Exploiting Unbalanced Thread Scheduling for Energy and Performance on a CMP of SMT Processors

Matthew DeVuyst, Rakesh Kumar, and Dean M. Tullsen

University of California, San Diego
Department of Computer Science and Engineering
La Jolla, CA 92093-0404
{mdevuyst, rakumar, tullsen}@cs.ucsd.edu

Abstract

This paper explores thread scheduling on an increasingly popular architecture: chip multiprocessors with simultaneous multithreading cores. Conventional multiprocessor scheduling, applied to this architecture, will attempt to balance the thread load across cores. This research demonstrates that such an approach eliminates one of the big advantages of this architecture – the ability to use unbalanced schedules to allocate the right amount of execution resources to each thread. However, accommodating unbalanced schedules creates several difficulties, the biggest being the fact that the search space of all schedules (both balanced and unbalanced) is much greater than that of the balanced schedules alone. This work proposes and evaluates scheduling policies that allow the system to identify and migrate toward good thread schedules, whether the best schedules are balanced or unbalanced.

1 Introduction

Prior work [15] has shown that system performance is sensitive to thread scheduling policies for simultaneous multithreaded (SMT) [21, 20] architectures. This will be even more true as the industry moves toward aggressive chip multithreaded architectures—containing multiple cores, each featuring multiple-thread execution. These architectures present both new opportunities and new challenges in achieving maximum performance out of the processor via effective thread scheduling.

The challenges arise from the fact that given a set of applications and a set of multithreaded cores, the space of possible schedules of threads to cores can be enormous—making it difficult to either predict or discover the best schedules. However, there are also tremendous opportunities. In this environment, we have much more control over which threads, and how many, are co-scheduled on cores. Even if we assume relatively balanced schedules (the same number of threads assigned to each core), we can select the groupings of threads assigned to each core so as to minimize negative interference between threads. But we show in this paper that we need not assume balanced schedules; and, in fact, the ability to create unbalanced schedules provides an important degree of freedom. This is an important result because conventional multiprocessor schedulers, applied to this architecture, will always seek to balance the number of threads on each core—we show that this is often the wrong decision.

Previous work [14] has shown that resource demands vary significantly between applications, and even between phases of the same application. Recently, heterogeneous (or asymmetric) multiple-core architectures have been shown to be effective at exploiting this phenomenon by mapping each job to the core that most closely matches the resource demands of the application [6, 7]. We can exploit the same principles in a CMP of SMT processors without the burden of hardware heterogeneity. In this case, the heterogeneity comes from the fraction of core resources made available to each thread. For example, consider a CMP where one core is already running two threads and another core is idle. In this case, a new thread could either be scheduled on the first core, providing low marginal performance but possibly expending even less marginal power, or it could be scheduled on the idle core, providing high marginal performance but resulting in high marginal power. Scheduling on the already loaded core may be best if the execution resource demands of the thread are low.

This paper examines system-level thread scheduling policies for a chip multithreaded architecture. Particular attention is paid to enabling performance and energy efficiency through unbalanced schedules. These schedules give the system the ability to cluster threads that have low execution demands and amortize the power cost of using a core.

Because the search space of possible schedules for such architectures is large, we rely on schedulers that learn from experience and migrate to the best schedules. They do so either through directed sampling or by making small adjustments to the current schedule (which is assumed to be good). Our studies cover many different degrees of thread-level parallelism. We consider schedules that leave cores idle, even when there are more threads than cores. This

1-4244-0054-6/06/$20.00

is particularly useful when energy and power are primary concerns. When both performance and power are first-class concerns for the scheduler, the nature of the best schedules become difficult to predict; thus, it becomes critical to have scheduling policies that dynamically adapt to the particular workload's execution behavior, and discover the right strategy.

This paper makes the following contributions. It studies, for the first time, a spectrum of scheduling policies for a chip-multithreaded architecture where both performance and energy are prime considerations. It shows that unbalanced schedules (uneven distribution of threads among the cores) often outperform balanced schedules—the best scheduling policies are those that consider both balanced and unbalanced schedules. We show that one can often get higher performance by clumping badly behaving threads together on the same core than by spreading them around. This is because such threads can interfere destructively with the otherwise high-performing threads. Running them with other low-performing threads is less likely to significantly impede those other threads. The benefits of unbalanced scheduling increase as the objective functions puts more emphasis on power efficiency.

Additionally, we demonstrate that intelligent non-sampling based scheduling policies can often outperform the policies that require sampling of the search space; this is significant because, given a moderate number of cores and a moderate number of threads, the search space for possible schedules can become large.

This paper also extends symbiotic scheduling [15] to a CMP of SMTs. Symbiosis-based random scheduling heuristics that performed well for a SMT core also perform well for a CMP of SMT cores but with smaller marginal gains. However, due to the much larger search space, we show that in this case there is even more gain to be had with more intelligent policies. Finally, we show that there are significant benefits to doing energy-aware scheduling. For 12 threads, it can result in up to 7.4% savings in energy, 10.3% savings in energy-delay product, and 35% savings in power. Savings are even greater with fewer threads.

The rest of the paper is organized as follows. Section 2 describes previous work related to thread scheduling and multithreading chip multiprocessors. Section 3 discusses the architecture that we are evaluating—a CMP of SMT cores. In section 4 we present our scheduling mechanisms and policies. We discuss our experimental methodology in section 5 and present and analyze the results in section 6. In section 7 we summarize our findings.

2 Related Work

While chip multithreaded processors are already on the market [2, 17], little research has been published on scheduling for such processors. Fedorova, *et al.* [3] examine scheduling for L2 cache miss rate on a CMP of SMT cores. They introduce an L2-conscious scheduling algorithm based on balance-set scheduling. They assume high thread level parallelism, and do not consider unbalanced scheduling. In this research, we focus on direct measures (i.e., performance, power, energy) rather than indirect—thus, if L2 miss rate is the dominant factor, we will migrate to schedules that minimize it (but perhaps more slowly). If there are other important factors, we will find better schedules. We also assume more aggressive cores.

Powell, *et al.*[5] seek to alleviate the power density problem, a condition found in many modern processors and potentially aggravated by a CMP/SMT architecture. Their work leverages the ability to schedule on two levels: inter-core and intra-core. Because their goal is very different, the proposed scheduling solution, Heat-and-Run, is not effective for reaching our goals. Heat-and-Run moves jobs off of a core (to a cooler one) before the core can exceed thermal thresholds. In a sense, they employ unbalanced schedules to favor cool cores, and let hot cores cool down.

Also, while Sasanka, *et al.* [9] do not discuss scheduling policies, they show analytically as well as through quantitative evaluations that a CMP of SMT cores is more energy efficient than a CMP or an SMT processor. In Section 6 we confirm their observation and use that as a basis for our energy-aware scheduling policies.

Significantly more work has been done on scheduling for SMT or CMP processors. Snavely, et al. [15] propose a job-scheduler for SMT architectures. They introduce the concepts of symbiosis (discovering jobs that run together with minimal interference) and sampling-based scheduling that are used in this research. While the kind of co-scheduling in their work is temporal, the co-scheduling in our research is spatial—co-scheduling symbiotic jobs on different cores. Also, they do not consider power or energy. Other notable work on SMT scheduling includes [12, 8, 16].

Scheduling for homogeneous CMPs (of single-thread cores) is trivial if all cores are identical and interference is only at the L2 level and beyond—though interesting issues [13] can arise with communication, data locality, etc. Heterogeneous CMPs, on the other hand, require a careful mapping of applications to cores. Recent work [6, 4] addresses some of the scheduling issues with such processors. Kumar, *et al.* [7] consider the scheduling problem for a heterogeneous multi-core architecture where a few cores are multithreaded. Again, their objectives are power-oblivious.

In this paper, we consider scheduling for a CMP of SMT cores for objectives that are a composition of both power and performance.

3 Architecture

We focus on a single hardware architecture but evaluate it under different constraints and different levels of thread parallelism (different loads). This architecture is large enough to make scheduling a complex problem; and we believe the principles exposed by our results will scale to larger configurations.

The architecture is a chip multiprocessor consisting of four homogeneous simultaneous multithreaded [21] cores implemented in the $0.1\mu m$ process with shared L2 and L3 caches. Each core has its own L1 data cache and L1 instruction cache. Because threads can migrate between cores, they will exercise the cache coherence policy to correctly

Cores	4	TLB miss penalty	790 cycles
Contexts per core	4	I cache size	32k
Reorder Buffer entries	256	I cache miss penalty	8 cycles
Active List entries	512	I cache associativity	4
Total fetch width	4	D cache size	32k
Integer registers per core	100	D cache miss penalty	8 cycles
FP registers per core	100	D cache associativity	4
shared L2 cache size	2 MB	shared L3 cache size	4 MB
L2 miss penalty	40 cycles	L3 miss penalty	315 cycles
L2 access time	12 cycles	L3 cache access time	35 cycles

Table 1. Architecture Detail

handle dirty data in the L1 data caches. When a thread migrates to another core and requests data that was dirty in the L1 data cache of the core it was previously running on, the dirty data will be written to both the L1 cache of the core on which the thread is now running and the shared L2 cache. The implementation is assumed to be at 2.1 GHz and latencies are determined accordingly.

There are four contexts per SMT core, for a total of 16 contexts in the system. Each SMT core is out-of-order. There exist two dimensions of thread level parallelism (TLP) in our architecture as there are multiple cores, each with multiple contexts. In the first dimension of TLP, threads co-scheduled in different contexts on the same core share many core resources including a register file, hardware queues, and a set of functional units. In the second degree of TLP, threads scheduled on different cores share fewer resource (such as the more distant levels of the memory hierarchy).

In this work, we assume that unused cores are completely powered down, rather than left idle. Thus, unused cores suffer no static leakage or dynamic switching power. This does, however, introduce a latency for powering a core on or off. In [6], it is estimated that a given processor core can be powered on in approximately one thousand cycles of the 2.1 GHz clock. They assume that when a processor core is powered down, the phase-lock loop that generates the clock for the core is not powered down. Rather, the same phase-lock loop generates the clock for all cores. Consequently, the power-up time of a core is determined by the time required for the power buses to charge and stabilize. However, in this work we did not want to assume constraints on power supply or PLL design, so we perform all experiments assuming that it takes $30\mu s$ to turn on or off a core. We estimate this time based on reported data on PLL stabilization times and system overheads. Note that this is over 60 times more conservative than was assumed before (in [6]), and it allows plenty of time for dirty data to be flushed from the caches, as well. We also assume, conservatively, that a core keeps dissipating idle power at a steady rate until it is completely powered off and also during the entire power-on procedure. We do not consider dynamic voltage scaling (DVS) in this work.

Table 1 contains more details of the architecture.

4 Scheduling Policies

We assume an operating system level thread scheduler that makes global (CMP-wide) scheduling decisions. Thus, scheduling decisions must be made at a very coarse granularity, and the cost of moving a thread is very small relative to the typical interval between moves. The processor typically makes scheduling decisions based on sampled data of processor power and performance. Processor power and performance can be estimated by reading counter registers that are already found in most modern processors. New samples are collected and new scheduling decisions can be made as often as every operating system timer interrupt (although, in general, we strive to change schedules much less often than the timer interrupt) or when the mix of jobs changes.

We assume all jobs have equal priority. Differing priorities could significantly increase the utility of unbalanced schedules, thus increasing the importance of scheduling algorithms that consider such schedules. However, the correct metrics to evaluate the goodness of our schedules are less clear in the presence of priorities, so we do not demonstrate that advantage in this work.

The scheduling policies we evaluate are described below. They come in two broad categories: *sampling-based policies* and *electron policies*. The electron polices are so called because the state of a core may cause it to either try to push a job away or to attract new jobs.

4.1 Sampling-based Policies

The sampling-based policies work in a series of alternating temporal phases: a sampling phase and a steady phase. During the sampling phase, a number of different schedules are tried; at the end of the sampling phase the sample schedule with the best *metric value* is chosen as the schedule to be in effect throughout the steady phase. The *metric value* depends on what we wish to optimize for, whether it be power, energy, energy delay product, performance, or something else. In addition to the sample schedules, the schedule in place during the last steady phase is also a candidate for selection—this ensures that in the ideal (but somewhat rare) case with no noise or phase behavior, we will always move toward better schedules. In all cases, we create 10 sample schedules, which we found to create a good balance between the desire to maximize our chances of finding a good schedule and to minimize the ratio of the duration of the sample phase to the duration of the steady phase. Each sample runs for two time slices, and the measurement takes place on the second, to eliminate cold start effects. The steady phase then runs 10 times as long as the sample phase.

While the creation of schedules is geared towards intelligently navigating through the permutation space, selecting the best schedule involves using the right metric to characterize the goodness of a schedule. The sampling policies, then, are generic and can be specialized to different objective functions by changing the evaluation metric. The evaluation metrics used to choose the best sampled schedule are described in Section 4.3.

We consider the following sampling-based scheduling policies:

Balanced Symbiosis It has been shown previously that some groups of applications run well together, while others result in destructive interference [19], causing individual

applications to slow down. This property can be used to enhance system performance by scheduling "friendly" threads together on one core. Threads are randomly assigned to contexts for each sample schedule. The only constraint, in this case, is that the number of threads scheduled on each core is the same, or within one if the threads don't divide evenly onto cores. At the end of the sampling phase, whatever decision metric has been chosen will be used as the criterion for selecting a schedule for the steady phase. This policy is closest to the symbiotic job schedulers described by Snavely, et al [15] for a single SMT core (those schedules were "balanced" in the sense that they always used all thread contexts). Because this scheme only considers balanced schedules, most schedules considered are reasonable ones—but potentially better schedules that are not balanced will never be considered.

Symbiosis This policy is similar to the prior one, except that it does not constrain the threads to be assigned evenly among the cores. As a result, it has the potential to find better schedules than the first policy can find; but it also has the potential to sample a larger number of clearly bad schedules.

Balanced Random This policy chooses a random, but balanced, schedule with no sampling. This is the closest approximation to what a conventional load-balancing multiprocessor scheduler would do, and is the baseline in many of our graphed results.

Prefer Last This is a class of policies that assume that the configuration we are running now has merit, and the next configuration will be similar. In this case, sampling is biased towards a similar configuration with only a fraction (30%) of the schedules deviating from that. The different forms of *prefer last* are described below—they differ in how we define "similar" schedules. With these policies, we can apply an intelligent bias to the sampling (e.g., use all cores, use few cores)—but the bias is not hard-coded in the scheduler; the bias is derived from recent history. *Prefer last* policies are introduced in [7]; however, we look at many more flavors and apply them in a different context.

Prefer Last – Numbers A new schedule is similar to the original schedule if it runs the same number of threads on each core as the original did. However, the particular assignment of threads is done randomly. This policy seeks to retain the same level of schedule imbalance, but does not strive to favor or co-schedule the same threads.

Prefer Last – Swap A similar schedule is created by choosing two threads (on different cores) from the original schedule to swap places. This policy evolves slowly both in the number of threads assigned to cores and the composition of threads co-scheduled.

Prefer Last – Move In this variant, a similar schedule is created by randomly choosing one thread and moving it to a randomly-selected empty context on another core and leaving all other threads in place. This policy tends to preserve the sets of threads co-scheduled, but allows the distribution of the load (in number of threads) to evolve more quickly. Note that this policy does not work in this exact form for very high system loads – i.e., when all the contexts are saturated.

All these policies are evaluated assuming that any unused cores are power gated. When a sampling phase begins, all the samples to be tried out in that sampling phase are created and then ordered by the number of cores they utilize. The schedules that use the same number of cores as were used in the last steady phase are tried first. All the schedules that use a different number of cores are grouped together by the number of cores they use and are ordered such that all the sample schedules using the same number of cores are tried consecutively. This minimizes the number of power gating changes that are made.

4.2 Electron Policies

The policies in the previous section rely on sampling for intelligent scheduling. However, such policies become less effective as the search space expands.

Rather than sampling, the electron policies rely on more explicit evidence that a particular core is over-scheduled, under-scheduled, or just poorly scheduled. Cores will attract threads that fill a void and repel threads when contention is high. Threads will move around each interval to create a better fit. The schedule naturally adapts as threads enter new phases of execution.

Following is the description of electron policies customized for each metric that we study.

Electron – Performance This policy assumes that utilization of a core's resources has a correlation with performance. The IPC of each core in the previous period is calculated. The core with the highest aggregate IPC repels one thread, and the core with the lowest IPC attracts a thread. If the latter has a free context, the thread is transferred. If the condition is not met, we do not change the schedule. Ninety percent of the time the selection of the thread to repel away from the high IPC core is the highest IPC thread on that core, and ten percent of the time a random thread from that core is selected. The thread selection policy is based on the assumption that the highest IPC thread has the most aggressive resource requirements and hence needs to find a core with the least current utilization. Occasional random selection guards against the same thread infinitely hopping from core to core.

Electron – Energy The objective here is to minimize the overall energy consumption of the processor. The energy of each core in the previous period is calculated. The (non-idle) core with the lowest energy repels one (randomly chosen) thread, and the core with the highest energy attracts a

thread. If the former was not idle and if the latter has a free context, then the thread is transferred; otherwise the schedule does not change. Over time, this policy tends to cluster threads so that more cores are left idle, then distributes the jobs efficiently among those cores.

Electron – EDP This policy tries to minimize overall energy-delay product of execution by identifying cores under-performing on this metric. The energy delay product of each core in the previous period is calculated. The core with the highest EDP attracts a thread, which is supplied from the core with the lowest EDP—the thread to move is chosen randomly. If the latter was not idle and if the former has a free context, the thread is transferred; otherwise the schedule does not change. If a core was idle (meaning it consumed no power but completed no instructions) its EDP is considered to be infinite. This assumption discourages idling of cores—we found that leaving a core idle (when EDP is the targeted metric) is usually inefficient.

Note that making locally good decisions for a metric like EDP does not guarantee making globally good decisions [10].

For the electron policies, the duration of a period is moderately long (i.e., a schedule change can occur at most once every 4 O/S time-slices in our experiments). If a new schedule results in a core being left idle, that core is powered down immediately. For all the policies, at every scheduling change, if the new schedule calls for cores to be powered on, the required cores are powered on before the new schedule takes effect. Also, if the new schedule calls for cores to be either powered on or powered off, the scheduler does not sample any performance or power counters during the transition, so as not to skew the statistics associated with the new schedule.

Note that electron schedulers run into the risk of continuing to alternate among two schedules. To help avoid such situations, history information has been incorporated into the scheduling policy decisions. A history is maintained of the last 10 schedules run (duplicate schedules included) and the associated performance and power. If the schedule that the electron policy proposed is found in the history, then the scheduler knows about how well it will perform. In such cases, it will select the "best" schedule among all the schedules in the history, where the "best" schedule is the one found to be most efficient with regard to the desired decision metric. If the proposed schedule is not found in the history, its performance is unknown, so it will be run. Because the history length is finite and contains previously used schedules regardless of their uniqueness among the other history entries, the electron scheduling policies will be able to adapt to workload phase changes. Stale performance data is evicted from the history and previously applied schedules are eventually applied again.

4.3 Decision Metrics

This section describes the metrics used to select from among the sampled schedules discussed in Section 4.1. The objective function during scheduling may be static for a given environment and a given market segment. In other cases, it may change dynamically as the processor changes power conditions (e.g., plugged vs. unplugged, full battery vs. low battery, thermal emergencies), as applications switch (e.g., low priority vs. high priority jobs), or even within an application (e.g., a real-time application is behind or ahead of schedule).

We consider four system-level objective functions in this paper—performance, power, energy, and energy-delay product—and tune the evaluation metric for those objective functions in the following ways.

Performance The metric for performance is *calculated weighted speedup* (CWS). This is derived from the weighted speedup metric proposed in [15], and used in this paper to evaluate simulated performance (described in Section 5). However, it cannot be applied in the same way at runtime without oracle information—this is because it depends on knowing how the program would run on a baseline configuration (e.g., single-threaded). Thus, we define each application's "average performance", for the purpose of calculating CWS, as the average of its performance over a number of sampled configurations. Thus, for the runtime scheduler, one thread's contribution to the total calculated weighted speedup is its IPC for that sample, divided by its average IPC for all samples. We found this to provide better performance than using IPC as the choice metric. With IPC, causing one thread's throughput to drop from 4 to 2 is twice as bad as causing another thread to drop from 2 to 1. With CWS, they are considered equally bad—this represents a more system-level view of performance which says that a 2X slowdown in any application is considered bad, regardless of the raw IPC.

Energy Delay Product The energy delay product (EDP) of the processor, which is $PowerPerCycle/IPC^2$, is computed for each of the sampled schedules. The schedule with the lowest EDP is considered the best.

Energy The energy of the processor, which is $PowerPerCycle/IPC$, is computed for each of the sampled schedules. The sample with the lowest energy is considered the best. Performance (IPC) is still a factor in the energy equation, because faster execution consumes power over a shorter time-span.

Power The total power is computed at each sampling schedule. The total power is the sum of the power of all the cores plus the power of the shared structures (L2 and L3 caches). The schedule with the lowest power is considered the best.

5 Experimental Methodology

In this section, we discuss the various methodological details of our evaluation framework and scheduling mechanisms.

5.1 Scheduling Parameters

Our thread scheduler is assumed to be a part of the operating system; it makes scheduling decisions based on sampled data of processor power and performance. New samples are collected and new scheduling decisions can be made at every operating system time-slice interval. We have assumed an operating system time-slice of a quarter million cycles. This time-slice is artificially short to keep our simulations from taking too long. It allows us to model a larger number of sample/steady intervals per simulation, and is still long enough to capture interesting application phases for most of our benchmarks. This enables our scheduling policies to be evaluated and compared based on how quickly and accurately they can adjust to the changing workload behavior. We also performed a few experiments with larger intervals and found no significant difference in results or trends. The time-slice interval durations we used are also significantly longer than any lingering cold-start artifacts of the simulation methodology.

5.2 Workload Construction

Twelve benchmarks from the SPEC 2000 benchmark suite were chosen to construct workloads for our evaluation. Sub-setting was done such that the fraction of compute and memory bound benchmarks is the same as that in the *entire* SPEC suite—so the subset is representative of the entire suite. Each benchmark was fast-forwarded for 2 billion instructions before detailed simulation. Table 2 contains a list of the twelve benchmarks. All simulations use the reference data sets.

We performed all our evaluations for various values of available thread-level parallelism for multiprogramming workloads. For each level of thread-level parallelism, we construct and use eight workloads by randomly selecting eight different subsets of the 12 benchmarks. These groups are formed such that the contribution of a benchmark to the result remains the same across different TLPs, similar to the sliding window methodology typically used in SMT research [21]. For the 12-thread experiments and higher, at most only one group could be constructed if we didn't allow duplicate threads. Thus, multiple instances of randomly-selected application(s) are run in a single group. Table 3 lists the workloads used for our study. In all the results reported in this paper, the results were obtained by averaging the statistics across the 8 groups.

5.3 Simulation Methodology

All our simulations are done using a chip-multi-threaded multiprocessor derivative of SMTSIM [18]. The simulator supports MESI coherence protocol and executes statically linked Alpha binaries. Appropriate modifications were done to simulate the effect of OS-level scheduling as well as the availability of hardware counters. To gather power statistics we integrated a modified version of Wattch [1] into our simulator. Wattch was modified to collect, calculate, and report power statistics on a multi-core architecture with a shared L2 cache. We made use of Wattch's conditional clocking power model (labeled `cc3` in Wattch source code). We also introduced core-level power gating into the model. The modeling details for power gating are discussed in Section 3.

Benchmark	Code	Description
gap	0	Interpreter
fma3d	1	Crash simulator
mesa	2	3D graphics
equake	3	Wave simulator
crafty	4	Chess game
wupwise	5	Quantum Chromodynamics
mgrid	6	Multi-grid solver
gzip	7	Compression
gcc	8	Compiler
apsi	9	Meteorology
swim	A	Shallow water modeling
ammp	B	Chemistry

Table 2. ***Benchmarks*****: Each benchmark is labeled with a code (used in Table 3) and a brief description**

4a	8165	6a	8165B0	8a	8165B072
4b	6359	6b	635924	8b	6359240B
4c	A960	6c	A96048	8c	A9604851
4d	5879	6d	5879B1	8d	5879B143
4e	A23B	6e	A23B79	8e	A23B7968
4f	4230	6f	423076	8f	423076A9
4g	47A8	6g	47A8A1	8g	47A8A10B
4h	1B20	6h	1B2035	8h	1B20354A

12a	8165B07284A3	16a	8165B07284A36359
12b	6359240B38A1	16b	6359240B38A1A960
12c	A96048516723	16c	A960485167235879
12d	5879B1435062	16d	5879B1435062A23B
12e	A23B79689514	16e	A23B796895144230
12f	423076A97B51	16f	423076A97B5147A8
12g	47A8A10B0962	16g	47A8A10B09621B20
12h	1B20354AB879	16h	1B20354AB8798165

Table 3. ***Workloads*****: The first part of each pair is the workload label. The second part of each pair encodes the benchmarks that form the workload—each digit in this number is the code of a particular benchmark. There are workloads consisting of 4, 6, 8, 12, and 16 threads.**

6 Analysis and Results

This section presents the effectiveness of the scheduling policies that adapt to varied program behavior and consider both balanced and unbalanced schedules. We pay particular attention to the case where both performance and energy are important and provide more detailed results for that case. We also examine the various policies for objective functions specific to energy, power, and performance.

6.1 Scheduling for Both Energy and Performance

Scheduling for both power and performance at the same time presents an interesting challenge for a CMP of SMT cores. The marginal performance achieved by using an additional core in a CMP of SMTs is typically higher than the marginal performance improvement from using an additional SMT context on the same core. Also, the marginal performance improvement from an SMT context continues

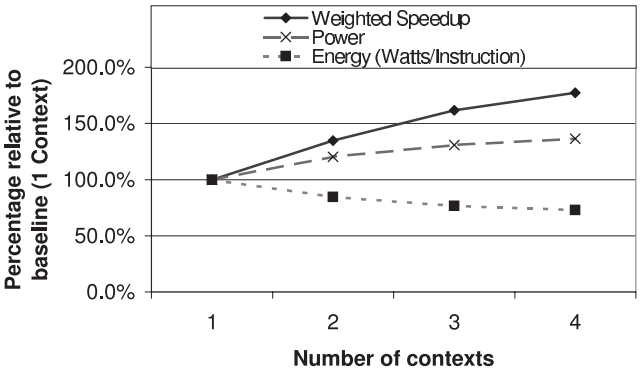

Figure 1. Average marginal utility and cost of using each SMT context on a single core.

to decrease as the number of threads increases [20]. So, it is often better to schedule as few applications on each SMT core as possible if scheduling only for performance—threads spread out across cores. On the other hand, the converse is true for energy. That is, energy efficiency increases with the number of contexts in operation [11], so we tend to aggregate threads when scheduling for energy. Figure 1 shows how power and performance vary with the number of contexts for our processor model. We see that, on average, marginal performance drops off as we add threads, and is typically much less than the performance of using a second core. Conversely, energy efficiency is maximized as we add threads to a single core. Scheduling for a CMP of SMTs such that both power and performance are optimized, then, requires a careful balance between these two competing objectives.

We perform all our evaluations in this section using the energy-delay product (EDP) metric. EDP recognizes the importance of both power and performance and is used widely as an important objective function for desktop as well as server processors.

Unbalanced Scheduling Schedulers for traditional multiprocessors seek to evenly distribute the system load over the available processing contexts. Such schedulers constrain the schedules to be balanced, limiting their flexibility to optimize for multiple competing objectives at the same time.

If our scheduler allows unbalanced scheduling, we have the opportunity to meet both of these objectives—we can aggregate jobs that have low execution resource demands for energy efficiency, while still giving more resources (e.g., on other cores) to those jobs that demand them for performance.

Figure 2 compares an optimal static scheduling policy (*Static Ideal*) that allows unbalanced scheduling against the best static scheduling policies that are constrained to be balanced. *Static Balanced* ensures that each core runs the same number of threads (or within one if the number of threads is not evenly divisible by the number of cores), and, hence, is similar to the traditional load-balancing schedulers. *Static Cluster Balanced* ensures that only as many cores as necessary to run a given number of threads are kept on and the rest are power gated; among the active cores, each core runs the same number of threads. This policy minimizes the system power consumption at the expense of performance. The results are shown for different levels of thread-level parallelism. Thus, if we have six threads, *Static Balanced* will only consider schedules of threads to the four cores like 2,2,1,1; and *Static Cluster Balanced* will only consider schedules like 3,3,0,0.

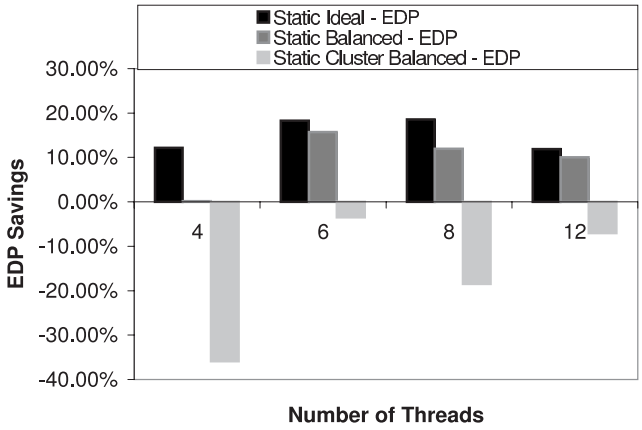

Figure 2. The effectiveness of unbalanced and balanced static scheduling policies in reducing energy-delay product. Results are presented as fraction of EDP savings relative to *Balanced Random***.**

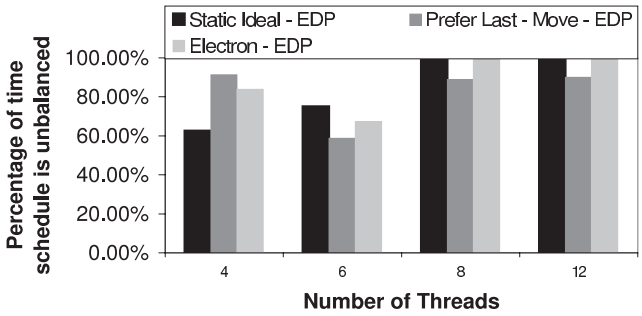

Figure 3. Extent of imbalance for the best schedules found by static policies targeting EDP

Figure 2 shows that there is a significant advantage to doing unbalanced scheduling—*Static Ideal* results in consistently higher EDP savings than the best balanced scheduling policy (*Static Balanced*). Savings are 12% higher for 4 threads and 6.6% higher for eight. As cores become more heavily saturated with threads, the flexibility for doing intelligent unbalanced scheduling decreases and the relative benefits decrease.

While these results are averaged over eight workloads, we observed that *Static Ideal* often resulted in unbalanced schedules (see Figure 3, black bar—the other bars are discussed later). For example, for 4 threads, five out of

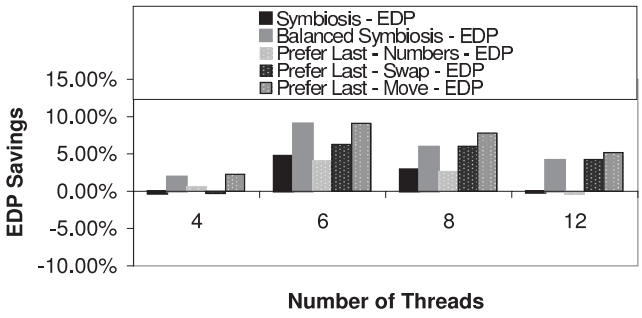

Figure 4. Energy-Delay product for sampling-based scheduling policies

eight workloads were scheduled in an unbalanced manner in the ideal case. For 6 threads, this number was six out of eight. All the best static schedules for 8 and 12-threaded workloads were unbalanced. We also observe that most of the schedules are unbalanced even for the best dynamic scheduling policies discussed in the following sections.

The advantages due to unbalanced scheduling depend on the characteristics of the workload. Benchmarks *gcc* and *gzip* are averse to running with other applications due to high ICache working set sizes and high core utilization, respectively. Balanced scheduling policies force these applications to be co-scheduled with some other thread on the same core resulting in a significant performance hit. However, *Static Ideal* allows these applications to be on a core by themselves resulting in high overall efficiency. On the other hand, we find that *ammp*, *swim*, and *apsi* are often co-scheduled, as they have lower inherent ILP and see minimal destructive cache interference. Workload 8e (refer to Table 3) contains both *gcc* and *gzip*, and the best unbalanced schedule has 14% lower EDP than the best balanced schedule.

The graph also makes a case for energy-aware scheduling. *Static Ideal* results in more than 12.1% EDP savings for 4 threads and 18.5% savings for 8 threads. In fact, we observed that the policy can result in savings of 8.7% even for 16 threads (not shown in the graph) over a naive schedule that only seeks to balance the load.

The static results are ideal, identified by exhaustive search. The next section presents realistic dynamic scheduling policies.

Exploring the Search Space through Directed Sampling Sampling-based scheduling policies try to adapt to the changing workload behavior by sub-setting the search space and then making the best choice of schedule from among the reduced space. The effectiveness of a scheduling policy is hence primarily determined by how effectively it does the sub-setting. Figure 4 compares the various sampling-based scheduling policies. Results are shown for both symbiosis-based policies as well as *Prefer Last* policies. The results are shown for various levels of thread-level parallelism and with *Balanced Random* policy as the baseline.

The graph leads to several interesting observations. First, there are again significant benefits to doing energy-aware dynamic scheduling. The best sampling-based policy (*Prefer Last – Move*) results in 2.3% EDP savings for 4 threads, 7.8% savings for 8 threads, and 8.3% savings for 16 threads (16 thread case not shown in graph). These benefits are achieved through unbalanced scheduling of threads to cores. For example, 91% of the schedules chosen by the *Prefer Last – Move* were unbalanced for 4 threads. The percentage of unbalanced schedules was 59%, 89%, and 90% respectively for 6, 8, and 12 threads. Figure 3 shows the fraction of unbalanced schedules for other policies as well.

The results show that symbiosis-based scheduling policies that performed well for an SMT core [15] also perform well for a CMP of SMT cores. The best symbiosis-based policy (*Balanced Symbiosis*) results in 1.9% EDP savings for 4 threads and 5.9% savings for 8 threads. The continued benefits due to symbiosis-based policies can be explained by the need to co-schedule "friendly" threads together even on a chip multi-threaded processor. A more surprising result is *Balanced Symbiosis* outperforming *Symbiosis*. Although *Symbiosis* has the freedom to try out both balanced as well as unbalanced schedules, it also has a greater likelihood of sampling bad schedules for a given number of samples. *Balanced Symbiosis*, on the other hand, is conservative and only samples reasonably good (balanced) schedules and converges on better schedules more quickly. This is particularly true with 6 threads where even the balanced scheduler is allowed (forced) to sample moderately unbalanced schedules.

Another significant observation is that the policies that learn from the makeup of the current configuration (*Prefer Last*) can result in high overall system efficiency. In fact, the best *Prefer Last* scheduling policy (*Prefer Last – Move*) outperformed the best symbiosis-based policy (31% higher savings for workloads of 8 threads). This is due to a more targeted sub-setting of the search space by the *Prefer Last* policies. *Prefer Last – Numbers* emerges as the weakest sampling-based policy because it does not preserve co-schedule relationships and changes the number of threads per core slowly. *Prefer Last – Swap* preserves co-schedule relationships, but also changes the distribution (in number of threads) slowly. *Prefer Last – Move* outperforms both the above policies as it not only preserves the sets of threads co-scheduled, but also allows the distribution of the load to evolve more quickly.

Non-sampling Strategies The previous policies rely on sampling for intelligent scheduling. However, such policies get increasingly less effective as the assignment space gets larger; for a given machine, the size of the assignment space increases with the number of threads that need to be scheduled. The sampling strategies also experience an overhead, as the sampling intervals will have lower performance than the steady intervals.

Figure 5 shows the results for the *Electron EDP* scheduling policy. The electron policies rely on more explicit evidence that a particular core is over-scheduled, under-

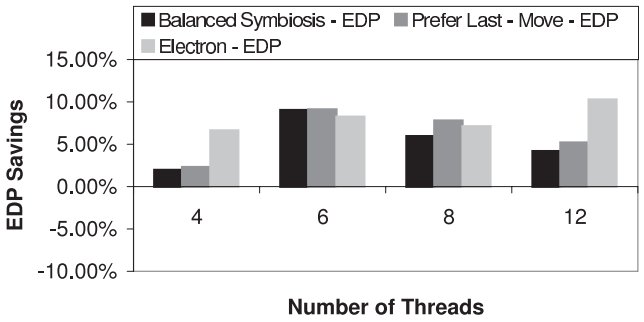

Figure 5. Effectiveness of the non-sampling electron policy, compared to two sampling policies.

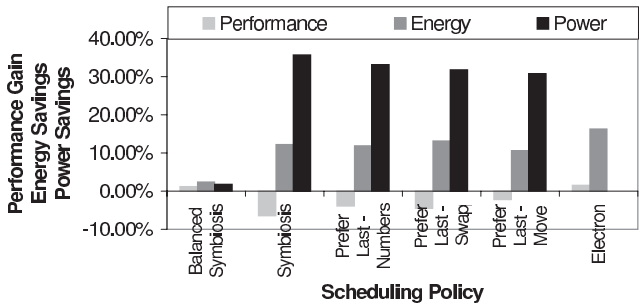

Figure 6. The impact on performance, energy, and power of various thread scheduling policies.

scheduled, or just poorly scheduled. This is especially useful when the assignment space is too large to sample effectively. In fact, as we can see, the electron policy outperforms all the sampling-based policies when thread-level parallelism is high. EDP savings are twice that of *Prefer Last – Move* and 2.4 times *Balanced Symbiosis* for 12 threads. While the sampling strategies struggle with the size of the search space, the fact that the electron policies incur no sampling overhead allows those policies to adapt much more often and navigate the large schedule space more effectively.

We also observed that the best schedules for *Electron EDP* are unbalanced. For 12 threads, for example, 100% of the schedules are unbalanced. 83% and 99% of the schedules are unbalanced for 4 and 8 threads respectively. This again confirms the usefulness of providing more flexibility to the scheduler.

6.2 Scheduling for Other Metrics

This section discusses scheduling for other scenarios where it is less critical to have both low energy and high performance at the same time. These scenarios differ from the previous section in that at least the shape of good schedules is more predictable. However, we find that even in these cases, directed scheduling policies still enable us to find specific schedules that better use the available resources and group threads in ways that minimize negative interference.

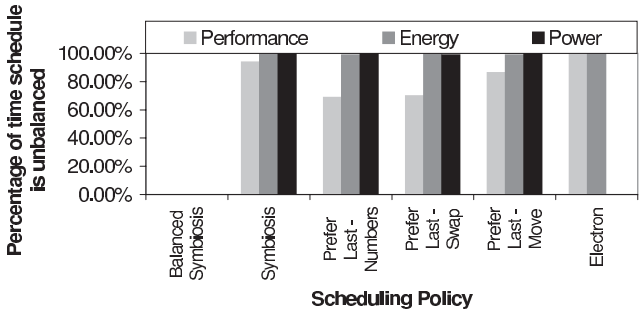

Figure 7. Extent of imbalance in the dynamic schedules targeting the various metrics.

Figure 6 compares various scheduling policies directed towards performance (measured as weighted speedup), energy, and power (using 8-thread workloads). What we find when targeting only performance is that there is still something to be gained from considering unbalanced schedules (see Figure 7), but in general there is less to be gained from these schemes. Simply balancing the system load evenly over compute nodes is often a sufficient mechanism for extracting good performance. What gain there is comes primarily from finding symbiotic schedules that are better than the random groupings.

For the power-related metrics (energy and power), we see that significant benefits can be had with directed scheduling policies. Note that no power bar is shown for the electron policy because no electron policy was optimized for power (since such a policy would be trivial). *Electron Energy* emerges as the best scheduling policy for energy. It results in over 15% energy savings—high gains can be attributed to the push/pull behavior trying to cluster threads intelligently on as few cores as possible. Other scheduling policies that allow unbalanced scheduling also result in over 10% savings. *Balanced Symbiosis* fails to achieve significant savings because it is constrained to utilize all the cores all the time. Scheduling for power exposes the value of unbalanced scheduling even more. *Balanced Symbiosis* results in less than 2% power savings. Policies that allow unbalanced scheduling can lead to more than 35% power savings.

To test the effect that core power gating has on our power-aware scheduling policies (including EDP), we did some experiments with power gating turned off. We observed two key differences. First, the power savings are markedly less when power gating is not used as ungated idle cores continue to consume power. Second, scheduling policies that consider both power and performance find less incentive to save power by leaving cores idle and thus produce more balanced schedules.

7 Conclusions

A chip multi-threading architecture, with multiple SMT cores on chip, has a unique ability to partition distributed execution resources to each application according to its individual needs. But this requires appropriately assigning

threads to cores, as the execution resources available to a thread depend on how many threads are assigned to the same core and exactly which threads it shares the core with.

A traditional multiprocessor scheduler, applied to this architecture, will not identify the best schedules. To do this, a good scheduler must be able to explore both balanced and unbalanced schedules. Additionally, it must be able to distinguish between good co-schedules and poor co-schedules and to navigate the huge space of potential schedules to continuously evolve toward better schedules.

This paper proposes several operating system level thread scheduling policies for such an architecture. These adaptive policies are particularly critical when both performance and energy are first-class concerns. In that scenario, neither distributing threads across cores nor aggregating threads on few cores is clearly the best policy, but the right schedules are workload-dependent and can only be identified by policies that adapt dynamically to the current program behavior. We show gains, versus a random scheduler that always uses balanced schedules, of 6-11% in energy-delay product. We also observe gains when scheduling for pure energy, pure performance, or power.

Acknowledgments

The authors would like to thank the reviewers for their feedback and Jeff Brown for his assistance with the simulator. This research was funded by NSF grant CNS-0311683, Semiconductor Research Corporation grant 2005-HJ-1313, grants from Intel, and an IBM Fellowship.

References

[1] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *International Symposium on Computer Architecture*, June 2000.

[2] J. Clabes, J. Friedrich, M. Sweet, J. DiLullo, S. Chu, D. Plass, J. Dawson, P. Muench, L. Powell, M. Floyd, B. Sinharoy, M. Lee, M. Goulet, J. Wagoner, N. Schwartz, S. Runyon, G. Gorman, P. Restle, R. Kalla, J. McGill, and S. Dodson. Design and implementation of the POWER5 microprocessor. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 670–672, 2004.

[3] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum. Performance of multithreaded chip multiprocessors and implications for operating system design. In *USENIX 2005 Annual Technical Conference*, Apr. 2005.

[4] S. Ghiasi and D. Grunwald. Aide de camp: Asymmetric dual core design for power and energy reduction. Technical report, University of Colorado, Boulder, 2003.

[5] M. Gomaa, M. D. Powell, and T. N. Vijaykumar. Heat-and-run: leveraging SMT and CMP to manage power density through the operating system. *SIGPLAN Not.*, 39(11):260–270, 2004.

[6] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, 2003.

[7] 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 *International Symposium on Computer Architecture*, June 2004.

[8] S. Parekh, S. Eggers, and H. Levy. Thread-sensitive scheduling for SMT processors. Technical report, University of Washington, Apr. 2000.

[9] R. Sasanka, S. V. Adve, Y.-K. Chen, and E. Debes. The energy efficiency of CMP vs. SMT for multimedia workloads. In *ICS '04: Proceedings of the 18th annual International Conference on Supercomputing*, pages 196–206, 2004.

[10] Y. Sazeides, R. Kumar, D. M. Tullsen, and T. Constantinou. The danger of interval-based power efficiency metrics: When worst is best. In *Computer Architecture Letters, Vol 4*, Jan. 2005.

[11] J. S. Seng, D. M. Tullsen, and G. Z. Cai. Power-sensitive multithreaded architecture. In *Proceedings of International Conference on Computer Design*, 2000.

[12] A. Settle, J. L. Kihm, A. Janiszewski, and D. A. Connors. Architectural support for enhanced SMT job scheduling. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 63–73, 2004.

[13] K. A. Shaw and W. J. Dally. Migration in single chip multiprocessors. In *Computer Architecture Letters, Vol 1*, Jan. 2002.

[14] T. Sherwood, E. Perelman, G. Hammerley, and B. Calder. Automatically characterizing large-scale program behavior. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[15] A. Snavely and D. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading architecture. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.

[16] A. Snavely, D. M. Tullsen, and G. Voelker. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In *SIGMETRICS '02: Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 66–76, 2002.

[17] Sun Microsystems. Throughput computing faq:http://www.sun.com/processors/throughput/faqs.html, 2005.

[18] D. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *22nd Annual Computer Measurement Group Conference*, Dec. 1996.

[19] D. Tullsen and J. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *34th International Symposium on Microarchitecture*, Dec. 2001.

[20] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *23rd Annual International Symposium on Computer Architecture*, May 1996.

[21] D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *22nd Annual International Symposium on Computer Architecture*, June 1995.