Dynamic Acceleration of Multithreaded Program Critical Paths in Near-Threshold Systems

Hyoun Kyu Cho Scott Mahlke
University of Michigan
{netforce,mahlke}@umich.edu

Abstract

Near-Threshold Computing (NTC) is an effective technique to improve energy efficiency. However, single thread performance can suffer dramatically in NTC systems as cores must be run at low frequency to ensure proper operation. A potential way to solve this problem is to accelerate a core for a short period of time using dynamic voltage and frequency scaling (DVFS). This fast-mode execution option must be selectively applied so as to not sacrifice the overall efficiency of the NTC system. To this end, this paper presents a novel software framework to improve the performance of multithreaded programs through smart scheduling of the fast mode cores. Our framework statically analyzes a target application and instruments dynamic monitoring and priority management code into the program. At runtime, the probabilistic scheduler assigns the cores to the fast mode according to the priority set by the instrumented code. In this way, the program critical path is dynamically accelerated by spending more time in the fast mode so that the overall performance gets improved.

1. Introduction

Power and energy efficiency has become the primary concern for the design of recent computer systems. While Moore's law keeps providing more transistors, power and thermal constraints will limit the number of cores that can be simultaneously turned on as well as the clock frequency. In order to continue delivering scalable computing performance, dramatic improvements in computational energy efficiency are necessary.

One of the most promising solutions to reduce energy consumption is voltage scaling. In particular, Near-Threshold Computing (NTC) [4] lowers the supply voltage to a value approximately equal to the threshold voltage of the transistors. NTC is expected to provide 10x or higher energy efficiency by reducing both dynamic and static energy superlinearly, so that it can enable many more cores to be powered on than conventional Super-Threshold Computing (STC).

Although NTC gives promising energy-frequency trade-offs, it also faces several major challenges. Among them is increased performance variation. As the dependencies of MOSFET drive current on the threshold voltage and the supply voltage becomes very steep in the near-threshold regime, the performance of NTC is very sensitive to process variation. The conventional approach for performance variation, adding margin so that all chips meet the specification in the worst case, is not effective for NTC and it imposes a daunting challenge to the chip designers [4, 11, 7].

One solution for the performance variation problem of chip multiprocessors (CMPs) is to allow each core to run at the maximum frequency it can operate at. In this case, the CMP becomes heterogeneous even though the individual cores are identical by design, and it can cause unpredictable performance.

A recent proposal, Booster [9], copes with this variation-induced heterogeneity by introducing dynamic heterogeneity to balance workloads. The Booster CMP includes two power supply rails set at two different voltages. This allows each core to quickly switch between two different maximum frequencies by dynamically assigning each core to either of the two power rails using a gating circuit. The Booster CMP virtually eliminates the effects of core-to-core frequency variation by letting slow cores spend more time on the high voltage rail. It further reduces the effects of workload imbalances by taking hints from synchronization libraries.

Another serious problem with NTC systems is single-thread performance. With each thread running at a relatively low frequency, the performance of multithreaded applications can suffer dramatically when parallelism is limited due to synchronization or other serializing events. This paper proposes a software framework to improve the performance of multithreaded programs using the mechanisms for dynamic heterogeneity introduced by Booster. Given the capability of selectively running a set of cores in faster mode, how to schedule the faster mode can make a big difference for the overall program performance. Our framework first analyzes a target application to understand the parallelism structure of the program. Then, it instruments the program with the code to dynamically monitor the program behavior and adjust the priority of each thread. At runtime, the probabilistic scheduler assigns the cores to the fast mode according to the priority set by the instrumented code. In this way, the program critical paths are accelerated by intelligently utilizing the fast mode and thereby improving overall program performance.

The rest of the paper is organized as follows. Section 2 analyzes parallel benchmarks to show the potentials whereas smart scheduling of fast mode can improve performance. Section 3 presents our probabilistic priority-based scheduler, and Section 4 describes how the combination of static analysis and dynamic monitor assigns priorities to accelerate program critical paths. Section 5 demonstrates the performance improvement of our framework, followed by Section 6 discusses related work. Finally, we summarize the contributions and conclude in Section 7.

2. Motivational Data

In this section, we discuss how well multithreaded programs can scale as the number of cores increases. We analyze the bottlenecks that prohibit the programs from scaling better to demonstrate the potentials where smart scheduling of dynamic heterogeneity can improve the performance. All the experiments in this section were conducted on a 32-core machine consisting of four 8-core Intel Xeon X7560 processors with 24MB per-chip shared L3 cache, and 32GB of memory.

Figure 1 shows the speedups of a subset of PARSEC [1] benchmarks with different number of threads normalized against their single thread execution. Ideally, a perfectly scaling application has to show the speedup equal to the number of threads. As can be

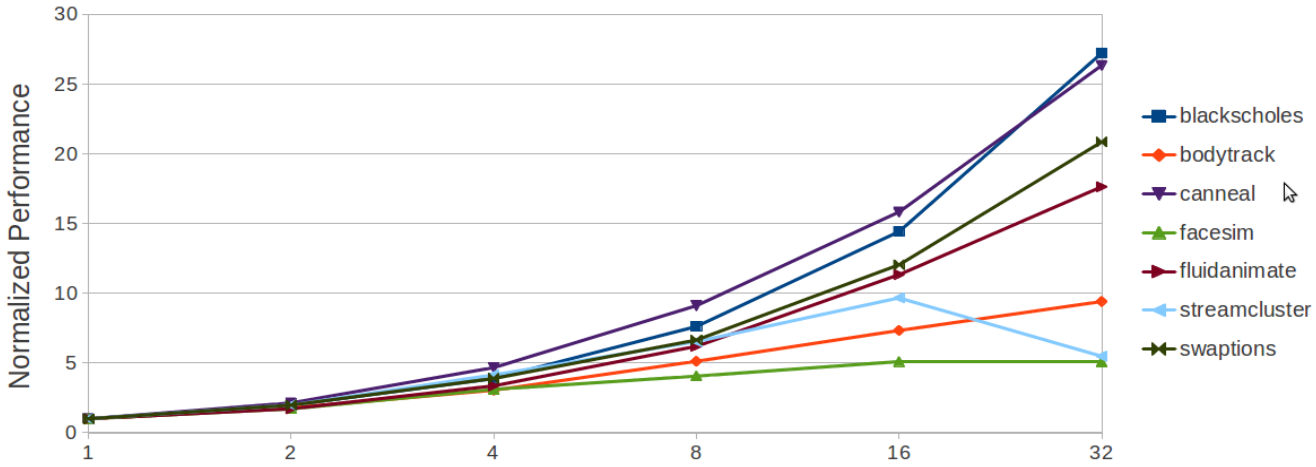

Figure 1: Speedup of PARSEC [1] benchmarks with varying number of threads compared to single thread executions.

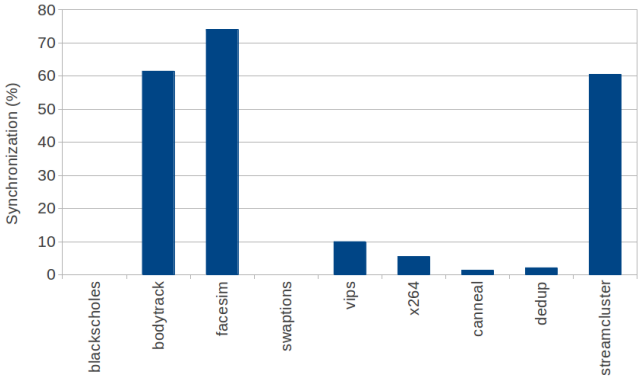

Figure 2: CPU cycles spent blocked for synchronization operations.

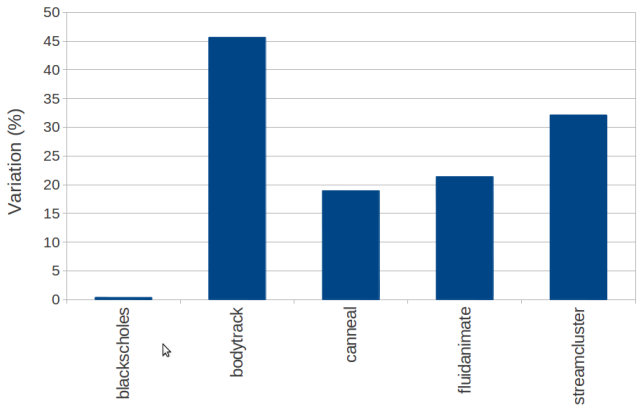

Figure 3: Average arrival time variation between the fastest thread and the slowest thread.

seen, the benchmarks possess varying amount of scalability. While some programs such as blackscholes and canneal scale pretty close to the ideal, others like streamcluster or facesim do not scale very well. Regardless of whether they scale well or not, all of them show less performance improvement per thread as the number of threads increases.

According to the recent studies [3, 5], there are several factors that hinder shared-memory parallel programs from scaling perfectly: contention for shared resources such as last-level cache (LLC) and memory bandwidth, synchronization stalls including spinning and yielding, and workload imbalance and parallelization overhead. Eyerman et al. [5] quantifies the impact of these scaling delimiters and show that synchronization is the most important component for the most of the benchmarks, especially the poorly scaling ones.

In order to re-confirm their findings, we focus on how much portion of processor time is wasted waiting for synchronization operations including mutex locks, condition variable waits, and barrier waits. We intercept every Pthread library calls by overloading LD_PRELOAD environment variable in Linux and measure waiting time for each operation. Figure 2 depicts the portion of time spent for synchronization.

Comparing Figure 2 and Figure 1, we can see the benchmarks that spend more time for synchronization do not scale very well, which supports the previous finding. Furthermore, this shows the potential of performance improvement if we can reduce the time spent for synchronization operations by assigning fast mode in a considerate way.

We further analyze the benchmarks to see if there are some common patterns for synchronization operation usage. The most common and basic task of synchronization operations is to use mutex locks to guarantee atomic access to shared variables. If there are mutexes that many threads often compete against each other to acquire, they can be performance and scaling bottleneck. The code regions which hold contended mutexes are important target of our fast mode scheduling.

Another common synchronization pattern is the phase control of data-parallel threads. Many shared memory multithreaded programs exploit data parallelism, whereas multiple threads execute the same code on different data regions. These programs often need to synchronize those threads to make sure they progress to the next step together.

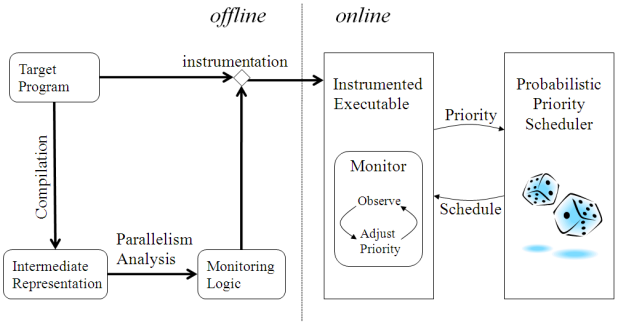

Figure 4: Overview of the proposed acceleration framework.

This type of synchronization can be implemented with barriers, join operations, and condition variables. The potential performance improvement for this synchronization pattern comes from the fact that not all threads arrive at the synchronization points at the same time. The arrival time for each thread varies, and the slowest ones forms the critical path for the execution.

Figure 3 presents the variation of the arrival times. We measure the time from the thread creation to the join point for blackscholes, and we use the time from the previous barrier to the next barrier for bodytrack, canneal, fluidanimate, and streamcluster. We take the geometric mean of the ratio of the fastest thread and the slowest thread. We can see there are significant variations in the arrival time for this type of synchronization and this is also an important performance improvement opportunity for our acceleration framework.

3. Probabilistic Priority-based Scheduling

Figure 4 presents the system architecture of our acceleration framework. It consists of two major parts: the static analysis and instrumentation part that works at compile time, and the monitor and scheduler part that works at runtime.

The static analysis and instrumentation part takes the target program as input, and compiles it into the intermediate representation first. Then, it analyzes the control and parallelism structure of the program. Based on the analysis information, it synthesizes and instrument the monitor logic into the program. The details of the static analysis and the optimization will be covered in Section 4.

At runtime, the instrumented code monitors the behavior of the program and adjusts the priority of the program. The basic idea is to let the thread executing the critical path runs at the fast mode so that the critical path gets accelerated. However, it is not possible to determine which thread becomes the critical path ahead of time. Therefore, the dynamic monitor increases the priority of the thread which is more likely to be included in the critical path. Then, the probabilistic priority-based assigns the fast mode to the higher priority threads with the higher probability by rolling a dice.

We adopt the interface of the probabilistic scheduler from Lottery scheduling [12]. It assigns lottery tickets to each thread according to the priority of the thread. In other words, the number of lottery tickets that each thread gets is proportional to the priority of the thread. The scheduler decide the winning lottery by rolling a dice and the thread which is holding the lottery gets assigned to the fast

```
1 :  double pgain(long x, Pointts *points, ...) {
2 :    long bsize = points->num / nproc;
3 :    long k1 = bsize * pid;
4 :    long k2 = k1 + bsize;
5 :    ...
6 :    pthread_barrier_wait(barrier);
7 :    /* INSTRUMENTED CODE BEGIN */
8 :    long PROGRESS_GRANULE = (k2 - k1) / NUM_STEPS;
9 :    /* INSTRUMENTED CODE END */
10:    for ( i = k1 ; i < k2 ; i++ ) {
11:      float x_cost =
12:       dist(points->p[i],points->p[x],points->dim)
13:       * points->p[i].weight;
14:      float current_cost = points->p[i].cost;
15:      if ( x_cost < current_cost ) {
16:        switch_membership[i] = 1;
17:        cost_of_opening_x += x_cost - current_cost;
18:      } else {
19:        int assign = points->p[i].assign;
20:        lower[center_table[assign]] +=
21:                      current_cost - x_cost;
22:      }
23:      /* INSTRUMENTED CODE BEGIN */
24:      if ( (i - k1) % PROGRESS_GRANULE == 0 )
25:        halve_priority_tickets();
26:      /* INSTRUMENTED CODE END */
27:    }
28:    pthread_barrier_wait(barrier);
29:    ...
30:  }
```

Figure 5: An example of progress monitoring instrumented code.

mode. Statistically, if we repeat this assignment process many times, the ratio of the time each thread gets the fast mode approaches to the ratio of priority. The lottery abstraction allows the flexible adjustment and delegation of the priority for the fast mode.

4. Dynamic Priority Management

The fundamental goal of our priority management is to assign higher priorities to the threads that are more likely to be included in the critical path. To do so, our framework analyzes the control and parallelism structure statically and instruments the monitor code that dynamically adjusts the priority of each code region according to its runtime behavior.

In this section, we explain how our monitor code observes the runtime behavior and adjust the priority. We follow the guide of the findings described in Section 2 to concentrate on the two very common patterns of synchronization. Section 4.1 focuses on the phase control of data-parallel threads, and we cover the mutexes used for guaranteeing atomic access to the shared variables in Section 4.2.

4.1. Progress Monitoring

As we measured in Figure 3, data-parallel threads often exhibit varying arrival times to the joining or barrier synchronization points. There are a number of factors that cause the variation. Among them are control flow deviation, non-uniform effect of memory subsystem, and other synchronization operations such as mutex locks. When these factors accumulate and make a specific thread the slowest, it turns out to be in the critical path. Therefore, if a thread progresses slower than other threads in the earlier stages of the task, it is likely to be also slower in the end and thus in the critical path.

From the previous observation, our framework tries to assign higher priority to the threads that shows slower progress. In order to do that, our static analysis divides a chunk of task into the multiple of the smaller chunks that approximately have the same

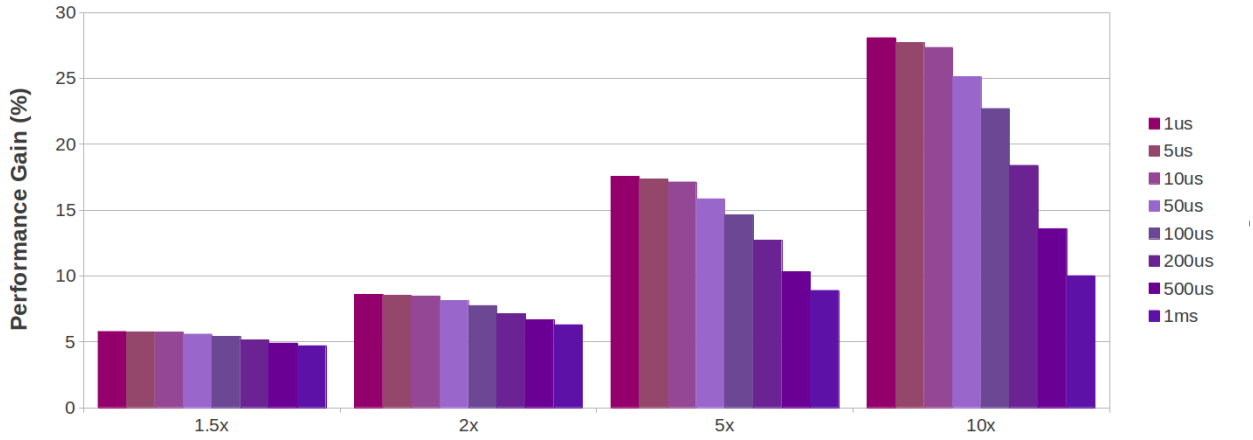

Figure 6: Performance gain of our framework for streamcluster.

amount of work. Then, it instruments the monitoring code between the chunks that lowers the thread's priority. The rationale behind this policy is that if a thread reaches a checkpoint earlier than the other threads it is less likely to be in the critical path.

For the straight lines of code, we use the type and the number of instructions to estimate the amount of work each chunk has. We further refine the model by profiling the average number of cycles each type of instructions takes. Control divergence makes it difficult to statically estimate the amount of work, if two different paths have significantly different number of instructions. For control divergence, we exploit edge profiling to approximate the amount of work. In order words, we use the weighted sum of each path's amount of work with the edge bias as weight.

Heavy workload chunks usually involve with loops. Based on the assumption that each iteration virtually has the similar amount of work, we use the number of iterations as the amount of work to divide the chunks. Figure 5 shows an example of our monitoring code instrumentation for loops. It is the workload phase that takes the most substantial amount of the time in the streamcluster benchmark. This phase consists of one big loop (line 10 line 27) between two barrier synchronization, and the number of iteration is determined at runtime by the parameters of the input size and the number of threads. We instrument the code to decide the monitoring granularity based on the parameters in line 8. Then, the instrumented code adjust the priority at every PROGRESS_GRANULE iterations in line 25.

4.2. Priority Delegation

Another type synchronization that we exploit to accelerate critical paths is mutexes. Since only one thread that is holding a mutex can progress among the threads that require the same mutex, the code region holding a mutex is more likely to be in the critical path than other regions without any mutex. We instrument after every mutex acquisition to increase the priority.

The threads holding the contended mutexes are even more likely to be in the critical paths. Furthermore, the contended mutexes can cause the priority inversion problem, in which a higher priority thread is waiting for a mutex held by a lower priority thread. In order to solve the priority inversion problem and give emphasis on the threads holding the contended mutexes, our runtime monitoring system enables temporary transfer of priority tickets. When a thread tries to acquire a mutex, it first checks whether the mutex is held by another thread or not. If so, it temporarily transfers its priority tickets to the thread holding the mutex and wait for the mutex. In this way, the threads holding the contended mutexes get higher priority.

5. Performance Evaluation

In this section, we present the effectiveness of our acceleration framework with the performance improvement on the streamcluster benchmark. We estimate the performance improvement by post-processing the traces generated by the instrumented code with the progress time indication. The traces are generated by the program running 32 threads on a 32-core machine consisting of four 8-core Intel Xeon X7560 processors with 24MB shared L3 cache per chip and 32GB of total memory. We assume that only one core can run in the fast mode at a time, but the same scheduling mechanism can be applied when more than one core can be accelerated simultaneously.

Figure 6 presents the performance improvement over the conventional round robin scheduling with the modern operating system scheduling quantum size, varying the acceleration of the fast mode from 1.5x to 10x and the scheduling quantum size from 1 microsecond to 1 millisecond. These are reasonable parameter settings as Miller et al. [9] varies the core and L1 clock frequency from 300MHz to 2300Hz and assumes 10 cycles (10 nsec for 1GHz frequency) of transition time between the fast and the slow mode.

As can be seen in the graph, our acceleration framework gives substantial performance improvement, from 5% to 27% without any hardware modification. Our acceleration scheduler works better with smaller time quanta as it assigns the fast mode probabilistically. In addition, our framework is relatively better than the round robin scheduler for higher acceleration rate because the round robin scheduler is not effective to distribute the computational power gained by the acceleration.

6. Related Work

The key idea of our framework is to accelerate the critical path identified by synchronization operations with the capability of dynamically

changing the performance of each core. In this section, we discuss the previous proposals related to our work focusing on two parts: i) dynamic heterogeneity, and ii) critical path identification via synchronization.

Our work is broadly based on the architectural capability of changing the throughput of each core. Miller et al. suggests the same capability to overcome the sensitivity to process variation of NTC in Booster [9]. Bower et al. [2] also expects dynamic heterogeneity due to process variability, physical faults, and dynamic voltage and frequency scaling. Other than changing the throughput of cores by scaling frequency, Lukefahr et al. [8] proposes fast switching by integrating two different types of architectural computing engines into a core. Assuming the low overhead dynamic heterogeneity suggested in these previous proposals, we further improve the performance of the multithreaded programs by smartly scheduling the fast mode of each core.

Another vein of previous work related to our proposal is to use synchronization operations to guide hardware acceleration. Booster SYNC [9] adjust thread priorities taking hints from barriers, locks, and condition waits. Suleman et al. [10] migrates the threads running critical sections to the fast cores, and Joao et al. [6] extends their work to cover other types of bottlenecks including barriers. The most important difference of our work from these previous proposals is that our scheduling works entirely in software. Moreover, our framework pro-actively accelerates the code between barriers by tracking their progress via the instrumented monitor.

7. Conclusion

We introduce a novel software framework to improve the performance of shared-memory multithreaded programs via considerate scheduling of the fast mode cores, given the capability of selectively running a set of cores in faster mode for near-threshold computing. The technique works by statically analyzing target program and instrumenting dynamic monitoring and priority management code into the program. At runtime, the probabilistic scheduler assigns the fast mode to the threads with higher priority by rolling a dice. We present the effectiveness of our framework with the post-processing result of the streamcluster traces on a 32 core machine. Varying the parameters corresponding to a previous work [9], we show our framework can give the substantial performance improvement (5% - 27%).

References

[1] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proc. of the 17th International Conference on Parallel Architectures and Compilation Techniques*, 2008, pp. 72–81.

[2] F. A. Bower, D. J. Sorin, and L. P. Cox, "The impact of dynamically heterogeneous multicore processors on thread scheduling," *IEEE Micro*, vol. 28, no. 3, pp. 17–25, 2008.

[3] G. Chadha, S. Mahlke, and S. Narayanasamy, "When less is more (limo): Controlled parallelism for improved energy efficiency," in *Proc. of the 2012 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2012.

[4] R. G. Dreslinski, M. Wieckowski, D. Blaauw, D. Sylvester, and T. Mudge, "Near-threshold computing: Reclaiming moore's law through energy efficient integrated circuits," *Proceedings of the IEEE*, vol. 98, no. 2, pp. 253–266, Feb. 2010.

[5] S. Eyerman, K. D. Bois, and L. Eeckout, "Speedup stacks: Identifying scaling bottlenecks in multi-threaded applications," in *Proc. of the 2012 IEEE Symposium on Performance Analysis of Systems and Software*, 2012, pp. 145–155.

[6] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt, "Bottleneck identification and scheduling in multithreaded applications," in *20th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012, pp. 223–234.

[7] U. R. Karpuzcu, K. B. Kolluru, N. S. Kim, and J. Torrellas, "Varius-ntv: A microarchitectural model to capture the increased sensitivity of manycores to process variations at near-threshold voltages," in *Proc. of the 2012 International Conference on Dependable Systems and Networks*, 2012, pp. 1–11.

[8] A. Lukefahr, S. Padmanabha, R. Das, F. Sleiman, R. Dreslinski, T. Wenisch, and S. Mahlke, "Composite cores: Pushing heterogeneity into a core," in *Proc. of the 45th Annual International Symposium on Microarchitecture*, 2012.

[9] T. N. Miller, X. Pan, R. Thomas, N. Sedaghti, and R. Teodorescu, "Booster: Reactive core acceleration for mitigating the effects of process variation and application imbalance in low-voltage chips," in *Proc. of the 18th International Symposium on High-Performance Computer Architecture*, 2012, pp. 1–12.

[10] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt, "Accelerating critical section execution with asymmetric multi-core architectures," in *17th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009, pp. 253–264.

[11] R. Teodorescu and J. Torrellas, "Variation-aware application scheduling and power management for chip multiprocessors," in *Proc. of the 35th Annual International Symposium on Computer Architecture*, Jun. 2008, pp. 363–374.

[12] C. A. Waldspurger and W. E. Weihl, "Lottery scheduling: Flexible proportional-share resource management," in *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*, 1994.