterized by workload asymmetry. The rest of the paper is organized as follows: in Section 2 we give a description of MONITOR/MWAIT. In Section 3 we propose a framework through which one can use these privileged instructions to build condition-wait and notification primitives, with the least possible kernel involvement. In Section 4 we present a bottom-up evaluation of our primitives, starting from an analysis of certain performance aspects of their implementation that reflect the execution model we consider, and ending to an evaluation of MONITOR/MWAIT-based barriers in real-world applications. Finally, Section 5 summarizes our conclusions and discusses future work.

2 Description of MONITOR/MWAIT

The MONITOR instruction sets up an effective address range that is monitored by the executing logical processor for write-to-memory activities. MWAIT places the logical processor in a "performance-optimized" state (which may vary between different implementations) until a write to the monitored address range occurs. This implements a condition-wait as close as possible to the hardware level, with the advantage that it prevents resource waste on a hyper-threaded processor and the notification of the waiting thread does not require operating system intervention but a single memory value update.

In particular, MONITOR sets up monitoring hardware to detect stores to an address range (generally a cache line) determined by the contents of EAX register. The MONITOR instruction relies on a state in the processor called the monitor event pending flag. Execution of the MONITOR instruction arms the monitoring hardware and clears the flag. A write to the address range being monitored, as well as other events such as interrupts, will trigger the monitoring hardware and set the flag. The state of monitoring hardware is not architecturally visible except through the behavior of the MWAIT instruction.

MWAIT puts the processor into the special low-power/optimized state until a store to any byte in the address range being monitored is detected, or if there is an interrupt or exception that needs to be serviced. MWAIT is architecturally identical to a nop instruction. It is effectively a hint to the processor to indicate that it may choose to enter an

implementation-dependent optimized state while waiting for an event or for a store to the address range set up by the preceding MONITOR instruction. On a hyper-threaded processor, a thread that calls MWAIT causes its logical processor to relinquish all its shared and partitioned resources and go to sleep.

As we mentioned, exits from the MWAIT state could be due to a condition other than a write to the triggering address. Software should explicitly check the current value of the triggering address against an original value, in order to determine if the exit from the MWAIT was due to a write to the monitored region or due to other event. If the exit was not due to a write then MWAIT must be executed again. However, MWAIT does not re-arm automatically the monitoring hardware, and thus MONITOR has to be executed again, as well. In other words, MONITOR/MWAIT need to be executed in the same loop.

The address provided to MONITOR instruction effectively defines an address range, within which a store will cause the exit from the optimized state. Write operations intended to cause the exit from this state must occur within this range. Thus, in order to avoid *missed wake-ups*, the data structure used to monitor stores must fit within the *smallest monitor line size*, and must be properly aligned so that it does not cross this boundary. Otherwise, the processor may not wake up after a write intended to trigger an exit from MWAIT. Similarly, write operations not intended to cause exit from the optimized state should not write to any location within the monitored range. Thus, in order to avoid *false wake-ups*, we should probably pad the data structure used to monitor writes to the *largest monitor line size*. This would preclude allocation of unrelated data structures within the monitored range. In our system, the smallest and largest monitor line sizes are both 64 bytes.

3 Framework for Implementing Synchronization Primitives

In their initial implementation, MONITOR and MWAIT instructions are available at privilege level 0 only. For this reason, in order to be able to implement a condition-wait primitive to be used at the application level, we had to extend the Linux kernel with a system call through which we can have access to these instructions. The overhead of this system call is the least that should be paid by any user-level application that wants to use MONITOR and MWAIT. 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 other words, we had to decide where to allocate the monitored memory region. If it is allocated at user-space, the kernel has to access the process address space every time it must check the contents of the monitored memory. This would require copying the monitored memory value to a kernel-space buffer (with a function like `copy_from_user`) within the MONITOR/MWAIT loop. If it is allocated at kernel-space, then we should add an additional system call through which a process can change its value. Consequently, both cases would incur notable extra overhead, either at condition-wait, due to multiple copy operations of the contents of the triggering address, or at notification, due to system call overhead.

In order to establish the fastest possible transfer of state between kernel and user-space, we followed an alternate approach: we chose to allocate the monitored memory region in kernel-space, in the context of a special character device, and then map the device to user-space. In this way, the memory region can be directly accessed both from kernel and user-space, without any copy operations or additional system calls required to access its contents. This mapping is depicted in Fig. 1.

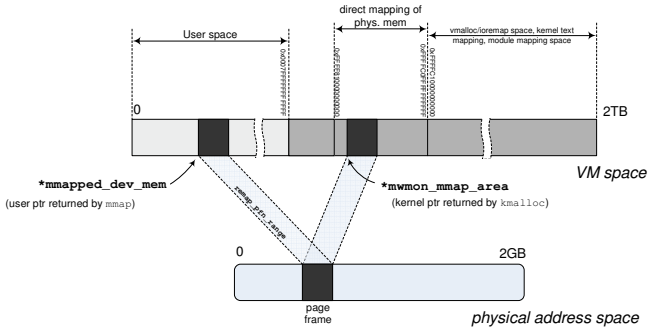


Figure 1. Mapping the monitored memory region allocated at kernel-space to process address space.

The driver for the special character device (*kmem_mapper*) is implemented as a loadable module. The monitored memory region is allocated when loading the module. Specifically, at module's initialization function, we call `kmalloc` to allocate 4096 bytes physically contiguous (effectively, a page frame in physical memory). The value returned by `kmalloc` initializes a pointer in kernel virtual address space that designates the start of the 64-byte sized monitored memory region (`*mwmon_mmap_area`). Of course, we take care so that this region is properly aligned to the smallest monitor line size boundary, as discussed in section 2. Furthermore, to prevent the containing page from being swapped to disk, we set the `PG_reserved` flag of the corresponding page frame. The `mwmon_mmap_area` pointer is hard-coded in the kernel's source and exported as a kernel symbol, and as we will see, provides the handle for the system call that implements the condition-wait to access the monitored memory region.

We implement three methods for kmem_mapper: open, mmap and close. In open and close methods, all bytes of the monitored region are initialized to an original, "unnotified" value (`MWMON_ORIGINAL_VAL`). Mmap method does the whole work: it calls `remap_pfn_range`, with `mwmon_mmap_area` among the arguments, to remap the driver allocated buffer (containing the monitored memory region) to user-space, when the user program mmap's kmem_mapper. The address returned by mmap will point to the start of the monitored memory region, exactly as `mwmon_mmap_area` does in kernel side. At module's cleanup function, the `PG_reserved` bit of the allocated page is cleared and the page is `kfree`'d.

After establishing fast communication between kernel and user-space, the implementation of a system call for condition-wait using MONITOR and MWAIT is straightforward. We extended the Linux kernel with the `mwmon_mmap_sleep` system call, as shown in Listing 1.

Listing 1. System call for implementing condition-wait using MONITOR/MWAIT.

```
/* Pointer to memory allocated by device driver */
char *mwmon_mmap_area;
EXPORT_SYMBOL_GPL(mwmon_mmap_area);

#define MWMON_ORIGINAL_VAL 'a'
#define MWMON_NOTIFIED_VAL 'b'
asmlinkage long sys_mwmon_mmap_sleep(void)
{
   do {
      local_irq_disable();
      monitor(mwmon_mmap_area,0,0);
      local_irq_enable();
      if(*mwmon_mmap_area == MWMON_NOTIFIED_VAL)
         break;
      mwait(0,0);
   } while (*mwmon_mmap_area != MWMON_NOTIFIED_VAL);
   *mwmon_mmap_area = MWMON_ORIGINAL_VAL;

   return 0;
}
```

After having loaded the module, the steps that should be followed in a typical multithreaded program in order to use the synchronization primitives based on MONITOR/MWAIT instructions, are the following: At initialization phase, the program should open *kmem_mapper* for read and write, and then mmap it in its address space (again, with read and write protection flags set). A thread that wishes to wait on a condition, calls `mwmon_mmap_sleep` (without any argument). A thread that wishes to notify the waiting thread, sets a random byte within the monitored memory region to `MWMON_NOTIFIED_VAL` value. The monitored memory region starts from the address returned by mmap, and ends after 64 bytes. At finalization, the program unmaps and then closes the device.

4 Performance Evaluation

We evaluate the efficiency of our synchronization primitives in a bottom-up manner: First, we compare our primitives with other possible implementations, by measuring directly various aspects of their performance such as resource consumption and responsiveness. As a second step, we use our primitives to build synchronization barriers. We quantify the same performance issues in this case, as well, and furthermore, we measure overall barrier efficiency through a manually constructed pseudo-benchmark. Finally, we evaluate our barrier implementation in real-world applications, in the context of applying thread-level speculative precomputation on them.

In every step of the evaluation process, we have considered an application model where two threads are executing on the two contexts of a hyper-threaded processor, with each thread occupying persistently a specific context throughout execution. These threads are characterized as *asymmetric*, mainly due to the amount of work that each thread executes, rather than the kind of work of each (e.g. floating-point and integer computations).Therefore, the common case scenario we examine involves one *heavyweight* thread (also referred as *worker*) performing computations throughout its entire execution, and one *lightweight* thread (also referred as *helper* or *waiter*) 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 (e.g., through a condition variable or a barrier). An example of this execution scenario is depicted in Fig. 2a. In this case, the two threads are synchronized with barriers. In each phase, helper performs only a small amount of work which is intended to facilitate the execution of worker at the next phase. Thus, worker is always busy while helper is periodically throttled. In real applications, the lightweight thread could be a helper thread that facilitates the execution of the main thread, e.g. by prefetching data from main memory into a shared cache, by performing network I/O and message processing in distributed memory parallel applications, by performing disk request completions in I/O-bound applications, etc.

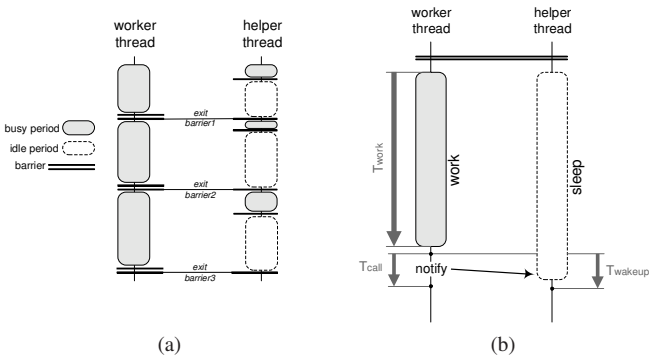


Figure 2. (a) Application model under consideration. (b) Execution scenario for the evaluation of condition-wait and notification primitives.

This discussion reveals a number of requirements that must be met in order the synchronization primitives to be effective for this execution model. First, the helper thread must not introduce significant impediment to the worker thread whenever it waits on synchronization events, by consuming shared resources. Second, the helper thread must resume as fast as possible from its sleep each time it is notified by the worker thread, in order its actions to be timely and accurate. This requirement becomes more important as the need for more fine-grained orchestration of the helper thread's actions increases. 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. Again, this is substantial in cases of frequent synchronization between threads. Summarizing, our primitives must incur *low resource consumption*, *high responsiveness* and *low call overhead*. Normally, these requirements are conflicting, e.g. fast propagation of state changes requires repetitive checks of memory values, and thus can consume measurable processing and memory bandwidth. In this study, we look for an option that best balances these requirements.

4.1 Experimental setup

We experimented on an Intel Xeon processor, running at 2.8GHz. This processor is based on Netburst microarchitecture, and is one of the first mainstream chips to encompass low-end simultaneous multithreading capabilities. The operating system was Linux v. 2.6.13 for the x86_64 ISA. We used the NPTL library for the creation and manipulation of threads. To force the threads to be scheduled on a particular processor, we used the `sched_setaffinity` system call. All user codes were compiled with gcc v. 4.1.2 using the O2 optimization level, and linked against glibc v. 2.5.

4.2 Evaluation of performance characteristics of synchronization primitives

Initially, we have considered an execution scenario like this presented in Fig. 2b. There are two threads, each scheduled on a specific context of a hyper-threaded processor. The heavyweight thread performs a fixed amount of floating-point work (a 512×512 matrix multiplication), while the lightweight just waits until notified by the former when it finishes its computations. For each version of synchronization primitives we examine, we measure the following:

- T_{work}: the time required for the worker to complete its computations. The larger this time is, the more disturbing is the co-existence of the waiting thread on the peer context.
- T_{wakeup}: the time between the notification of the waiting thread and the moment that it is actually awakened. This is a direct indication of the responsiveness of the wait primitive.
- T_{call}: the time that the worker thread spends in invoking the appropriate notification primitive.

We evaluate and compare the following options for the primitives for condition-wait and notification of threads: our proposed implementation with MWAIT/MONITOR instructions (referred to as *mwmon*), spin-wait loops that use the PAUSE instruction (*spin-loops*), spin-wait loops that use the HALT instruction (*spin-loops-halt*), and the synchronization primitives of the NPTL library (*pthreads*).

In *mwmon*, the waiter calls `mwmon_mmap_sleep` to go to sleep, and it is awakened when the main thread updates the monitored memory region. This process has been already described in section 3. In *spin-loops* version, the waiting thread spins repeatedly on a user-level variable, until the main thread updates its value. In order to make the spin loop less aggressive than it is required (as explained in section 1), we embedded the PAUSE instruction in the loop body.

The *spin-loops-halt* implementation is quite the same, with the difference that the waiting thread invokes the HALT instruction instead of PAUSE, and upon notification, the worker thread sends additionally an inter-processor interrupt after updating the user-level variable, to awaken the halted thread. In this way, the waiting thread puts its processor into a suspended state, offering all of its resources for exclusive use by the peer thread. It may be awakened periodically by IPIs sent by the OS (e.g. timer interrupts), but will exit the loop only when it notices the change in the variable's value. To be able to use the privileged instruction HALT and send IPIs from userspace, we extended the Linux kernel with appropriate system calls.

The fourth alternative we examined was the native implementation of condition variables in the Pthreads library. The latest version of NPTL library we used, makes use of *futexes*, a mechanism provided by the Linux kernel as a building block for fast userspace locking. Further details on the internals of the mechanism are out of our scope, however a good discussion is done in [3]. In this implementation, a thread that waits on a condition variable (via `pthread_cond_wait`) makes a `futex` system call with a `FUTEX_WAIT` argument, which causes the thread to be suspended in the kernel. The thread is actually descheduled, and assuming that there are not other runnable processes, all its resources are released and made available to the other thread (i.e. the processor switches from Multi-threading to Single-threading mode). When the worker notifies the blocked thread (via `pthread_cond_signal`), it actually calls `futex` with a `FUTEX_WAKE` argument to awaken and reschedule the waiter.

Primitive	Work time (in seconds)	Wakeup latency (in cycles)	Call latency (in cycles)
spin-loops	4.2446 ($\pm$0.2987)	584 ($\pm$191)	785 ($\pm$223)
spin-loops-halt	3.6243 ($\pm$0.3036)	37470 ($\pm$3975)	27813 ($\pm$3768)
pthreads	3.5919 ($\pm$0.2367)	116989 ($\pm$5795)	70042 ($\pm$3569)
mwmon	3.4821 ($\pm$0.2996)	25381 ($\pm$2426)	5470 ($\pm$545)

Table 1. Performance of various synchronization primitives for condition-wait and notification of threads (see Fig. 2b).

Table 1 presents the results from the evaluation of each implementation on the execution scenario presented in Fig. 2b. We have repeated each experiment multiple times (50), and we present the average values from all measurements. As expected, *spin-loops* provide minimum response time and call overhead, since it does not invoke any OS intervention, but it is the most aggressive in terms of resource consumption. On average, the spinning thread decelerates worker thread 19% more than the other three implementations. However, decreased interference of the waiting thread in these implementations comes at the cost of two or three orders of magnitude larger wakeup and call latencies. In particular, *pthreads* suffer the worst wakeup and call times, possibly due to the largest kernel control paths for notifying and rescheduling the waiting thread. In *mwmon*, the waiting thread resumes 47% faster compared to *spin-loops-halt*, and has almost 5 times less call overhead. This implies that the "performance-optimized" sleeping state of MWAIT is probably more responsive than the state of HALT, yet equally resource-conserving. Obviously, sending IPIs through system calls to awaken the waiting thread is far more expensive than a single variable update in user-space. Regarding work time, the last three implementations perform equally well, with *mwmon* performing slightly better. Overall, *mwmon* is the option that best balances low resource consumption, high responsiveness and reduced call overhead.

4.3 Evaluation of barrier implementations

Using our primitives for condition-wait and notification with MONITOR/MWAIT, we were able to build synchronization barriers. These barriers are intended to be used in the execution scenario presented in the beginning of this section, i.e. by a pair of threads executing on hyper-threads. They lack the generality of other implementations, posing limitations regarding the number of threads that can synchronize on a single barrier or the number of barrier variables that can be in use at any time during application execution, but they can be extended in a straightforward manner in order to overcome these shortcomings (e.g., associating multiple monitored memory regions with different barrier variables, using per-physical package local *mwmon*-like barriers to build global, hierarchical barriers, etc.). In this work, our interest concentrates mainly on the efficiency of our implementation on the execution model we consider, rather than the generality of use.

The basic points of our barrier implementation (henceforth called *mwmon*) are presented in Listing 2. Each thread that enters the barrier atomically increases a global counter using a spin-lock. The waiter thread makes the barrier available (releases the lock) before suspending, and it is awakened when the last thread updates the monitored memory region. In order to make the barrier reusable and avoid deadlocks, e.g. in case it is used within a loop, we count departure of threads apart from counting their arrival, using atomic decrement operations. The barrier is made available for subsequent uses only by the last thread that leaves it. In this way, a thread cannot enter a barrier if there is another thread that has not yet departed from the previous instance of this barrier.

Listing 2. Sample barrier implementation with MWAIT/MONITOR-based primitives.

```
void mwait_barrier(mwait_barrier_t *barrier)
{
   unsigned int total = barrier->total;
   spin_lock(&barrier->lock);
   ++barrier->gathered;

   if (barrier->gathered == total) { /*last thread*/
      *mmapped_device_memory = MWMON_NOTIFIED_VAL;
   } else { /*intermediate thread*/
       spin_unlock(&barrier->lock);
       mwmon_mmap_sleep();
   }
/*Is this the last woken thread? If yes, then unlock.*/
   if (atomic_dec_and_test(&barrier->gathered))
      spin_unlock(&barrier->lock);
}
```

The rest implementations we tested include a version of barriers with sense-reversing [7] which uses spin-wait loops with the PAUSE instruction (*spin-loops*), a similar version which uses the HALT instruction along with the accompanied mechanisms to send IPIs (*spin-loops-halt*), and the native implementation offered by the NPTL library (*pthreads*). In these versions, the actions performed by the intermediate and the last thread entering the barrier, are quite the same to those performed by the waiting and worker thread, respectively, in the execution scenario we examined in section 4.2.

At first, we evaluate all implementations on a simple execution scheme like this presented in Fig. 3a. Again, we consider a heavyweight and a lightweight thread scheduled on different contexts of a hyper-threaded processor. The heavyweight performs the same amount of floating-point work as in the previous scenario, while the lightweight does nothing. The threads are synchronized at the end with a barrier. The helper enters immediately the barrier, waiting to be notified by the worker when it finishes its computations and enters its barrier. For each implementation, we measure the time required for the worker to complete its computations (T_{work}), the time between the arrival of the worker on the barrier and the departure of the helper from it (T_{wakeup}), and the time that the worker spends in its barrier (T_{call}).

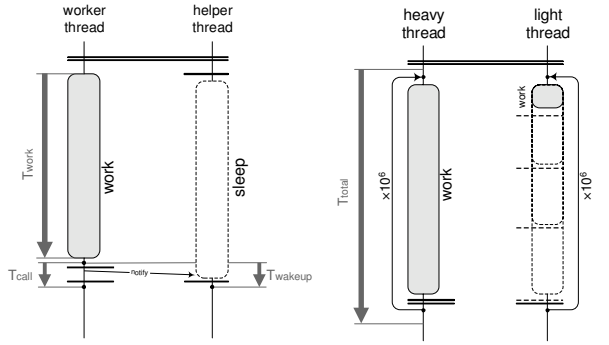


Figure 3. Execution scenario (a) and pseudo-benchmark (b) for the evaluation of synchronization barriers.

Barrier implementation	Work time (in seconds)	Wakeup latency (in cycles)	Call latency (in cycles)
spin-loops	4.3897 ($\pm$0.3461)	1236 ($\pm$340)	1173 ($\pm$338)
spin-loops-halt	3.5720 ($\pm$0.2624)	49953 ($\pm$11502)	51329 ($\pm$11879)
pthreads	3.5917 ($\pm$0.2345)	45035 ($\pm$3608)	18968 ($\pm$1343)
mwmon	3.5266 ($\pm$0.2549)	11319 ($\pm$1770)	5470 ($\pm$644)

Table 2. Performance of various barrier implementations for the scenario of Fig. 3a.

Table 2 presents the results from the evaluation of the various barrier versions. Again, *mwmon* offers the best combination of low resource consumption, high responsiveness and low call overhead. With respect to *spin-loops*, the waiting thread in *mwmon* introduces 24% less interference to the main thread. Compared to *pthreads*, which is the best option among the three most resource-friendly versions, it has almost 4 times lower wakeup latency and 3.46 times reduced call overhead.

As a next step, we evaluate barriers efficiency in fine-grained synchronization of threads with varying workload asymmetry. For this purpose, we have constructed a pseudo-benchmark, again with two threads executing on different hyper-threads of the same processor (see Fig. 3b). The unit of work in this case is considered a 10×10 floating-point matrix multiplication. Within a loop, the first thread ("heavy")

executes always the maximum amount of work, which is 10 units of work. The second thread ("light"), executes a smaller amount of work within a same loop, which ranges between 0 and 10 units of work. Both loops iterate for 10^6 times, and between successive iterations threads are synchronized with barriers. For all different degrees of light thread's workload, we measure the overall completion time of the benchmark using various barrier implementations. To a large extent, this time reflects the throughput of each barrier version, considering the relatively short work executed in each loop iteration. From this evaluation process we have omitted the *spin-loops-halt* implementation, since it yielded extremely large completion times for this number of loop iterations, probably due to its large call and wakeup overhead.

Results are presented in Fig. 4. For all levels of threads asymmetry, the *mwmon* implementation outperformed all other versions by a considerable factor. On average, it yielded 12% better throughput compared to *pthreads*, and 26% better throughput compared to *spin-loops*. As expected, in the case of complete asymmetry (light thread workload=0), *mwmon* enjoys best speedups (1.22 with respect to *pthreads* and 1.62 with respect to *spin-loops*), since this case involves the largest wait periods in the light thread's execution, which can best advocate the *mwmon*'s ability to conserve execution resources efficiently. As threads profiles converge to absolute symmetry (light thread workload=10), the resource-conserving characteristics of each implementation become less important, and what matters more is the call overhead and the wakeup latency. Again, *mwmon* performs slightly better compared to *spin-loops* (by a factor of 1.02), and quite better compared to *pthreads* (by a factor of 1.09), due to the larger performance gap of *pthreads* with respect to call overhead and wakeup latency (see Table 2).

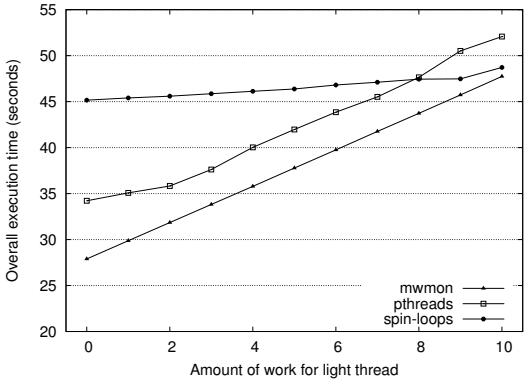


Figure 4. Throughput of various barrier implementations for different levels of threads asymmetry (see Fig. 3b).

4.4 Evaluating barriers performance for speculative precomputation

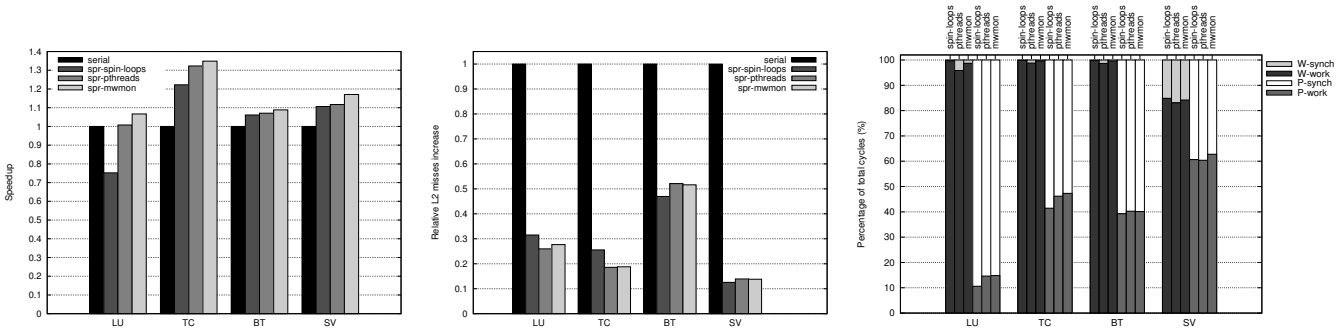
The last step in our evaluation process involves testing the various barrier implementations for applying *Speculative Precomputation* (SPR) on a series of real-world applications. SPR is a technique that targets at utilizing the multiple contexts of a processor in order to accelerate a sequential application. In SPR, the execution of an application is facilitated by additional helper threads, which speculatively prefetch data that are going to be used by the main computation thread in the near future, thus hiding memory latency and reducing cache misses [2, 5].

In our work, we create at the beginning of the program a single helper thread, that persistently occupies a specific hardware context throughout execution. The worker executes on the peer context in the same package. The 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 between threads can be implemented with barriers, following an execution scheme like this presented in Fig. 2a. Further elaboration on issues such as generating code for the prefetcher or selecting proper points to insert barriers, is out of our scope. The reader is referred to [1] for a thorough discussion on these issues. What is of main interest, is the barrier implementation itself, because operational characteristics such as call and wakeup latency or resource consumption, determine key issues for the efficiency of SPR, such as the miss coverage ability of the prefetcher and extra overhead posed to the worker.

We used four benchmarks to evaluate SPR performance using different barrier implementations: LU decomposition (2048×2048, 10×10 blocks), transitive closure computation of a directed graph in dense representation (1600 vertices, 25000 edges, 16×16 blocks), the BT benchmark from NAS Benchmark suite v. 2.3 (Class A), and a sparse matrix-vector multiplication kernel (9648×77137, 260785 non-zeroes).

Fig. 5a presents the attained speedups of SPR using the *spin-loops*, *pthreads* and *mwmon* barrier implementations. Fig. 5b shows the numbers of L2 misses of the worker thread normalized with respect to serial execution, which is a direct indication of the miss coverage ability of each SPR version. The counts were gathered using performance counters. Furthermore, we measured for each application the fraction of cycles during which the worker and prefetcher threads perform useful computations or wait on synchronization barriers. This cycle breakdown is presented in Fig. 5c. In this diagram, periods of work and wait are denoted as *work* and *synch*, respectively. We expect that best SPR performance can be achieved under combined occurrence of good miss coverage, uninterrupted work of the worker (minimal time spent in barriers), and short workload of the prefetcher, which introduces the least possible contention for shared resources, and entails long idle periods where the benefit from resource-friendly barriers can be maximized. Of course, the inherent potential of an application to benefit from perfect cache locality plays always a significant role.

At a first glance, SPR with *mwmon* barriers outperformed all other versions and boosted application performance offering speedups between 1.07 (LU) and 1.35 (TC). Its average speedup across all applications was 1.17. On average, *mwmon*



(a) Speedup (b) Miss coverage (c) Cycles breakdown

Figure 5. Experimental results for SPR with different barrier implementations.

yielded 15% better execution times compared to *spin-loops*, and 3.6% compared to *pthreads*. TC benefits the most from SPR, probably because it enjoys great L2 miss reductions, having originally bad cache locality. Likewise, despite its initial bad locality, BT gets limited performance gains from SPR, possibly due to insufficient miss coverage. SV and LU are two benchmarks where SPR with *mwmon* adds notable performance boost compared to the rest versions. LU, due to its already good locality, seems to be more sensitive to the interference introduced by the prefetcher than the L2 miss reduction achieved, and *mwmon* seems to be the option that best satisfies this requirement. This assertion is also corroborated by the fact that *spin-loops* implementation affects performance of LU negatively. In SV, on the other hand, not only is the execution of the worker significantly interfered due to the relatively large workload of the prefetcher, but the worker is also delayed on synchronization events for a non-trivial portion of its execution time. Nevertheless, the prominent reduction of L2 misses, along with any resource-conserving offerings by *mwmon*, make *mwmon* the best choice in this case, as well.

5 Conclusions and Future Work

In this paper we explored the possibility of using the MONITOR/MWAIT instructions for synchronization of application threads with asymmetric workloads, executing on hyper-threaded processors. We presented a framework through which one can use these privileged instructions efficiently, and based on this infrastructure we built basic condition-wait and notification primitives, as well as synchronization barriers. Compared to other spin-loops-based versions and Pthreads, our implementations proved to be quite efficient for the application model we consider, making the best compromise between low resource waste and low call and wakeup latency. Noteworthy performance gains were achieved when we tested our barriers for fine-grained synchronization of threads and speculative precomputation.

As a future work, we intend to extend our framework in order to support multiple MONITOR/MWAIT-based synchronization variables within the same application, and to support synchronization in multi-SMT systems. 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

[1] E. Athanasaki, N. Anastopoulos, K. Kourtis, and N. Koziris. Exploring the Performance Limits of Simultaneous Multithreading for Memory Intensive Applications. *The Journal of Supercomputing*, 2007. (to appear).

[2] J. Collins, H. Wang, D. Tullsen, C. Hughes, Y. Lee, D. Lavery, and J. Shen. Speculative Precomputation: Long-Range Prefetching of Delinquent Loads. In *Proc. of ISCA '01*, Göteborg, Sweden.

[3] U. Drepper. Futexes are Tricky. Dec 2005.

[4] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, vol. 3A: System Programming Guide, Part 1*, Aug 2007. Order Number: 253668-024US.

[5] D. Kim, S. Liao, P. Wang, J. Cuvillo, X. Tian, X. Zou, H. Wang, D. Yeung, M. Girkar, and J. Shen. Physical Experimentation with Prefetching Helper Threads on Intel's Hyper-Threaded Processors. In *Proc. of CGO 2004*, San Jose, CA.

[6] D. Marr, F. Binns, D. Hill, G. Hinton, D. Koufaty, J. Miller, and M. Upton. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal*, 6(1), Feb 2002.

[7] D. Patterson and J. Hennessy. *Computer Architecture. A Quantitative Approach*. Morgan Kaufmann, 3rd edition, 2003.

[8] D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proc. of ISCA '95*, Santa Margherita, Italy.

[9] D. Tullsen, J. Lo, S. Eggers, and H. Levy. Supporting Fine-Grained Synchronization on a Simultaneous Multithreading Processor. In *Proc. of HPCA '99*, Washington, DC.