An Infrastructure for Efficient Synchronization of Asymmetric Threads on Hyperthreaded Processors

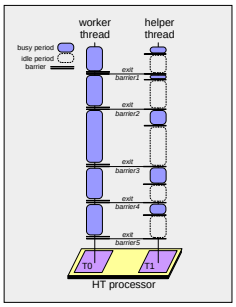
Nikos Anastopoulos, Nectarios Koziris

{anastop,nkoziris}@cslab.ece.ntua.gr

Computing Systems Laboratory, School of Electrical and Computer Engineering, National Technical University of Athens

Motivation

The privileged instructions MONITOR/MWAIT introduced with Intel Prescott core, offer a "performance-optimized" way for threads involved in synchronization events to wait on a condition.
So far, they have been used mostly for inter-thread synchronization in operating systems code.
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.



The application model under consideration: a worker and a helper thread executing in parallel on a hyper-threaded processor. In each phase, helper performs only a small amount of work which is intended to facilitate worker in the next phase, and then waits until it is notified by worker before proceeding.
In real-world applications, the helper could perform e.g. speculative precomputation (prefetching of future memory accesses), network I/O and message processing in distributed memory applications, etc.

How should synchronization be implemented?

- low resource consumption (idle periods may dominate execution time of helper)
- worker: fast notification of helper (low call overhead)
- helper: fast resumption (high responsiveness)

Options for synchronization

- **spin-wait loops:** they provide high responsiveness, but consume significant resources. Even if we loosen the spinning of helper using PAUSE, the worker still experiences notable interference (15-20% on average). The reason is that not all resources are released (i.e. the statically partitioned ones).
- **spin-wait loops w/ HALT:** the spinning thread halts, and partitioned resources are recombined for full use by the worker. Executing HALT and sending inter-processor interrupts to wake up the sleeping thread require kernel privileges, which translate into system call overhead.
- **explicit processor yield:** e.g., through Pthreads condition-wait primitives. The helper is de-scheduled and suspended in the kernel, and the processor switches to single-threaded mode (all resources available for worker). OS scheduler intervention incurs high notification and resumption latency.
- **MONITOR/MWAIT loops:** MONITOR arms special hardware to monitor an address range for writes, MWAIT causes the calling logical processor to enter an "optimized " state until a write to the specified address range (or interrupt, exception, fault) occurs.

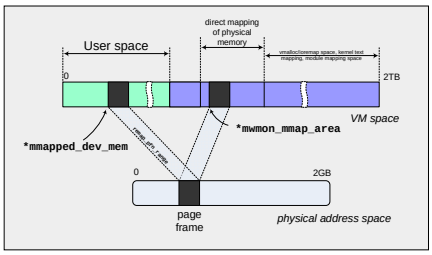
```
while (spinvar!=NOTIFIED){
    MONITOR(spinvar,0,0)
    MWAIT
}
```

- condition-wait close to the hardware
- all resources (shared & partitioned) relinquished
- require kernel privileges, **but**
- obviate the need for expensive IPIs delivery for notification (a single memory update suffices).

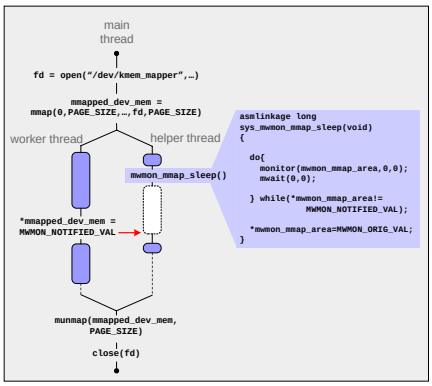
Implementing basic primitives with MONITOR/MWAIT

Where to allocate the region to be monitored?

- in user-space: fast notification, but requires copying the contents of monitored region to kernel-space on each condition check
- in kernel-space: requires additional system call to enable update of monitored memory from user-space
- in kernel-space, but map it to user-space for direct access



Establishing fast data exchange between kernel-space and user-space: a page frame to host the monitored memory is kmalloc'ed in kernel-space, in the context of a special char device ("kmem_mapper"). When the device is mmap'ed in the user program, the page is remapped to user-space and can be directly accessed. The "mwmon_mmap_area" pointer in kernel-space, and the "mmapped_dev_mem" pointer in user-space, both point to the beginning of the monitored region.



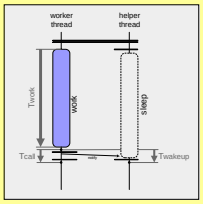
System call interface and use example of our MONITOR/MWAIT-based condition-wait and notification primitives in a multithreaded program.

Experimental evaluation

System configuration:

- Intel Xeon@2.8GHz, 2 hyper-threads
- Linux 2.6.13, gcc-4.12, glibc-2.5

Case 1: Barriers – raw performance



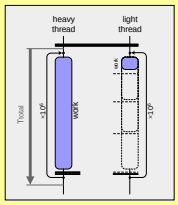
The worker performs a fixed amount of work, which is a 512x512 fp matrix multiplication. The helper does nothing and waits until it is notified, when the worker enters its barrier.

- T_{work}: reflects amount of interference introduced by helper
- T_{wakeup}: responsiveness of wait primitive
- T_{call}: call-overhead of notification primitive

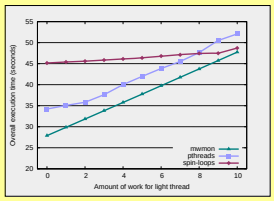
	Twork (seconds)	Twakeup (cycles)	Tcall (cycles)
	lower is better		
spin-loops	4.3897	1236	1173
spin-loops-halt	3.5720	49953	51329
pthreads	3.5917	45035	18968
mwmon	3.5266	11319	5470

"mwmon" best balances resource consumption and responsiveness/call overhead. It introduces 24% less interference compared to "spin-loops", and has 4× lower wakeup latency and 3.5× lower call overhead w.r.t. "pthreads".

Case 2: Barriers – fine-grained synchronization



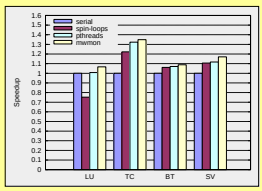
The unit of work is considered a 10x10 matrix multiplication. Within a loop, the worker performs always 10 units of work. The helper has a smaller workload, ranging between 0 and 10 units. Both loops iterate for 10^6 times, and successive iterations are synchronized with barriers. Considering the relatively short work executed in each iteration, the overall completion time reflects the throughput of each barrier implementation.



Completion times for the micro-benchmark of case 2. Across all levels of asymmetry, "mwmon" outperforms "pthreads" by 12% and "spin-loops" by 26%. As threads become symmetric, the resource-conserving characteristics of "mwmon" become less important and its performance converges with that of "spin-loops".

Case 3: Barriers – Speculative Precomputation (SPR)

SPR: thread-based prefetching of top L2 cache-missing loads. In each phase, the helper prefetches for the next phase, filling up to ½ of the L2 cache, and it is then throttled.
Good SPR performance can be achieved under good miss coverage from helper, and minimal synchronization overhead and resource contention for worker.



"mwmon" offers best speedups, between 1.07(LU) and 1.35(TC). With equal miss-coverage ability, it succeeds to boost "interference-sensitive" applications such as LU. In SV, it offers notable gains even though worker is delayed in barriers and prefetcher has relatively large workload.

Conclusions – Future work

MONITOR/MWAIT-based primitives make the best compromise between low resource waste and low call and wakeup latency for our considered model.

Possible directions of our work:

- "mwmon"-like hierarchical schemes in multi-SMT systems (e.g. tree barriers)
- other "producer-consumer" models (disk/network I/O applications, MPI programs, etc.)
- multithreaded applications with irregular parallelism

References

- *Facilitating Efficient Synchronization of Asymmetric Threads on Hyper-Threaded Processors*. N. Anastopoulos and N. Koziris. In 2nd Workshop on Multithreaded Architectures and Applications (MTAAP 2008), Miami, FL.
- *Exploring the Performance Limits of Simultaneous Multithreading for Memory Intensive Applications*. E. Athanasaki, N. Anastopoulos, K. Kourtis and N. Koziris. In The Journal of Supercomputing, Volume 44 , Issue 1 (April 2008).