Static Micro-Scheduling: Resource Contention Relief in Multithreaded Programs

Yuanfang Chen, Xiaoming Li
Electrical and Computer Engineering Department
University of Delaware
Newark DE 19716, USA
Email: {cheny, xli}@udel.edu

Abstract— **Parallelism helps performance but at the same time stresses computer resources that are shared among threads. In this paper, we propose a low-overhead hardware counter based profiling method to accurately identify time-relevant contention locations in the program, then these contentions are mitigated so that performance of multithreading tasks can be boosted by the reduction of unnecessary contention cycles. In our preliminary experiment using NAS Parallel Benchmark (NPB), the contention searching algorithm is able to find an severe memory contention loop in FT code. After contention mitigation, more than 10% of the total cycles is eliminated, and the execution time of FT is reduced by 3% at the same time.**

Keywords-**profiling; compiler; hardware counter; contention; optimization;**

I. Introduction

All threads running on a multicore processor share resources such as the last-level cache, memory bandwidth, prefetcher etc.. It is not hard to envision that the resource sharing can cause contentions among threads. This resource contention problem has been addressed in prior researches such as [1]–[4] with the optimal scheduling methods. However little research has been done on contention mitigation from the perspective of code transformation within multithreaded programs. That is, prior works mostly accept the executables of programs as they are and try to react to resource pressures by optimizing process scheduling. However, the deeper problem is the interaction among threads that belong to one task, whose solution is critical to achieve the best overall performance for a multithreaded task that runs on a multicore architecture.

This paper makes three main contributions:

1) Resource Pressure Profiling. We developed a hardware counter based profiler to track several key hardware events that help identifying the most contentious parts in a multithreaded program in a portable way.
2) Identifying Resource Contentious Code and Type. Using information gathered from profiling, we developed a searching algorithm to back-project the contention spot to the program code precisely and help identifying the type of resources that threads compete for.
3) Contention-mitigating Transformations. We developed two prototype transformations to mitigate resource contentions. One purposefully reduces the number of concurrent threads on the most contentious program spot. The other converts the pressure on cache capacity to the utilization of memory bandwidth. These methods do not require programmer intervention, and can be standalone or be integrated into compilers as one of the feedback directed optimization.

The rest of this paper is organized as follows. Section II presents program metrics profiled and how profiling is performed with little overhead. Section III tells how we organize profiling output in a meaningful way and derive precise contention characteristics. Section IV presents experiment results. Finally, we draw conclusions in section V.

II. PMU Based Profiling

To make our method work with whichever parallel programming model or compiler is used, PMU profiling is the best choice in that it does not require code instrumentation. The first task is to find a set of desirable PMU events that help identifying contentions, at the same time, preserve portability across architectures. We argue that commonly available events “CPU Cycles” and “Instructions Retired” are these desirable events that would enable us to find every nontrivial contention spot in the program. “CPU Cycles” and “Instructions Retired” are portable in that almost all architectures define these two hardware events. They are effective in finding contention spots in that any contention caused execution inefficiency will be reflected in the increase of “CPU Cycles” for some sequence of instructions.

Our profiler runs a program twice. The first run is called “sequential” run, in which we deliberately leave exactly one core visible to the program, so that at any moment during the program execution, only one thread can have all the chip resource. The second run is called “parallel” run, in which the multithreaded program will run in the same environment as it does in production stage. Fig 1 shows the result of our profiling method testing on NPB benchmark. 19 executables in the benchmark are tested. Some executables share the same program code but with different problem sizes. In 13 of 19 examples, total cycles of the process are increased when the program executing in parallel; for the other 6 examples, total cycles of the process does not change much. Another observation is that total number of instructions retired does not change, which is within our expectation as the workload remains the same for both runs.

III. Contention Site Identification

To compare performance of same thread under different chip resource allocation setups, we need a base metric that makes the comparison valid and effective. We propose using “Instructions Retired” as the base metric, since it is very stable from run to run for most programs.

Our contention site identification algorithm works in six steps:

1) All samples are grouped according to their thread id.
2) For each thread id, samples are subgrouped according to their event number and sorted by timestamp.
3) For each event of each thread, hardware event values are changed from non-cumulative to cumulative.
4) Cycle interpolation is done for each “Instructions Retired” sample.
5) For “sequential” run, cycle interpolation is done for each “Instructions Retired” sample relative to the interpolated cycle from last step.
6) A summary of cycles difference contribution list is generated.

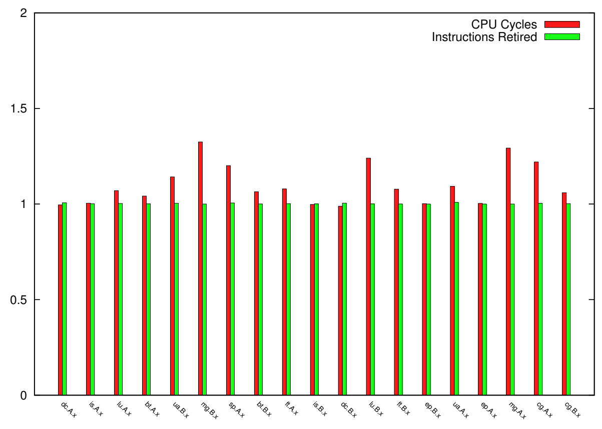

Fig. 1: Sample Output of the Profiling Tool

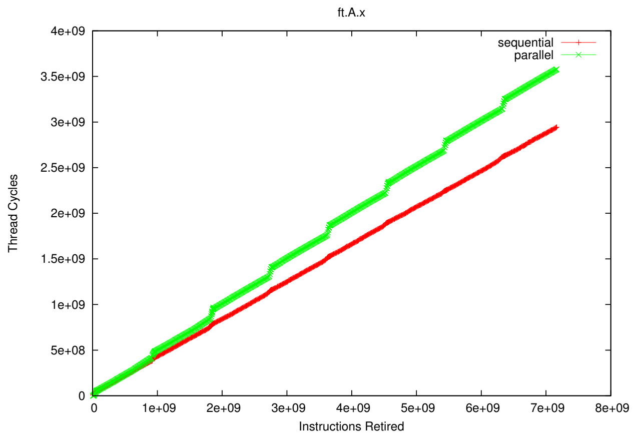

Fig. 2: FT Contention Overhead - Parallel vs. Sequential

IV. Experiment

First, we did a simple test to see how much contentions exist in NPB benchmark. The test result is shown in Fig 1. The X axis is the test programs in NPB benchmark. The Y axis is the ratio of hardware counter value of “parallel” run to that of “sequential” run. Event “Instructions Retired” is used to show that the number of instructions retired keeps almost the same between “parallel” run and “sequential” run. Event “CPU Cycles” indicates how much contention exists. The larger ratio of “CPU Cycles” is, the more contentions there are.

Next, we use NPB benchmark to test our contention spot identification algorithm. The test result of FT is shown in Fig 2. The X axis, event “Instructions Retired”, serves as the base on which “parallel” run and “sequential” run can be compared. Each point in X axis maps to two cycle values for each run respectively. A periodical development of cycle difference in Fig 2 reveals that a specific code location has caused the contention. To reduce the amount of contentions, only two threads, instead of four as specified in the original benchmark code, are needed to execute this section of code on our four-core machine, even though other parts of the program are still running with 4 threads. As shown in Fig 2, 10% of total cycles can be attributed to this contention. Those wasted cycles are spent on competing for memory bus, and no useful work is done with those cycles. This case shows that our contention identification algorithm is effective in identifying the contention hotspot.

V. Conclusion

In this paper, we propose a method to relieve the contention on shared resources in multithreaded programs and translate the reduced contention into higher overall performance. The method has three main components: a profiler to identify the most resource contentious spots, a micro scheduler that selectively reduces the number of concurrent threads in those contentious spots, and a transformation to shift the pressure on shared cache capacity to usage of memory bandwidth. This is a work in progress, and we verify the effectiveness of our approach with several programs in NPB benchmark and a matrix-multiplication based synthetic benchmark.

References

[1] S. Blagodurov, S. Zhuravlev, and A. Fedorova, “Contention-aware scheduling on multicore systems,” vol. 28, no. 4, pp. 8:1–8:45, Dec. 2010.

[2] J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa, “Contention aware execution: online contention detection and response,” in *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, ser. CGO ’10. New York, NY, USA: ACM, 2010, pp. 257–265.

[3] D. Chandra, F. Guo, S. Kim, and Y. Solihin, “Predicting inter-thread cache contention on a chip multi-processor architecture,” in *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, ser. HPCA ’05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 340–351.

[4] S. Zhuravlev, S. Blagodurov, and A. Fedorova, “Addressing shared resource contention in multicore processors via scheduling,” in *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ser. ASPLOS ’10. New York, NY, USA: ACM, 2010, pp. 129–142.