Matthew Moua

Seza Habibi

**Kristof Du Bois, Stijn Eyerman, Jennifer B. Sartor, and Lieven Eeckhout. 2013. Criticality stacks: identifying critical threads in parallel programs using synchronization behavior. In Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA ’13). ACM, New York, NY, USA, 511-522. DOI:** <http://dx.doi.org/10.1145/2485922.2485966>

Today’s multi- core processor has to make sure to provide enough synchronize parallel work to ensure correct execution. Multi- threaded program has a challenge to reach the best performance while saving energy. There are several limitations to reaching this goal:

1. Barriers: a thread cannot pass through a point until all other threads reach that point
2. Critical Section: only one thread can access special data to avoid hazard of updating data by several threads.
3. Consumer- producer synchronization in pipeline program: a thread calculation can only do work when other thread produce data.

To reduce effects of these problems on performance they are Identifying critical threads. Critical threads are important because in the lower perspective they are causing other threads to wait for them and in the larger perspective they can improve performance optimization. Therefore, they are defining a concept of thread’s criticality that depends to the two characteristics: if thread doing useful work, if other threads are waiting for it. But choosing between a thread that do most useful work and a thread that most number of threads are waiting for it can be complicated, so they consider both running time and number of waiting thread in their critically metric.

Computing thread criticality is done in two steps: first, determining how many threads are performing useful work. In this step the challenge is identifying which thread are running, and It could be done using software or hardware. They choose a software solution because software can detect spinning(constantly detecting the synchronized variable by using a waiting loop) is more timely manner and it is easier to implement. Second step is calculating criticality of a thread. To do that they use a small hardware that keep track of running thread and criticality of those, so they used a 64-bit criticality counter and an active bit for each thread. Based on their calculation this piece of hardware consumes between 7.39 µW and 0.12 mW which is very small.

For implementing their program they used full-system simulation using gem5. They consider 8 and 16 core processor that running 8 and 16 threaded version in parallel. They used a criticality stacks for their parallel benchmark to summarize thread criticality. Experiment shows that for some benchmarks which have equal size of critically component, speed up of one thread doesn’t improve performance as they expected for other benchmarks which has one thread significantly larger than other, except one case(FMM) speeding up that treat, results in the largest speed up in the performance. They compare their new metrics with pervious one that was based on cache misses and they find their metric is most effective at finding the most critical thread.

After this experiment, they try to find out the influence of the change frequency in amount of speed up and they find out different benchmarks showing different behavior to increase of frequency. In some of them speed increase at first step and remain constant later although frequency constantly increase. Some of benchmarks getting more and more speed while frequency increased, and others act in between, remain constant at first and after some frequency the get speed up.

They describe three most use of critically stacks. The first one is getting software optimization by using critically stacks. They able to distinguish most critical thread causing by software and try to find solution to balance critical threads. This change speed up performance between 1.67 to 2.16 times. The second one is critical thread acceleration that dynamically accelerates a critical thread while using frequency scaling. It measures thread critical over a time and scale the most critical one in the next slice. using this approach, they improve performance by average of 4.6 % in compare with BIS for 8 core. the third one is saving energy, their experiment show when they are accelerating only the most critical thread they consume 11% less for BFS 16 core benchmark.