Asymmetry Aware Scheduling Algorithms for Asymmetric Multiprocessors

Nagesh Lakshminarayana   Sushma Rao   Hyesoon Kim
School of Computer Science
Georgia Institute of Technology
{nageshbl, sushma16, hyesoon}@cc.gatech.edu

Abstract

Multiprocessor architecture is becoming popular in both desktop processors and mobile processors. Especially asymmetric architecture shows promise in saving energy and power. However, how to design applications and how to schedule applications in asymmetric multiprocessors are still challenging problems.

In this paper, we evaluate the performance of applications in asymmetric multiprocessors to understand the characteristics of asymmetric processors. We also evaluate a task size aware scheduling algorithm and a critical section length aware scheduling algorithm in asymmetric multiprocessors. We show that when workload is asymmetric, the task size aware scheduler can improve performance by up to 14% compared to a scheduler which does not consider asymmetric characteristics.

1. Introduction

Asymmetric multiprocessor architecture has been proposed to be a power efficient multiprocessor architecture [4, 10, 2, 15]. Research has shown that these architectures provide power-performance effective platforms for both throughput-oriented applications and applications that would benefit from having high performance processors.

Unfortunately, the performance behavior of multithreaded applications in asymmetric architectures has not been studied widely. Balakrishnan et al. [3] evaluated the performance behavior in an asymmetric multiprocessor (AMP) and a symmetric multiprocessor (SMP) . However, in their work, they only showed performance effects. Grant and Afsahi [9] studied power-performance efficiency but they only focused on scientific applications.

Unfortunately, job scheduling polices for multithreaded applications on an AMP have not been studied widely. The performance of multithreaded applications in an AMP is heavily dependent on thread interactions. Balakrishnan [3] showed that the performance of multithreaded applications could be limited by a thread that is running on a slow processor in an AMP. They suggested that using dynamic scheduling in OpenMP reduces this problem, but dynamic scheduling is not directly applicable to all multithreaded applications.

In this paper, first we analyze the performance behavior of multithreaded applications in AMPs. After that we propose two new scheduling polices for AMPs: *the longest job to a fast processor first (JFFPF)* and *critical job to a fast processor first (CJFPF)*. The basic idea of these polices is that when a thread is likely to take longer than other threads, we send the thread to a fast processor. We evaluate these two policies using micro-benchmarks.

The contributions of our paper are

1. We evaluate the performance behavior of multithreaded desktop applications in AMPs.
2. We propose two new job scheduling algorithms for AMPs and show that they provide performance improvement.

2. Methodology

2.1. Evaluation System

We use an 8-core multiprocessor system as shown in Table 1 to measure performance. All applications are running with 8 threads. We use *SpeedStep* technology [1] with the `cpufreq governors` to emulate an AMP. Table 2 describes four machine configurations. We use RHEL Desktop 5 (Linux kernel 2.6.18).

Table 1. The System Configurations

processor	2 socket 1.87 GHz Quad-core Intel Xeon
memory system	4MB L2-cache, 8GB RAM
I/O	40GB HDD, Quadro NVS 285

2.2. Benchmarks

We use PARSEC [5], and ITK [16] (a medical image processing application) for our evaluations. We

Table 2. Four different machine configurations

All-slow (SMP)	All 8 processors are running at 1.6GHz
One-fast (AMP)	1 processors are running at 1.87GHz 7 processors are running at 1.6GHz
Half-half (AMP)	4 processors are running at 1.87GHz 4 processors are running at 1.6GHz
All-fast (SMP)	All 8 processors are running at 1.87GHz

use the native input set for the PARSEC benchmarks. We also design micro-benchmarks, `matrix multiplication`, and `globalSum` applications to evaluate thread behavior more closely. The PARSEC and ITK benchmarks are compiled with gcc 4.1.2 [8] with `-O3 -fprefetch-loop-arrays` flags.

3. Performance Evaluations in SMPs and AMPs

We evaluate the PARSEC benchmarks on the four machine configurations. Based on the results, we classify the benchmarks into three categories: *slow-limited* (the performance of half-half is the same as that of all-slow), *middle-perf* (the performance of half-half is between that of all-slow and all-fast), and *unstable* (the performance of an application varies significantly across runs).

Figure 1 explains why there are slow-limited, middle-perf and unstable applications. In case (a), there is a barrier at the end of the program. Therefore, the overall performance of the application is dominated by the slowest thread. Hence, such an application will be slow-limited. If sequential sections of the code dominate the overall execution time like in case (b), the performance of the application would be between the performance of all-fast and all-slow (i.e., middle-perf). Case (c) explains the unstable case. The application has several fork-join sections. After threads have joined, depending on where the single thread executes, the performance varies. This causes unstable behavior.

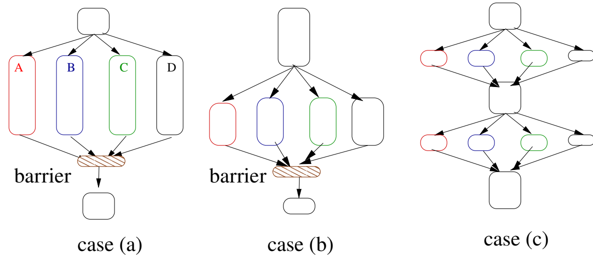

Figure 1. Fork-join cases

Figure 2 shows the performance of the PARSEC benchmarks in the four machine configurations. The results show that on average half-half, one-fast, and all-slow increase the execution time by 7.4%, 10.5%, and by 10.6% respectively compared to all-fast. Half-half performs more similarly to all-slow due to several slow-limited benchmarks. Since one-fast and half-half show similar trends, we only use half-half in the rest of the evaluations.

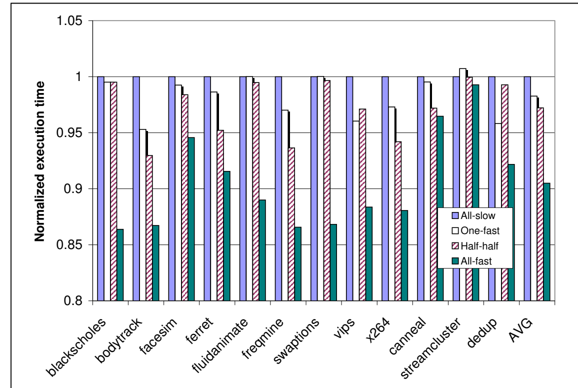

Figure 2. Experiments in the four machine configurations (PARSEC)

Table 3 summarizes the category of benchmarks and also shows the number of synchronization primitives in the PARSEC Benchmark suite. The data is collected with Pin [14]. All numbers are totals across all threads. Numbers for synchronization primitives also include primitives in system libraries.

`BlackScholes` is the typical slow-limited benchmark. It has only one barrier[1] at the end of the application. Hence, the overall performance is limited by the slowest thread. `Facesim` and `swaptions` are also limited by the barriers. Applications that have a large number of locks or barriers (`fluidanimate`, `vips`, `dedup`, and `bodytrack`) show unstable behavior. The remaining applications show middle-perf behavior.

4. Longest Job To a Fast Processor First (LJFPF) Policy

4.1. Mechanism

We propose a new scheduling policy, called *the longest job to a fast processor first (LJFPF)*. The basic algorithm of LJFPF is that when a thread has a longer task than others (the application provides the relative task length information), the scheduler sends the thread to a fast processor. A task is the work assigned to each thread. In this paper, the size of a task is usually deter-

[1]In Table 3, `BlackScholes` has 8 barriers. This is because all 8 threads encounter the same barrier at run-time.

Table 3. Characteristics of the Parsec Benchmarks

Application	Locks	Barriers	Cond. variables	AMP performance category
BlackScholes	39	8	0	slow-limited
Bodytrack	6824702	111160	32361	unstable
canneal	34	0	0	middle-perf
dedup	10002625	0	17	unstable
facesim	1422579	0	330521	slow-limited
ferret	7384488	0	16938	half-half
fulidanimate	1153407308	31998	0	unstable
freqmine	39	0	0	middle-perf
streamcluster	1379	633174	1036	middle-perf
swaptions	39	0	0	slow-limited
vips	792637	0	147694	unstable
x264	207692	0	13793	half-half

mined by the number of iterations. We modify the applications so that they send the relative task length information to the kernel using a system call before a thread is created. It is almost impossible to predict the exact execution time at compile time, so we estimate the length of a task based on how many iterations are assigned to each thread. Since, the division of work for each thread is done statically, the application can know the number of iterations for each thread at compile time. Note that the total number of iterations is all determined at compile time in all of the evaluated applications in this section.

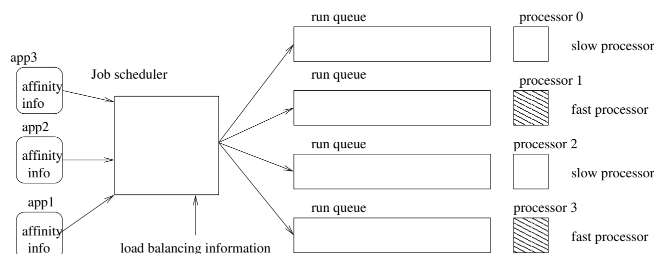

Figure 3. Job scheduling mechanism

Figure 3 shows the conceptual view of the asymmetry aware scheduling mechanism. Each task has the affinity information about the relative task size. The scheduler uses that information to decide whether a thread should be sent to a slow processor or a fast processor.

Figure 4 shows the scheduling algorithm. We use Linux Kernel 2.6.18-8. The shaded boxes indicate the modified code sections. When an application calls *pthread_create()*, it calls *sys_clone()*, *do_fork(),* and *copy_process()* functions sequentially. The scheduler logic for scheduling a newly created thread is in the *sched_fork()* function. We pass information about a task to the scheduler using new system calls and modify both *copy_process()* and *sched_fork()* functions to use this information.

4.2. Evaluation

4.2.1. Matrix Multiplication

Figure 5 compares the performance of the `matrix multiplication` application in three different machine configurations (all-fast, all-slow, and half-half). LJFPF and round robin (RR) scheduling polices are used for half-half. All-fast and all-slow use RR (Note that RR performs similar to the default scheduler in the Kernel).

The `matrix multiplication` application computes the product matrix of two 2400X2400 matrices by dividing the computation among 8 threads. X-Y means 4 threads compute X rows each of the product matrix and the other 4 threads compute Y rows each of the product matrix.

There are *pthread_join()* function calls at the end of the `matrix multiplication` application. Hence, when the workload is symmetric (300-300), the performance of half-half is slow-limited. However, when the application has strongly asymmetric characteristics (340-260, 350-250, 360-240), half-half with LJFPF actually performs as well as all-fast. In this case, the application is mainly limited by the long task threads. The long task threads execute on fast processors on both all-fast and half-half configurations, so all-fast and half-half show the same performance. As we can expect, half-half with RR performs poorly. Therefore, we can conclude that for a strongly asymmetric workload, LJFPF improves performance by up to 14% compared to the scheduler which does not consider asymmetric characteristic.

4.2.2. ITK

To test a real application with an asymmetric workload, we use the ITK application [16] (MultiRegistration), a medical image processing program. The main parallel loop in the ITK benchmark has 50 iterations and the number 50 is statically determined from the algo-

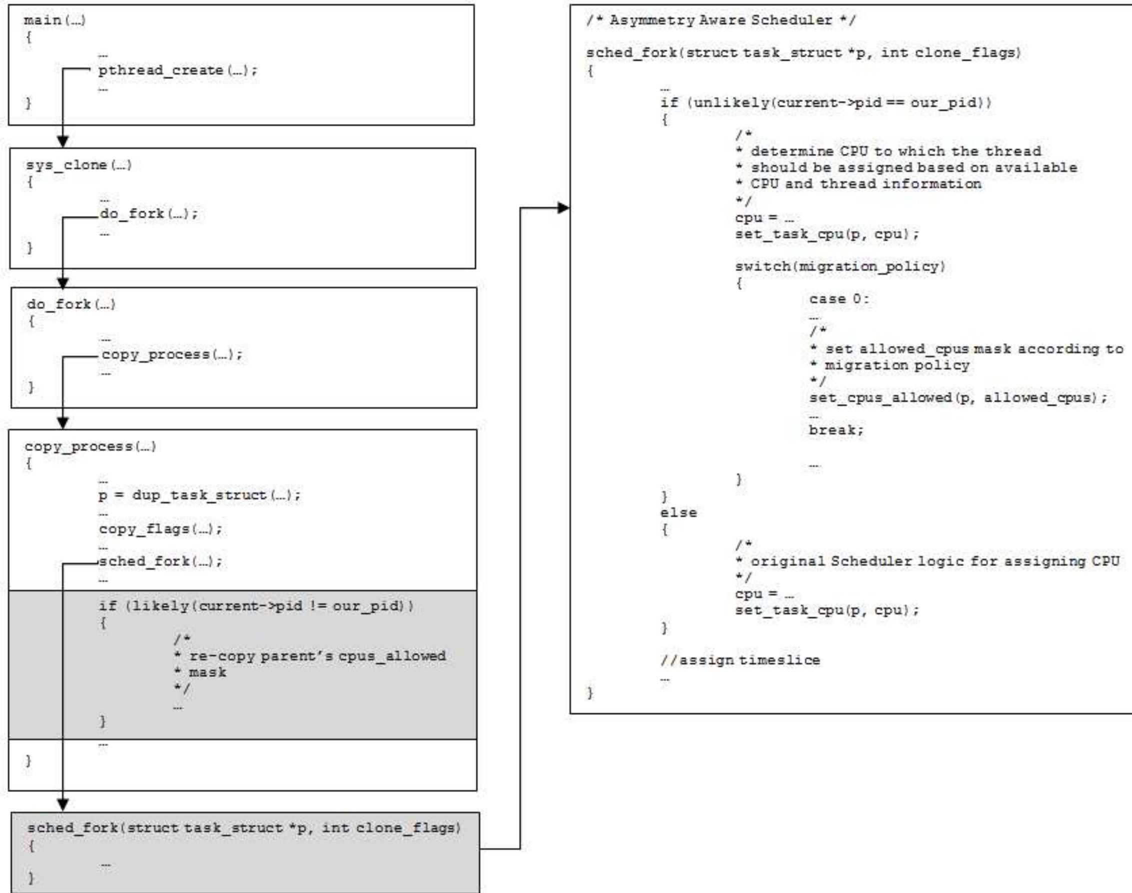

Figure 4. Thread Scheduling Algorithm

rithm. Since 50 is not a multiple of 8, ITK is a naturally asymmetric workload. Each thread executes 7, 7, 7, 7, 6, 6, 5, and 5 iterations of the loop respectively. This division is done at compile time. Our scheduling algorithm (LJFPF), at run-time, sends all 7-iteration task threads to fast processors and 6 and 5-iteration task threads to slow processors. Figure 6 shows the normalized execution time (all the data is normalized to all-fast).

The results show that half-half with LJFPF performs as well as all-fast. Furthermore, LJFPF improves performance by 2.3% compared to RR scheduling. The results also imply that LJFPF on half-half could save energy compared to using all-fast since half-half would consume less power compared to all-fast.

5. Critical Job to a Fast Processor First (CJFPF) Policy

5.1. Background

Multi-threaded applications are different from multiple single threaded applications because of thread interactions. Waiting for entering critical sections (acquiring a lock) and waiting for all the threads to finish (barrier) are the major sources of thread interactions. A lock is implemented using `futex` in Linux Kernel 2.6 [7]. Using futex, when a thread cannot acquire a lock, the system puts the thread into the sleep state. When the thread that had the lock releases the lock, it also wakes up a waiting thread. When a thread wakes up, the scheduler sends the thread to an idle processor. In our experiments, we set the number of threads equal to the number of processors, so a thread is usually sent to the same processor where it was executing before going to the sleep state.

Figure 7 shows two scenarios of critical section intensive applications. For every loop, each thread needs to enter the critical section to update a shared data structure, such as a hash table. When the critical section is short (case (a)), all three threads can perform useful work all the time. However, when the critical section is long (case (b)), threads spend most of their time waiting for acquiring locks [17]. Therefore, they would be in the sleep state for most of their execution time.

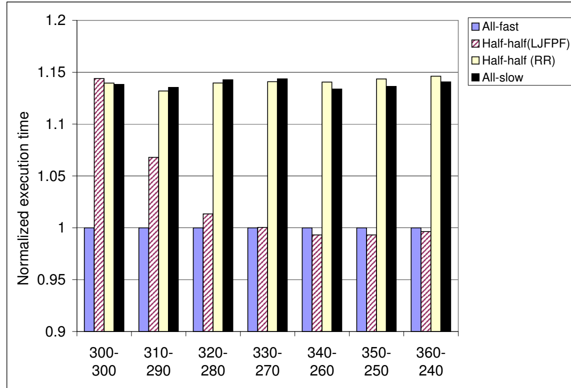

Figure 5. The performance behavior in the `matrix multiplication` application with the LJFPF scheduling policy

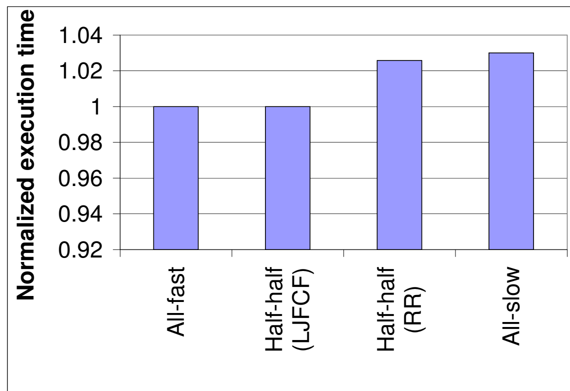

Figure 6. The performance and energy consumption behavior of the ITK benchmark

5.2. Mechanism

We propose another new scheduling policy, called *a critical job to a fast processor first (CJFPF)*. The basic algorithm of CJFPF is that when a thread has longer critical sections than others, the scheduler sends the thread to a fast processor. Threads can have critical sections of different length due to how they access the shared data structure. Again, we modify the application so that it sends the relative critical section length information to the scheduler. The rest of the scheduling mechanism is the same as LJFPF.

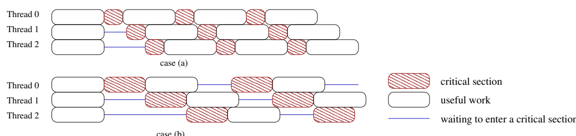

Figure 7. Example of critical section limited benchmarks

5.3. Critical Section Length Effects

First, we evaluate the effect of critical section lengths. We design a micro-benchmark in which we can adjust the length of critical sections. The application (`globalSum`) computes the sum of the elements of a large array using multiple threads. Each thread computes the sum of a contiguous section of the array. Each thread, after computing the sum of a certain number of elements (determined by the frequency of the critical section), enters a critical section to update the global sum value. We vary the length of critical sections from 10%, 15%, 20% of the total execution time.[2]

Figure 9 shows the speedup of the applications in three different configurations. When critical section length is 10%, all-slow and half-half perform worse than all-fast. However, when critical section length increases the difference between half-half and all-fast decreases. This is because processors spend more and more time in the sleep state. So, whether threads are waiting on fast processors or slow processors does not change the overall execution time significantly. However, all-slow is always slower than the other two. This is because both critical sections and non-critical sections are executed on slow processors.

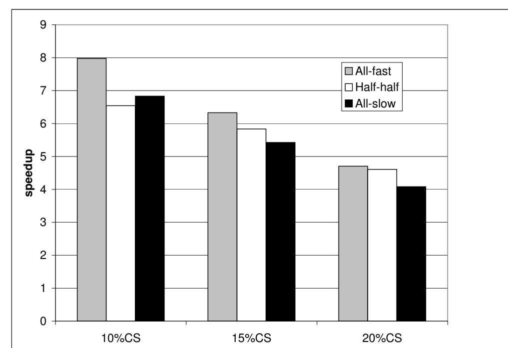

Figure 8. Critical Section length effects on performance

5.4. Evaluation

Figure 9 shows the speedup (compared to the sequential version of the code) between the CJFPF and the RR scheduling policies on half-half machine. X-Y means that 4 threads have X% critical section length and the other 4 threads have Y% critical section length. When

[2]The length of critical section is defined to be the sum of the total execution time spent in the critical section divided by the total sequential program execution time. The time that is spent in the critical section is also measured using the sequential version of the code. We vary the length by inserting extra computations inside the critical section

the critical length is 8-12, CJFPF on half-half performs 2.5% better than RR on half-half. However, when the critical length is 40-60, CJFPF improves performance by only 1.2% when compared to RR. This results show that when a critical section is short, it is important to schedule long critical section threads to fast processors. However, when the threads of an application are mostly waiting in sleep state due to long critical sections, the scheduling policy becomes less important.

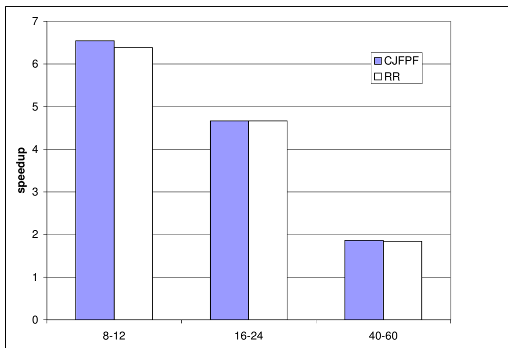

Figure 9. CJFPF and RR scheduling polices on half-half

6. Symmetric and Asymmetric Workload

We proposed two scheduling policies which are using asymmetric characteristics of workloads. In this section, we evaluate whether the PARSEC benchmarks have symmetric or asymmetric characteristics. We use two characteristics to evaluate the symmetric nature of workloads: the number of instructions in each thread and the number of synchronization primitives in each thread. We also define a *thread-group* of threads for this evaluation. Figure 10 illustrates the concept of thread-group. The thread-group can be either a parent thread itself or a group of threads which are running concurrently. Case (a) has two groups of threads, but case (b) has 4 groups. Although the application has a maximum of 4 threads running at any time, the total number of threads that ever exist can be higher. Case (a) has 5, and case (b) has 13.

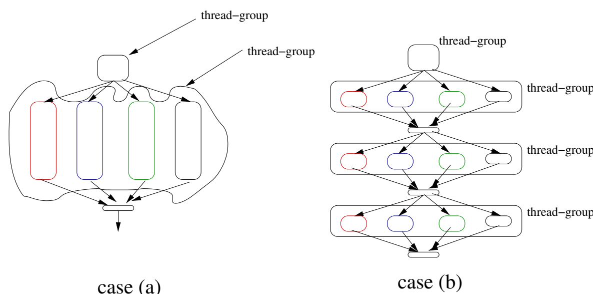

Figure 10. Thread-group examples

We define two new metrics to measure symmetry of workloads. *SymInst* and *SymLock* calculate the average of normalized standard deviations within each thread-group. We normalize the standard deviation to the total number of instructions or primitives inside a thread-group.

Table 4. Symmetry of workloads

Application	# of threads	# of thread-groups	SymInst	SymLock
BlackScholes	9	2	0.000	0.073
Bodytrack	9	2	0.003	0.007
canneal	9	2	0.003	0.000
dedup	25	4	0.009	0.95
ferret	35	6	0.014	0.83
facesim	8	2	0.03	0.000
fluidanimate	9	2	0.02	0.10
freqmine	8	2	0.12	0.098
streamcluster	49	7	0.013	0.017
swaptions	9	2	0.00	0.00
vips	11	3	0.0049	0.007

Table 4 shows the two metrics for each benchmark. `Freqmine` and `fluidanimate` benchmarks have the highest SymInst and SymLock respectively. Those two benchmarks are more likely to be asymmetric. Even though `streamcluster` has 49 threads during the execution time, it is a fairly symmetric workload.[3] `swaptions` shows the most extreme case. Both metrics have 0s. Overall, most benchmarks are symmetric workloads. However, since there are some applications which might be asymmetric, in future work, we will evaluate our scheduling policies with the PARSEC benchmarks.

The question is why are most applications symmetric? There are two possible answers. First, it is easy to divide data equally across all threads. Second, programmers or compilers already have divided the work equally to reduce the load imbalance problem. The next question is what the application writers should do for an application that might be run on asymmetric processors? We believe that this is an important question to research.

7. Related Work

Recently, task scheduling algorithms on heterogeneous/asymmetric architectures have been actively studied. For example, Sun [18, 6] and Intel [13, 20, 19] are looking at operating system managed solutions for heterogeneous/asymmetric processors. Many academia researchers [12, 11, 4, 21] are also developing task

[3] If both SymInst and SymLock of an application are less than 0.01, we consider the application to be symmetric.

scheduling algorithms. The major difference between other work and our work is that our work demonstrated the benefit of using the asymmetry aware scheduling algorithms in a real machine. Furthermore, none of the work has discussed the effect of critical sections in AMPs.

The most relevant work to our work is Li et al.'s thread migration policies in an AMP. In their work, the scheduler migrates a thread running on a slow processor to a fast core if there is an idle fast processor. Their mechanism reduces the load imbalance problem at the cost of thread migrations. However, our work is more focused on thread assignment. If long task threads are assigned to a fast processor from the beginning, there is no need for thread migrations. Furthermore, our work can be used together with Li et al.'s mechanism.

8. Conclusion

In this work, we evaluate the characteristics of multi-threaded applications in AMPs. We observe that barriers and critical sections are important characteristics of applications which decide the performance of AMPs. We propose two new simple but effective scheduling algorithms for AMPs. The scheduling algorithms send long task threads or long critical section threads to fast cores. Using knowledge of the applications and processor characteristics, these simple scheduling algorithms can improve performance by up to 14% on an AMP compared to the scheduler which does not consider the asymmetric characteristics.

In future work, we will focus on predicting application characteristics (e.g., the length of a task) without requiring information from programmers and designing task scheduling algorithms that use the predicted information for an AMP to improve performance thereby improving energy efficiency.

Acknowledgments

We thank Min Lee for helping us understand the Linux Kernel. We also thank Richard Vuduc and Onur Mutlu for insightful discussions and Jaekyu Lee and Sunpyo Hong for initial settings for the benchmarks. We thank the anonymous reviewers for their comments and suggestions. This research is supported by gifts from Microsoft Research.

9. References

[1] Enhanced intel speedstep technology for the intel pentium m processor– white paper, March 2004.

[2] M. Annavaram, E. Grochowski, and J. Shen. Mitigating amdahl's law through epi throttling. In *ISCA-32*, 2005.

[3] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The impact of performance asymmetry in emerging multicore architectures. In *ISCA-32*, 2005.

[4] A. Baniasadi and A. Moshovos. Asymmetric-frequency clustering: a power-aware back-end for high-performance processors. In *ISLPED*, 2002.

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. Technical Report TR-811-08, Princeton University, 2008.

[6] A. Fedorova, D. Vengerov, and D. Doucette. Operating system scheduling on heterogeneous core systems. Technical report, Sun Microsystem, 2007.

[7] H. Franke, R. Russell, and M. Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in linux. In *Ottawa Linux Symposium*, 2002.

[8] GCC-4.0. GNU compiler collection. http://gcc.gnu.org/.

[9] R. Grant and A. Afsahi. Power-performance efficiency of asymmetric multiprocessors for multi-threaded scientific applications. In *IPDPS*, 2006.

[10] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *MICRO-36*, 2003.

[11] R. Kumar, N. P. Jouppi, and D. M. Tullsen. Conjoined-core chip multiprocessing. In *Micro-37*, 2004.

[12] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *ISCA-31*, 2004.

[13] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architecture. In *In Proceedings of Supercomputing 07*, 2007.

[14] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[15] T. Y. Morad, U. C. Weiser, A. Kolodny, M. Valero, and E. Ayguad. Performance, power efficiency and scalability of asymmetric cluster chip multiprocessors. 5(1), 2006.

[16] National Library. Medicine insight segmentation and registration toolkit (ITK). http://www.itk.org/.

[17] M. A. Suleman, M. K. Qureshi, and Y. N. Patt. Feedback driven threading: Power-efficient and high-performance execution of multithreaded workloads on cmps. In *ASPLOS-XIII*, 2008.

[18] D. Vengerov, L. Mastroleon, D. Murphy, and N. Bambos. Adaptive data-aware utility-based scheduling in resource-constrained systems. Technical Report TR-2007-16, Sun Microsystem, 2007.

[19] P. H. Wang, J. D. Collins, G. N. Chinya, H. Jiang, X. Tian, M. Girkar, N. Y. Yang, G.-Y. Lueh, and H. Wang. Exochi: architecture and programming environment for a heterogeneous multi-core multithreaded system. In *PLDI-07*, 2007.

[20] P. H. Wang, J. D. Collins, G. N. Chinya, B. Lint, A. Mallick, K. Yamada, and H. Wang. Sequencer virtualization. In *Proceedings of the 21st annual international conference on Supercomputing*, 2007.

[21] S. Zhang and K. S. Chatha. Automated techniques for energy efficient scheduling on homogeneous and heterogeneous chip multi-processor architectures. In *Proceedings of the 2008 conference on Asia and South Pacific design automation*, 2008.