Facilitating Efficient Synchronization of Asymmetric Threads on Hyper-Threaded Processors
A critique on James Coleman’s paper presentation in class

In this paper, they suggest using MONITOR/MWAIT instructions to synchronize application level threads, which execute on hyper-threaded processors and which are characterized by workload asymmetry. They presented a framework that makes use of these privileged instructions efficiently. A basic condition-wait and notification primitives were built based on their infrastructure, as well as synchronization barriers. They have evaluated using an application model where two threads are executing on a hyper-threaded processor. These threads are characterized as asymmetric, based on the amount of work that each thread executes (worker- performing computations throughout its entire execution, helper- execution alternates between short periods of useful work and long idle periods)

Strengths:

1. 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.
2. This technique establishes the fastest possible transfer of state between kernel and user-space.
3. In mwmon, the waiting thread resumes 47% faster compared to spin-loops-halt, and has almost 5 times less call overhead. This indicates that the “performance-optimized” sleeping state of MWAIT is better responsive than the state of HALT, yet also equally resource-conserving.
4. Overall, mwmon is the option that best balances low resource consumption, high responsiveness and reduced call overhead. It gives better throughput, prominent reduction of L2 misses, along with any resource-conserving offerings as against spin-loops, spin-loops-halt, pthreads.
5. The state of monitoring hardware is not architecturally visible except through the behavior of the MWAIT instruction.

Weaknesses:

1. The helper thread must be programmed such that it must not cause delay to the worker thread whenever it waits on synchronization events, by consuming shared resources 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 is crucial for more fine-grained orchestration of the helper thread’s actions increases.
2. The time that the main thread needs to invoke a synchronization primitive in order to notify the helper, must be made as little as possible. It increases drastically in cases of frequent synchronization between threads.
3. Summarizing the above constraints this technique must incur low resource consumption, high responsiveness and low call overhead.
4. Technique is valid only for x86 family of processors.
5. Exits from the MWAIT state could be due to a condition other than a write to the triggering address. Software must compare every time the current value of the triggering address with an original value, to know if the exit from the MWAIT was due to a write to the monitored region or due to other event. If it was not a valid write then MWAIT must be executed again. MONITOR has to be executed again because MWAIT does not re-arm on its own.
6. 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 write intended to trigger an exit from MWAIT may not wake up the processor. 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, extra padding of the data structure to the largest monitor line size is required which can be an unnecessary overhead.
7. In their system, the smallest and largest monitor line sizes were both 64 bytes. They could have studied the performance by changing these parameters.