Understanding Performance, Power and Energy Behavior in Asymmetric Multiprocessors

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

***Abstract*—Multiprocessor architectures are becoming popular in both desktop and mobile processors. Among multiprocessor architectures, asymmetric architectures show promise in saving energy and power. However, the performance and energy consumption behavior of asymmetric multiprocessors with desktop-oriented multithreaded applications has not been studied widely.**

In this study, we measure performance and power consumption in asymmetric and symmetric multiprocessors using real 8 and 16 processor systems to understand the relationships between thread interactions and performance/power behavior. We find that when the workload is asymmetric, using an asymmetric multiprocessor can save energy, but for most of the symmetric workloads, using a symmetric multiprocessor (with the highest clock frequency) consumes less energy.

I. Introduction

Asymmetric multiprocessor architectures have been proposed to be power efficient multiprocessor architectures [3], [13], [1], [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 and energy behavior of multithreaded applications in asymmetric architectures has not been studied widely. Balakrishnan et al. [2] evaluated the performance of applications in an asymmetric multiprocessor (AMP). However, in their work, they only showed performance effects in an AMP. Grant and Afsahi [11] studied power-performance efficiency but they only focused on scientific applications.

In this study, we evaluate the performance and power consumption behavior of multithreaded applications in an AMP. We emphasize on understanding thread interactions since many modern applications have many locks and barriers. To understand the overall power consumption behavior, we measure the power consumption of two systems (8 processors and 16 processors). We measure the power consumption of whole systems rather than the power consumption of only processors, since performance and energy trade-offs should consider the entire system including DRAM memory and disk.

We use PARSEC [4], the recently released multithreaded benchmark suite for desktops, for our evaluations. We also design several microbenchmarks to understand thread interactions better. Furthermore, we modify the Linux scheduler to evaluate asymmetry aware scheduling algorithms on an AMP.

Our experiments yield three major conclusions. First, when threads do not interact intensively and when all threads have similar amounts of work, a symmetric multiprocessor (SMP) with fast processors (i.e., the highest clock frequency) consumes the least amount of energy. Second, when thread interactions increase, an SMP with slow processors or an AMP could provide the best energy savings. Third, when the workload is strongly asymmetric (i.e., each thread in the workload has different amount of work), an AMP consumes the least amount of energy. Hence, depending on the thread characteristics in multithreaded applications, a different machine configuration would provide the best energy savings.

The contributions of our paper are

1) To our knowledge, this is the first work that evaluates performance and the overall system power consumption behavior in an AMP for multithreaded desktop applications.
2) We thoroughly evaluate thread interaction behavior to study performance and energy trade-offs in an AMP.
3) We propose a new, simple, but effective job scheduling algorithm for an AMP and show that it provides the best energy savings for asymmetric workloads.

II. Methodology

A. Evaluation System

We use two systems as shown in Table I to measure performance and energy consumption.[1] Applications running on machine-I have 8 threads and applications running on machine-II have 16 threads. We use *SpeedStep* technology [12] with `cpufreq governors` to emulate an AMP. Table II describes three machine configurations. Machine-I runs RHEL 5 Desktop (Linux Kernel 2.6.18), while Machine-II runs RHEL 4 WS (Linux Kernel 2.6.9).

[1]Since machine-I and machine-II show similar trends, we mainly report results from machine-II except in Section VI.

TABLE I
THE SYSTEM CONFIGURATIONS

Machine-I (Dell Precision 490)	2 socket 1.87 GHz Quad-core Intel Xeon; 4MB L2-cache, 8GB RAM, 40GB HDD, Quadro NVS 285
Machine-II	4 socket 2 GHz Quad-core AMD Opteron 8350; 2MB L3-cache, 32GB RAM, 1TB HDD, Tesla C-870

TABLE II
THREE DIFFERENT MACHINE CONFIGURATIONS FOR MACHINE-II

All-fast	All 16 processors are running at 2GHz
All-slow	All 16 processors are running at 1GHz
Half-half	8 processors are running at 2GHz; 8 processors are running at 1GHz

B. Benchmarks

We use PARSEC [4], and an ITK [16] application (a medical image processing application) for our evaluations. We use the native input set for the PARSEC benchmarks. We also design microbenchmarks, `matrix multiplication`, `globalSum` and `parallel-for` applications to understand thread behavior more accurately. The PARSEC and ITK benchmarks are compiled with gcc 4.1.2 [9] with `-O3 -fprefetch-loop-arrays` flags.

C. Power Measurement

We use Extech 380801 AC/DC Power Analyzer [7] to measure the overall system power consumption. The power data is sent to a datalog machine using RS232 every 0.5 seconds.

III. PERFORMANCE AND ENERGY CONSUMPTION BEHAVIOR OF PARSEC BENCHMARKS

We evaluate the PARSEC benchmarks on the three 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.

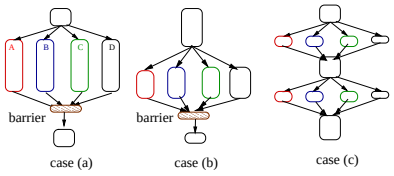

Fig. 1. Fork-join cases

Figure 2 shows the performance of the PARSEC benchmarks in the three machine configurations. The results show that on average half-half and all-slow increase the execution time by 43% and by 61% respectively compared to all-fast. Half-half performs more similarly to all-slow than to all-fast due to several slow-limited benchmarks. However, half-half performs similarly to all-fast for some applications such as `canneal` and `dedup`.

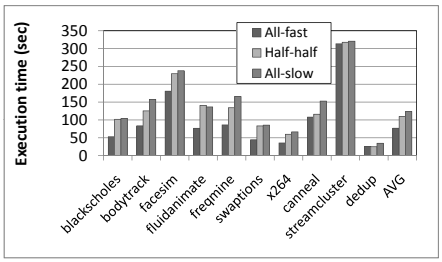

Fig. 2. Experiments in the three machine configurations (PARSEC)

Table III summarizes the categorization of benchmarks and also shows the number of synchronization primitives in the PARSEC benchmarks. The data is collected using Pin [6]. Note that the experiments are done with the 8 thread configuration. 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[2] at the end of the application. Hence, the overall performance is limited by the slowest thread. `facesim` and `swaptions` are also limited by barriers. Applications that have a large number of locks or barriers (such as `fluidanimate` and `bodytrack`) show unstable behavior. The remaining applications show middle-perf behavior.

TABLE III
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	middle-perf
facesim	1422579	0	330521	slow-limited
fulidanimate	1153407308	31998	0	unstable
freqmine	39	0	0	middle-perf
streamcluster	1379	633174	1036	middle-perf
swaptions	39	0	0	slow-limited
x264	207692	0	13793	half-half

A. Power Consumption Measurements

Table IV summarizes the average power consumption of different number of threads for each machine configuration. We use the `matrix multiplication` application[3] to measure the average power. A machine in idle state consumes about 10–20% less power than a machine that

[2]In Table III, `blackscholes` has 8 barriers. This is because all 8 threads encounter the same barrier at run-time.

[3]This application computes the product of two 2400X2400 matrices by dividing the computation among n number of threads.

executes 16 threads. All-slow (16 @ 1 GHz) with 16 threads consumes 15.7% less power than all-fast (16 @ 2 GHz) with 16 threads.

In this study, we focus on energy consumption rather than power consumption itself. The amount of energy consumed is proportional to both execution time and power. Hence, even though low frequency machine configurations consume less power than high frequency machine configurations, if they increase the execution time significantly, they would not result in energy savings. For example, if all-slow (with 16 threads) increases execution time by more than 15.7% in comparison with all-fast (with 16 threads), it would consume more energy than all-fast. Note that this 15.7% number is dependent on workloads.

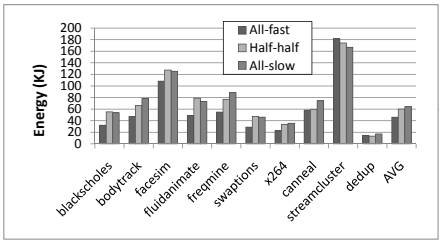

Fig. 3. Energy consumption behavior in three machine configurations (PARSEC)

Figure 3 shows the energy consumption behavior in three different machine configurations. On average, all-fast shows the best energy consumption behavior. Half-half consumes 30% more energy and all-slow consumes 39.5% more energy compared to all-fast. Since the performance slowdown is more than the benefit of power savings, both half-half and all-slow do not results in energy savings. Only `streamcluster` shows opposite behavior, wherein all-slow consumes the least amount of energy. The most distinguishable characteristic of `streamcluster` is that it has the most number of barriers (more than 600K barriers) among all the evaluated benchmarks. We suspect that the number of barriers also plays an important factor, which is why we evaluate the effect of barriers more carefully in the next section. For `dedup`, all-fast and half-half consume similar amounts of energy because the differences in execution times on the three machine configurations are small. From the results, we can conclude that on average using an SMP with fast processors saves energy except for a few applications. We will investigate more on why there are some applications that are exceptions in the following sections.

IV. Performance and Energy Consumption behavior with Critical Sections

Multithreaded applications are different from multiple single threaded applications because of thread interactions. Locks and barriers are the major sources of thread interactions. Waiting to enter critical sections (acquiring a lock) and waiting for all the threads to finish (barrier) are instances of thread interactions. In this section, we analyze how critical sections and barriers affect performance and power consumption behavior.

A. Background

Figure 4 shows a scenario of a critical section limited application [17]. When an application is limited by a critical section, most of the threads are in the idle state, waiting to acquire a lock. A lock is implemented using `futex` in Linux Kernel 2.6 [8]. 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, a waiting (sleeping) thread is woken up. 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 likely to be sent to the same processor where it was executed before going to the sleep state.

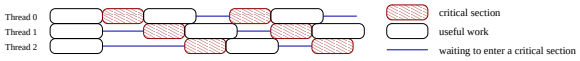

Fig. 4. Critical section limited application example

B. Critical Section Length Effects

To evaluate the effect of critical sections, we design a microbenchmark 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 block of elements in 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%, 50% to 75% of the total execution time.[4]

Figure 5 shows the average power consumption with critical sections of different lengths normalized to the average power of the 16 thread `matrix multiplication` workload for each machine configuration respectively . As we increase the critical section length, the average power consumption decreases because many threads are waiting in the idle state. For example, the 75% critical section workload on the 16 @ 2GHz configuration consumes only 85% of the 16 thread `matrix multiplication` workload at the same machine configuration.

Figure 6 shows the execution time and energy consumption normalized to the 16 @ 1GHz configuration. The execution time (lower is faster in the execution time graph) shows a similar trend across all SMPs regardless of critical section length (i.e., each SMP shows the same speedup across all the critical section lengths). However, the

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

TABLE IV
AVERAGE POWER CONSUMPTION (UNIT W) OF MACHINE-II(X@Y GHZ MEANS X NUMBER OF Y GHZ PROCESSORS)

Machine configuration	idle	1 thread	2 threads	4threads	8 threads	16 threads
16 @ 1 GHz (SMP)	480	485.5	488.5	494.5	506.5	531.5
8 @ 1 GHz 8 @ 2 GHz (AMP)	504	509.5	512.5	525	543.5	581
16 @ 1.2 GHz (SMP)	491	496	499.5	507.5	522	552
8 @ 1.2 GHz, 8 @ 2 GHz (AMP)	510	515.5	519.5	531	533	592
16 @ 1.4 GHz (SMP)	501	507	510.5	520	537.5	571.5
8 @ 1.4 GHz, 8 @ 2 GHz (AMP)	515	521	526.5	538.5	559.5	601
16 @ 1.7 GHz (SMP)	515	521	527.5	537.5	558.5	602.5
8 @ 1.7 GHz, 8 @ 2 GHz (AMP)	515	529	533.5	546	570	615.5
16 @ 2 GHz (SMP)	522	536.5	541	555.5	579.5	630.5

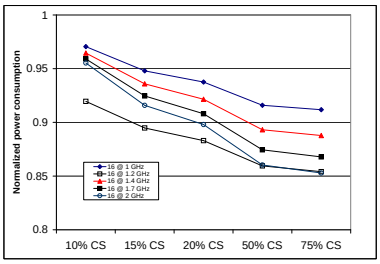

Fig. 5. Average power vs. different critical section length (normalized to the 16 thread `matrix multiplication` workload)

performance of AMPs is very sensitive to the critical section length. For example, the 8 @ 1.2 GHz, 8 @ 2 GHz machine configuration in the 10% critical section length benchmark is 37% slower than all-fast, but in the 75% critical section length benchmark it is only 10% slower than all-fast. This is generally true for all AMP configurations except for a few cases. This is because processors spend more and more time in the idle state. When we increase the frequency of entering a critical section, this trend becomes clearer. We will discuss this effect in the next section.

C. Critical Section Frequency Effects

Not only the length of critical sections, but also how often a thread enters a critical section impacts performance and energy consumption. In Figure 7, we vary the frequency of the critical sections. X-Y means that X% critical section length and Y frequency. The frequency of critical sections is varied from f10000 to f10. f10 means that the application enters a critical section approximately every million instructions.[5] As the results show, when the critical section frequency is higher, the execution time differences between half-half and all-fast are reduced. Consequently, half-half in both the 75%-f10000 and 75%-f1000 cases consumes less energy than all-fast.

The results show that if the majority of the execution time is spent waiting to acquire locks, having some slow processors does not affect overall performance significantly. This could be because threads on fast processors execute a critical section while threads on slow processors are still doing computation before entering the critical section. When the threads on the fast processors exit the critical section, the threads on the slow processors are ready to enter the critical section. In this case, slow processors do not increase the execution time significantly. However, all-slow increases the execution time significantly because even critical sections execute on slow processors all the time. If a thread scheduler can intelligently schedule threads that execute critical sections into fast processors, half-half would perform as fast as all-fast, then it would result in energy savings.

[5]f1000 enters a critical section 10 times frequently than f100. Section IV-B uses f100 for the experiments.

D. Barrier effects

Figure 8 shows the performance results when we vary the number of barriers in the `globalSum` program. Similar to the results of critical section experiments, when the number of barriers is small (1000), half-half performs as slow as all-slow. This is because the total execution time is dominated by the slowest thread. However, when the number of barriers increases, half-half performs similar to all-fast. When there are many barriers, the waiting time for barriers becomes a significant portion of the overall execution time. Again, when threads are waiting for other threads, having a few slow processors does not reduce performance significantly. With 1000 barriers, half-half consumes 74% more energy than all-fast, but with 1M barriers, half-half consumes only 7.5% more energy. Note that, when the number of barriers increases, the application also shows unstable behavior because the operating system interference increases [2].

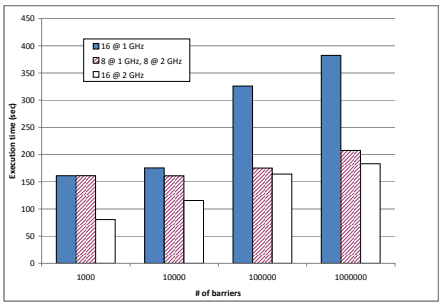

Fig. 8. Barrier effects on performance

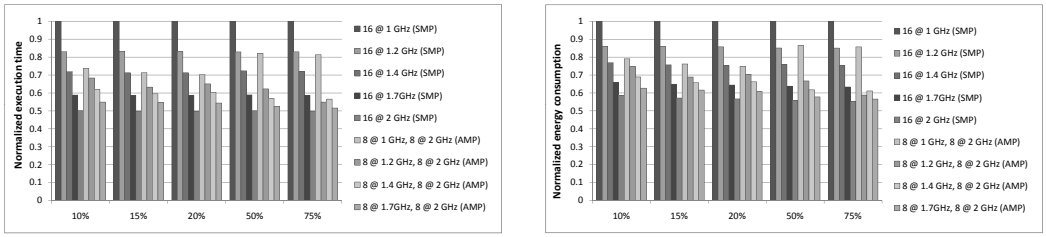

Fig. 6. Critical Section length effects on performance and energy (left: execution time, right: energy consumption)

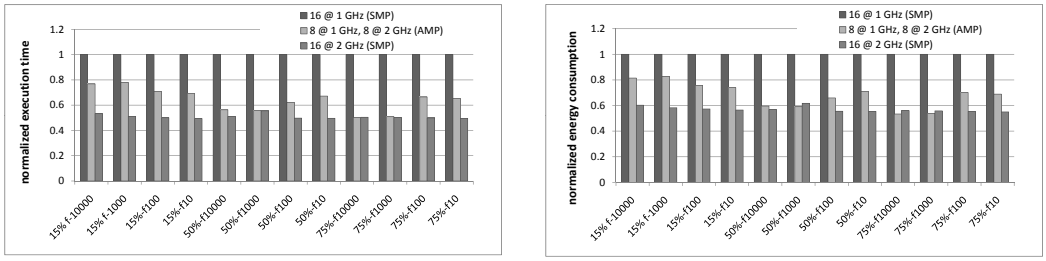

Fig. 7. Critical section frequency effects on performance and energy (left: execution time, right: energy consumption)

V. Dynamic Scheduling Effects in OpenMP

To improve the performance of an AMP, Balakrishnan et al. [2] suggested using a dynamic scheduling policy in OpenMP [18] programs. OpenMP supports several types of scheduling policies such as `static`, `dynamic`, and `guided`. The default is static scheduling, which statically divides the number of iterations among threads. In dynamic scheduling, each thread is assigned some number of iterations (`chunk` sets this number) at the start of the loop. After that, each thread requests more iterations after it has completed the work already assigned to it [5]. Guided scheduling is similar to dynamic scheduling except that dynamic scheduling uses a constant chunk size while guided scheduling adjusts the chunk size at run-time. Guided scheduling has the highest overhead since it needs to calculate the chunk size at run-time. Dynamic has medium overhead. Static has the lowest overhead since all the scheduling is done statically.

TABLE V
Performance and energy consumption of `parallel-for`

machine configuration	normalized exec. time static/dynamic	normalized energy static/dynamic
16 @ 1 GHz (SMP)	1.00	1.00
16 @ 1.2 GHz (SMP)	0.83	0.87
16 @ 1.4 GHz (SMP)	0.71	0.78
16 @ 1.7 GHz (SMP)	0.59	0.68
16 @ 2 GHz (SMP)	0.50	0.61
8 @ 1 GHz, 8 @ 2 GHz (AMP)	1.00/0.67	1.05/0.73
8 @ 1.2 GHz, 8 @ 2 GHz (AMP)	0.83/0.63	0.90/0.70
8 @ 1.4 GHz, 8 @ 2 GHz (AMP)	0.71/0.59	0.80/0.67
8 @ 1.7 GHz, 8 @ 2 GHz (AMP)	0.59/0.54	0.69/0.63

Table V shows the effects of static and dynamic scheduling[6] on the `parallel-for` application which computes the square of each element of a large array using the OpenMP *parallel for* directive.

The performance of static scheduling is dominated by the threads on slow processors in an AMP. However, dynamic scheduling can alleviate load imbalance, so the performance of the AMP is between all-slow and all-fast. Therefore, dynamic scheduling can reduce energy consumption significantly compared to static scheduling. In case of 8 @ 1.7GHz and 8 @ 2GHz, dynamic scheduling consumes almost the same amount of energy as 16 @ 2GHz (all-fast).

Among our evaluated PARSEC benchmarks, only `freqmine` can utilize dynamic scheduling. Dynamic scheduling improves the execution time of `freqmine` by 13% and reduces energy consumption by 12% compared to static scheduling in half-half. However, compared to all-fast, half-half still consumes 13% more energy. The main reason is that not all parallel code in `freqmine` can utilize the benefit of the dynamic scheduling (only loops can use dynamic scheduling). Hence, when an application can support dynamic scheduling, an AMP should utilize it, but we need other mechanisms to make AMPs perform as well as SMPs.

VI. A New Job Scheduling Policy For Asymmetric Workloads

In the previous sections, all benchmarks have symmetric workloads (i.e., all child threads have similar amounts of work).[7] When an application has asymmetric workloads, the performance of the application on an AMP is very dependent on the operating system's job scheduling policy. To evaluate the performance and energy consumption behavior with asymmetric workloads, we modify the Linux

[6]We also evaluate the guided scheduling policy but due to the overhead of the guided scheduling policy, it always increases the execution time more than 2 times. Hence, we only report the results of the dynamic scheduling policy.

[7]We measured the dynamic number of instructions for each thread in all applications. Most benchmarks have almost the same number of instructions across all child threads.

kernel scheduler to implement a new, simple, but effective scheduling algorithm, *a 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. In this experiment, 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. Hence, 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 knows 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. We use machine-I and all applications have 8 threads.

A. Matrix Multiplication

Figure 9 compares the performance of the `matrix multiplication` application in three different machine configurations (all-fast-I (8 @ 1.87GHz), all-slow-I (8 @ 1.6GHz), and half-half-I (4 @ 1.87GHz, 4 @ 1.6GHz)). LJFPF and round robin (RR) scheduling polices are used for half-half-I. All-fast-I and all-slow-I use RR.

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 is a *pthread join* function call at the end of the `matrix multiplication` application. Hence, when the workload is symmetric (300-300), the performance of half-half-I is slow-limited. However, when the application has strongly asymmetric characteristics (340-260, 350-250, 360-240), half-half-I with LJFPF actually performs as well as all-fast-I. In this case, the application is mainly limited by longer task threads. Longer task threads execute on fast processors on both all-fast-I and half-half-I, so all-fast-I and half-half-I show the same performance. Since, half-half-I consumes less power than all-fast-I, half-half-I consumes the least amount of energy among all three configurations for the 340-260, 350-250 and 360-240 cases. Therefore, we can conclude that for a strongly asymmetric workload, an AMP with LJFPF can save energy even compared to an SMP with fast processors.

B. ITK

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

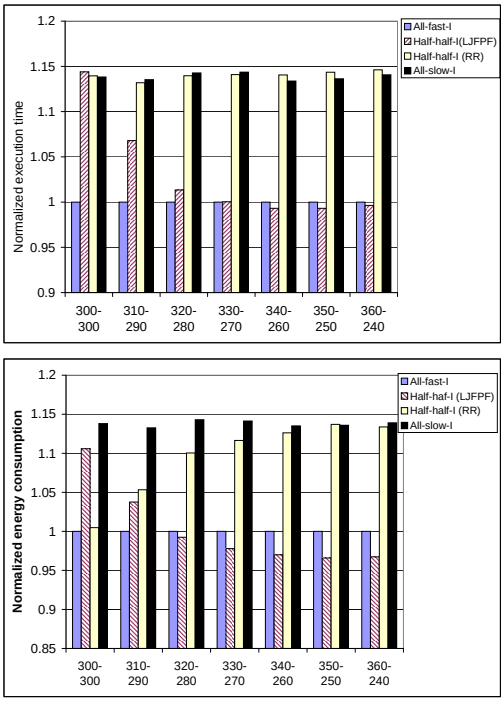

Fig. 9. The performance and energy consumption behavior in the `matrix multiplication` application with the LJFPF scheduling policy (top: execution time, bottom: energy consumption)

of the loop respectively. This division is done at compile time. Our scheduling algorithm (LJFPF), at run-time, sends all 7-iteration task threads fast processors and 6 and 5-iteration task threads to slow processors. Figure 10 shows the normalized execution time and energy consumption (all the data is normalized to all-fast-I). The results show that half-half-I with LJFPF performs as well as all-fast-I, and it also reduces energy consumption by 3.4% compared to all-fast-I. Hence, even in a real application like ITK, when the workload is asymmetric, using an AMP with our new scheduling policy (LJFPF) results in the best energy savings.

VII. Related Work

Many researchers have shown that asymmetric/heterogeneous multi-core/multiprocessor architectures can save power and energy [3], [13], [1], [15]. Since the focus of our work is evaluation of the behavior of multithreaded applications in a real system, we only discuss previous work that also use real heterogeneous/asymmetric systems.

Balakrishnan et al. [2] studied the performance behavior of multithreaded applications in an AMP using real systems. They also observed similar performance behavior on an AMP as us. We extend their work in two directions. First, we evaluate how thread interactions can affect performance and power in more detail. Second, we measure the system power and energy consumption, not only performance.

Annavaram et al. [1] studied energy per instruction (EPI) throttling ideas in AMPs. They measured basic power and

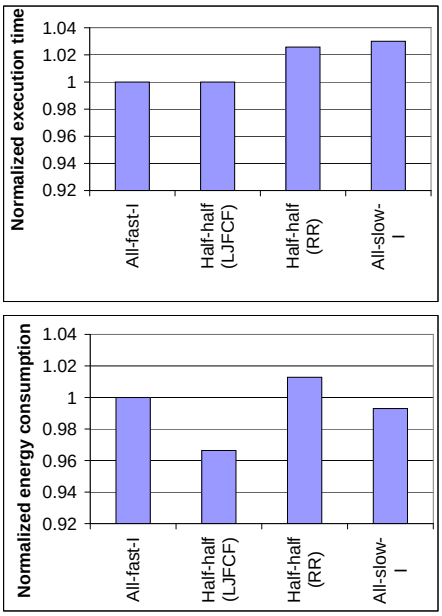

Fig. 10. The performance and energy consumption behavior of the ITK benchmark (top: execution time, bottom: energy)

performance data in real systems and used software simulations to predict the benefit of their throttling mechanism. Their work focused on dynamic voltage/frequency scaling mechanisms. However, our work focuses on understanding the effects of thread interactions in an AMP.

Li et al. [14] also measured the performance of AMPs by changing the clock frequencies. However, their work focused on proposing thread migration polices in AMPs, rather than understanding the performance/power behavior in AMPs.

Both Ge et al. [10] and Grant and Afashi [11] also used a real system to measure performance and power consumption behavior in AMPs. Both works presented only the trade-offs between power and energy consumption in multithreaded scientific applications whereas we evaluate thread interaction effects thoroughly.

VIII. Conclusion

In this work, we evaluate the performance and energy consumption behavior of desktop-oriented multithreaded applications in AMPs. We also evaluate the effects of critical sections and barriers thoroughly to understand thread interaction behavior on AMPs. We use real 8 and 16 processor systems to measure performance and energy consumption.

The conclusions of our experiments are that (1) when the workload is symmetric, it is usually better to use an SMP with fast processors than an AMP to reduce both the execution time and the energy consumption, (2) when an application has frequent and long critical sections, using an AMP could be better than using all fast processors to save energy, and (3) when the workload is highly asymmetric, using an AMP provides the lowest energy consumption.

We also propose and evaluate a new, simple scheduling algorithm for an AMP. The scheduling algorithm simply sends the longest thread to a fast processor. Using knowledge of the application and processor characteristics, this simple scheduling algorithm can reduce energy consumption by up to 4% on an AMP compared to the best energy efficient SMP configuration.

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

Acknowledgments

We thank Min Lee and Sushma Rao 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 Aemen Lodhi, Sungbae Kim and the anonymous reviewers for their comments and suggestions. This research is supported by gifts from Microsoft Research.

References

[1] M. Annavaram, E. Grochowski, and J. Shen, "Mitigating Amdahl's Law through EPI throttling," in *ISCA-32*, 2005.
[2] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai, "The impact of performance asymmetry in emerging multicore architectures," in *ISCA-32*, 2005.
[3] A. Baniasadi and A. Moshovos, "Asymmetric-frequency clustering: a power-aware back-end for high-performance processors," in *ISLPED*, 2002.
[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," Princeton University, Tech. Rep. TR-811-08, 2008.
[5] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, and J. McDonald, *Parallel Programming in OpenMP*. Morgan Kaufmann, 2000.
[6] C.-K. L. et al., "Pin: Building customized program analysis tools with dynamic instrumentation," in *PLDI*, 2005.
[7] "Extech 380801," http://www.extech.com/instrument/products/310_399/380801.html, Extech Instruments Corporation.
[8] H. Franke, R. Russell, and M. Kirkwood, "Fuss, futexes and furwocks: Fast userlevel locking in linux," in *Ottawa Linux Symposium*, 2002.
[9] GCC-4.0, "GNU compiler collection," http://gcc.gnu.org/.
[10] R. Ge, X. Feng, and K. W. Cameron, "Improvement of power-performance efficiency for high-end computing," in *IPDPS'05*, 2005.
[11] R. Grant and A. Afsahi, "Power-performance efficiency of asymmetric multiprocessors for multi-threaded scientific applications," in *IPDPS*, 2006.
[12] "Enhanced Intel SpeedStep Technology for the Intel Pentium M Processor–White Paper," Intel, March 2004.
[13] 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.
[14] 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.
[15] T. Y. Morad, U. C. Weiser, A. Kolodny, M. Valero, and E. Ayguad, "Performance, power efficiency and scalability of asymmetric cluster chip multiprocessors," vol. 5, no. 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 multi-threaded workloads on cmps," in *ASPLOS-XIII*, 2008.
[18] "OpenMP," http://openmp.org/wp/, The OpenMP Architecture Review Board.