Online Strategies for High-Performance Power-Aware Thread Execution on Emerging Multiprocessors

Matthew Curtis-Maury, James Dzierwa, Christos D. Antonopoulos and Dimitrios S. Nikolopoulos
Department of Computer Science
College of William and Mary
P.O. Box 8795 – Williamsburg, VA 23187–8795
{mfcurt,jadzie,cda,dsn}@cs.wm.edu

Abstract

Granularity control is an effective means for trading power consumption with performance on dense shared memory multiprocessors, such as multi-SMT and multi-CMP systems. With granularity control, the number of threads used to execute an application, or part of an application, is changed, thereby also changing the amount of work done by each active thread. In this paper, we analyze the energy/performance trade-off of varying thread granularity in parallel benchmarks written for shared memory systems. We use physical experimentation on a real multi-SMT system and a power estimation model based on the die areas of processor components and component activity factors obtained from a hardware event monitor. We also present HPPATCH, a runtime algorithm for live tuning of thread granularity, which attempts to simultaneously reduce both execution time and processor power consumption.

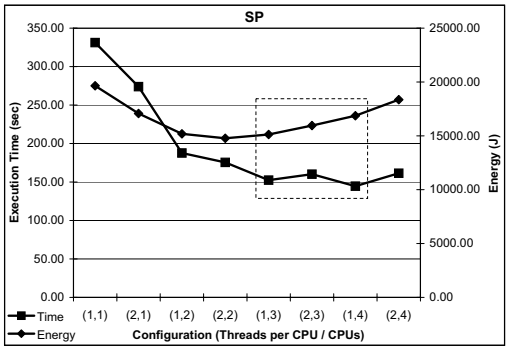

Figure 1. Execution time and CPU energy consumption of NAS SP on a 4-way SMP of Intel Xeon HT CPUs. The highlighted area indicates opportunities for improvement of energy and performance by coarsening thread granularity.

1 Introduction

Power consumption and heat dissipation have recently emerged as central themes in high-end computing. The operating costs of Teraflop-scale supercomputers run up to millions of dollars annually, due to power consumption, cooling requirements and frequent component failures [2]. The higher component counts and component density of emerging supercomputers pose a hard requirement for power-aware system design, and systems such as the BlueGene/L respond to this challenge by supporting aggressive power optimizations [1].

A unique aspect of power optimization in high-end computing systems is that, unlike mobile devices, the systems need to sustain high performance, regardless of power optimizations. Although power optimization techniques originating from the embedded computing domain, such as dynamic voltage and frequency scaling, are applicable in parallel programs [2, 7], concurrency itself is a natural optimization tool for both performance and power in the context of parallel computing. This paper explores the use of concurrency control, and more specifically the use of thread-level granularity control, as a means for improving the power and performance efficiency on dense shared-memory systems. Our work targets emerging multicore and multi-SMT processors [5, 8, 14, 18], which are currently dominating the server and high-end computing markets.

The idea of granularity control has been explored earlier in the context of schedulers for multiprogrammed shared-memory multiprocessors [19]. There are two compelling reasons for using granularity control in standalone multithreaded applications. The first is performance. Depending on their characteristics, applications may lack enough concurrency to exploit all available processors and threads in certain phases of their code. At the other end, they may exercise too much pressure on shared resources within each processor, such as cache memories and TLBs [6, 9], or resources shared between processors, such as memory bandwidth. Therefore, throttling concurrency by increasing thread granularity can potentially improve performance. The second reason is power. The system can selectively deactivate execution cores within processors, or entire processors, to save CPU power. Because CPUs

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

are generally the dominant power consumer in modern systems [7], they are an excellent target for power reduction.

Figure 1 shows an example of the potential for performance and CPU energy consumption improvement. Granularity control can be applied at all levels of nested multiprocessors, by deactivating processors, processor cores or individual execution contexts, and is orthogonal to power saving strategies such as clock/frequency scaling and synergetic to processor halting by the operating system.

We present a quantitative study of the opportunities for improving CPU power and performance efficiency on dense multiprocessors using thread granularity control. Our study uses a real system (a multiprocessor with Hyperthreaded Intel processors) running parallel codes from the NAS Benchmarks [4] and a realistic runtime CPU power estimation model based on realtime execution data from hardware event counters [3]. To our knowledge, this is the first study of simultaneous power and performance optimizations for nested multithreading multiprocessors, using a real system and set of applications. Our study reveals numerous opportunities for performance and power optimization via granularity control, as well as the power/performance trade-off of activating simultaneous multithreading on the Intel Xeon processors.

We also present HPPATCH, an online thread-level granularity control scheme, which adaptively optimizes code for power and performance by performing localized searches and live timing analysis of thread granularities which are likely to reduce CPU power without negatively affecting performance. HPPATCH achieves substantial improvements of power and performance efficiency, while also allowing the user to trade a specified performance penalty with power savings.

The rest of this paper is organized as follows: Section 2 discusses the merits of adaptive thread granularity control and presents HPPATCH. Section 3 presents our experimental setup and Section 4 presents the most important results from our experiments. Section 5 summarizes related work. Finally, Section 6 concludes the paper.

2 Power-Aware Thread Granularity Control

We propose an online thread-level granularity control scheme, whereby an estimate of the optimal number of threads and physical processors is determined at runtime and used for the execution of the application. The objective is to consume less CPU power by executing some phases of the application with fewer processors, without negatively affecting performance. Since some phases may actually execute more quickly on fewer processors, due to alleviation of thread interference and bus contention, the total execution time of the application may be reduced as well by using this technique.

Our adaptive thread-level granularity control is implemented through instrumentation of codes parallelized using OpenMP and requires no additional compiler support. We used OpenMP parallel regions as phase markers. The parallel regions have been marked with calls to our adaptation library which denote the beginning and end of each region. Since iterative codes have multiple outermost loop iterations, each phase will be executed potentially many times. The library records the execution times of each phase with varying numbers of threads and processors to find the combination with the lowest execution time for that phase. Once the best configuration for a particular phase has been found, it is applied for the execution of the remaining invocations.

Phase analysis is a widely used process for identification and optimization of dominant patterns in programs. Although we have used a manual phase instrumentation process for the purposes of this work, we note that our instrumentation can be generalized and implemented in an automated tool. We are currently integrating an automatic phase identification scheme in our software, which is loosely based on basic block vector analysis [17]. Our scheme uses the entry points of parallel loops and regions for phase identification and characterization and the Manhattan distance between vectors of disjoint entry points for detecting dominant program phases.

2.1 Heuristic Searches of Thread Granularities

An exhaustive search of all possible numbers of processors and threads per processor to identify the optimal thread granularity [21] incurs unnecessary overhead, particularly in large-scale or multi-level systems. Besides a potentially large number of configurations to be searched, exhaustive searching has to test even those configurations that are expected to perform poorly, despite the low probability of selecting them as the best. In applications with few outermost loop iterations, it is not possible to amortize the cost of such attempts and performance and CPU power consumption may suffer. Therefore, our heuristic opts for a localized, non-exhaustive search strategy. The following discussion of our heuristic assumes a multi-SMT or multi-CMP system.

HPPATCH (for *High-Performance Power-Aware Thread Control Heuristic*) approximates the optimal configuration while requiring drastically ewer iterations than exhaustive searches to come to a decision. This approach is based on the conservative expectation that more processors are likely to result in faster executions for most program phases.

HPPATCH begins by recording the execution times of each program phase with all processors and all threads per processor active. The search sequentially tries fewer processors with all threads per processor active until a number of processors is encountered which results in an increase in the execution time of the program phase under consideration. The local minimum of the execution time function, with respect to the number of processors, is used as the preferred number of processors for the specific phase.

The second stage of the search attempts to reduce the number of active threads per processor, given the number of processors selected during the first stage of the search. Although we have so far experimented with 2-way SMT processors, in which the search for the optimal number of threads per processor is trivial, in the general case we recursively apply the search strategy used to find the optimal number of processors per phase. This means that we sequentially decrement the number of active threads per processor until we begin to ob-

serve performance degradation. For architectures with three levels of parallelism, namely multiprocessors with CMPs of SMTs, HPPATCH includes an additional stage before selecting the number of threads per processor, where the number of cores on each processor to leave active is determined. This stage employs the same approach as is used to determine the number of physical processors and the number of execution contexts per SMT to use. Since HPPATCH performs only a localized search of program configurations, it is possible to end-up with sub-optimal configurations.

Note that processor deactivation (simply leaving a processor idle, though not explicitly transfered to lower power mode) always ends up in power savings, whereas thread deactivation within active processors may or may not result in power savings. Power consumption is linear to the number of active processors. Whether deactivating threads within a processor reduces power or not, depends on the processor architecture. On a multicore processor, thread deactivation will always result in power savings, since entire cores, each occupying significant area of the die, will be powered down. On SMT processors, which share a common pipeline between multiple threads, power savings are not necessarily linear to the number of deactivated threads, since the remaining active threads may still occupy the released resources to maximize their ILP.

On a system of n physical processors, each with m cores of l execution contexts, HPPATCH will try no more than $n+m+l-2$ configurations and may try as few as 3 for 2 levels of parallelism or 4 for 3 levels of parallelism. Furthermore, the search approach used in HPPATCH is likely to avoid trying the configurations that have the worst execution times, i.e. those with too few processors. As a result, this technique is likely to outperform the exhaustive search for applications with too few outermost loop iterations to amortize the overhead of the initial search phase, as well as for scalable applications with program phases which execute significantly slower when the number of active processors is reduced.

To accurately select the most power-efficient configuration in all cases, we would need to employ runtime power estimates, and combine them with live timing of program phases. The model we are currently using for power estimation requires four executions of each phase with different event counter rotations to collect the data needed for an estimate. Therefore, the model is hard to apply on-line in HPPATCH.We are currently investigating the use of power predictors derived from a few hardware event counts, which could be used for live estimation of energy/delay and, eventually, accurate selection of the most power-efficient thread granularity within each SMT processor.

2.2 Searches with Performance Tolerance

We extend the basic framework described in the preceding section, with a notion of quality of service. The latter provides additional flexibility, in order to take into account CPU power considerations in the decision of the number of threads and processors to employ. Rather than simply choosing the configuration with the lowest execution time, the adaptive algorithm is allowed to choose a configuration with a slightly higher execution time if it results in the use of fewer processors. We implemented this performance-tolerant approach in both the exhaustive and the HPPATCH search strategies.

In systems where performance is the primary concern, such an approach is not likely to be beneficial. However, systems where minimal energy consumption for a given program execution is desired may profit. We provide a simple interface to define the allowed degree of performance tolerance. It is expressed as the percentage of performance loss allowed in return for the deactivation of one processor.

3 Experimental Setting

To evaluate the effectiveness of HPPATCH, we use a power estimation model proposed by Isci *et al* [3]. This model provides a formula to calculate the power consumption of a processor based on hardware parameters and hardware performance counters. The model works by approximating the power consumption of each of the 22 major CPU components using an estimate of the maximum power and the access rates for each component. The formula for power consumption of each component is the following:

$$Power(C_i) = AccessRate(C_i) * ArchScaling(C_i) * MaxPower(C_i) + NonGatedClockPower(C_i)$$

The maximum power of each processor component is estimated to be the maximum power of the processor scaled down by the ratio of the die area occupied by the component [3]. We determined the die area ratio of each component using a floorplan layout of the Pentium 4 provided in [10]. The access rate is the number of "accesses" to the component per cycle, and is computed using hardware performance counters to record events associated with each component. The architectural scaling of each component is the number of accesses that can be simultaneously executed. The access rate multiplied by the architectural scaling equals the percentage of the total execution time during which each component is active and, therefore, how much of its maximum power it is consuming. Finally, non-gated clock power gives an additional constant that is consumed by each component when not idle. This term accounts for power consumption that grows non-linearly with the access rate. The total CPU power consumption is taken to be the sum of the power consumption of all components, plus a fixed idle power. The idle power is charged to CPUs even when they have been deactivated under the conservative assumption that they will not be transitioned to a lower power mode by the operating system. This assumption biases the results against our approach, since in reality processors will frequently be available for power mode transitions and therefore additional power savings. In the reported experimental results, we sum the power consumption of all 4 processors.

Hardware performance counters were collected using PACMAN (for *PerformAnce Counters MANager*), a library we have written to collect event counts on Hyperthreaded processors. PACMAN provides low-overhead access to the Pen-

Bench	BT	CG	FT	LU	LU-HP	MG	SP	UA
Iters	200	15	6	250	250	4	400	200

Table 1. Benchmarks used from the NPB suite and the number of outermost loop iterations.

tium 4 counters using the Perfctr interface [16]. Although up to 18 events can theoretically be recorded simultaneously using Pentium 4 event counters, there are complex limitations as to which can be recorded at the same time. As a result, the events required for power estimation must be recorded in four different rotations. To estimate the CPU power of a given configuration, the program needs to be executed four times with one rotation of events being recorded each time. The CPU power is then computed based on these values.

We experimented using eight applications from the OpenMP version of the NAS Parallel Benchmarks suite (version 3.1) [4]. The benchmarks were compiled using the Intel FORTRAN compiler (version 9.0) with the problem size set to class A. This problem size is large enough to yield realistic results while being small enough to ensure the working sets of all applications fit entirely in main memory. The NAS benchmarks are iterative with a consistent workload between iterations, and are representative of the vast majority of parallel applications. Furthermore, they contain applications with both few and many outermost loop iterations (see Table 1).

We performed our evaluation of thread granularity control on a Dell PowerEdge server, composed of 4 Hyperthreaded Intel Xeon MP 1.4GHz processors. Each processor can simultaneously execute up to 2 threads. The machine has 1GB of main memory and a 512KB level-3 cache, 256KB level-2 cache, 8KB level-1 data cache and 12KB level-1 instruction trace cache, per processor. Bus-based multiprocessors are a target platform for thread granularity control, since bus bandwidth (3.2 GB/s in our system) may limit the effectively exploitable number of processors. Although our target platform has a relatively low bus bandwidth, our work is still relevant for emerging architectures, considering that the number of cores in CMPs (which currently stands at 2-8) is projected to exceed 64 by 2010 [15], thus making bandwidth between processor and on-chip or off-chip memory a major limitation in next generation systems as well. Experiments were run using Linux kernel version 2.4.25.

4 Results

To better understand the CPU energy and execution time properties of the 8 benchmarks from the NAS suite, we first executed them using 8 different static configurations on our 4-processor, 2-way SMT system. Specifically, we ran the benchmarks using 1 to 4 processors and either 1 or 2 threads per processor. In static executions, the same configuration has been used throughout the execution life of the application. We then executed the applications using the two adaptive strategies, namely the one based on exhaustive trial of all configurations and the HPPATCH heuristic, both under different degrees of performance tolerance.

Figure 2 depicts the execution time and CPU energy consumption of each application with the 8 static configurations (left side), as well as with the exhaustive and HPPATCH adaptive configurations with no performance tolerance (right side of each diagram). The configurations that yield the lowest execution time and energy consumption are tagged in the diagrams with a striped bar and a star mark respectively.

Figure 3 focuses mainly on the behavior of the adaptive strategies. More specifically, we execute each application under the exhaustive and HPPATCH configurations. Each diagram reports 2 metrics, normalized with respect to a static execution with 8 threads: i) The energy consumed for the execution of the application, and ii) The $Energy * Delay^2$ metric (ED^2). The latter is a popular metric for evaluating power / performance tradeoffs in the context of high performance computing [12, 13, 22]. ED^2 takes into account both the energy requirements and execution time of applications, however it treats the execution time (*Delay*) as the first-class target for optimization. The right segment of each diagram depicts the same ratios for the static configurations that yielded the lowest execution time, ED^2 and energy consumption.

In sections 4.1 and 4.2 we analyze the experimental results of the static and adaptive execution strategies respectively.

4.1 Static Configurations

Figure 2 outlines many opportunities for potential CPU energy savings, without significant performance degradation. All applications – with the exception of BT – are characterized by a global CPU energy consumption minimum at 2 or 3 physical processors, across static configurations. The increase in CPU power requirements when more than 2 or 3 processors are used outweighs potential execution time improvement, thus resulting in a negative CPU energy balance. Even in terms of performance, the improvement attained by activating the 4th processor is usually minimal. It is, thus, possible to identify favorable configurations in terms of both CPU energy consumption and execution time. In the case of FT, for example, the execution with 3 threads on 3 processors proves both energy- and time-optimal.

The effect of the activation of the second execution context of each processor on performance is not consistent. Certain applications benefit from the exploitation of the second execution context, either in all (UA) or at least in some configurations (CG, LU, LU-HP, MG, SP). For other applications the exploitation of the second execution context consistently leads to performance degradation (BT, FT). The power estimation model indicates that using the second execution context usually results in a minimal 3%-5% increase in the power consumed by each CPU. There are, however, cases where power consumption is slightly increased during single-threaded execution due to fewer conflicts, which result in higher occupancy of resources. As a result, the CPU energy balance is mainly dependent on the respective performance balance.

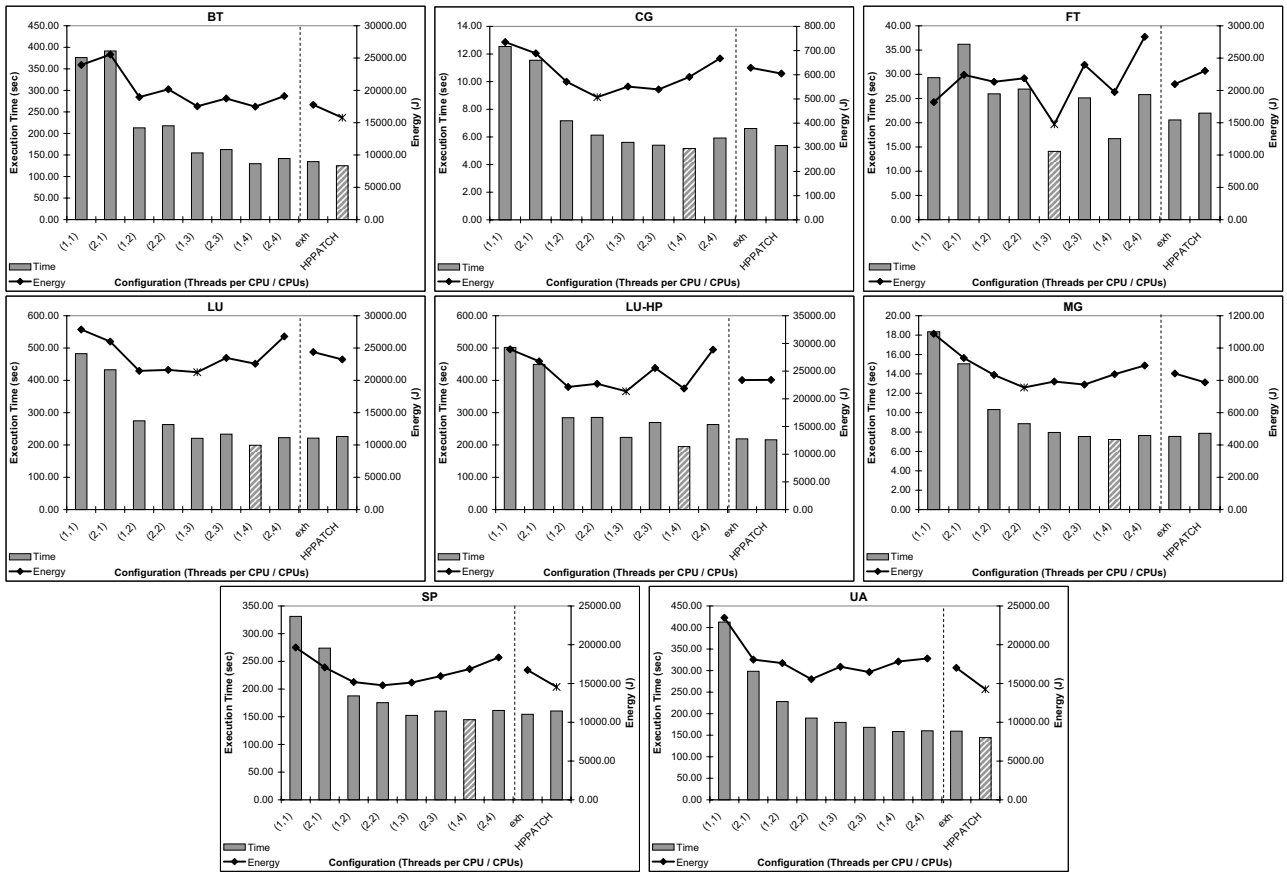

Figure 2. Execution time and CPU energy requirements of each application under the 8 different static and the 2 adaptive configurations.

Our experimental results (not depicted due to space limitations) verify the intuitive expectation for a strong correlation between the optimal performance and ED^2 static configurations. In 6 out of 8 applications the best performing static execution is the one that yields the lowest ED^2 as well. As a consequence, performance-centric adaptive execution strategies, such as HPPATCH, are likely to identify sweetspots in the performance/energy tradeoff as well.

4.2 Adaptive Execution

It is clear from the previous discussion that deriving the optimal (performance- or power-wise) configuration for each application is not a straightforward task, even for an experienced programmer. In fact, our profiling results indicate that different loops of the same application may execute optimally under different configurations. In UA, for example, each of the 8 static configurations proves optimal for at least one loop. The shortcomings of statically configured executions motivate the use of dynamic, adaptive execution strategies.

Figure 2 depicts the execution time and CPU energy consumption attained by an exhaustive search (exh) strategy and the HPPATCH adaptive heuristic, with no performance tolerance. The execution times for exh and HPPATCH are on average 14% and 11% higher, respectively, than the fastest static configuration for each application. If the execution time of the adaptive policies is compared to that attained by a static execution with 8 threads – which would be the natural configuration choice on the specific system – exh and HPPATCH policies are, on average, 5% and 7% faster respectively. In the case of UA, HPPATCH outperforms even the best static configuration by 9%, since the adaptive policy allows different loops to execute with different numbers of threads, according to their characteristics.

FT and MG are not susceptible to adaptive optimizations. They have a small number of outer iterations (6 and 4 respectively). As a result, the adaptive policies do not have enough opportunities to evaluate different configurations and, even if they reach a decision, the remaining iterations are not enough to amortize the cost of the search phase. However, even a moderate external iteration count – 15 in the case of CG – proves sufficient for the adaptive strategies to be effective. In the rest of the discussion we exclude the results from FT and MG when calculating averages of metrics across applications.

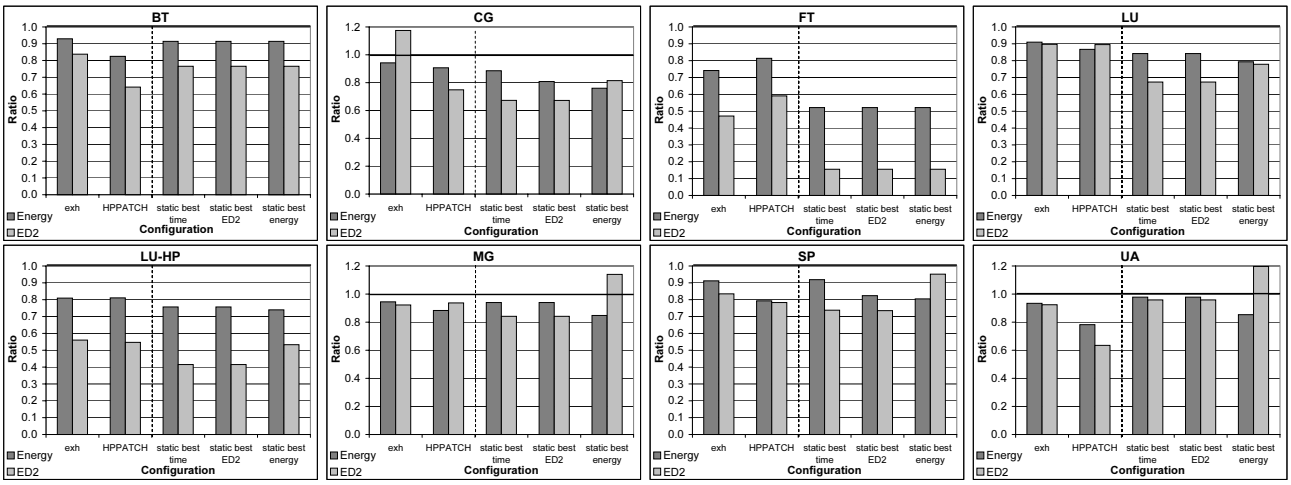

Figure 3. Normalized values of CPU energy consumption and ED^2 over the static execution with 8 threads. We report the results for the exhaustive search and HPPATCH adaptive strategies as well as for the fastest, the lowest ED^2 and the lowest energy consuming static executions.

Excluding FT and MG, exh and HPPATCH are a mere 10% and 5% slower than the execution time-optimal static configuration respectively.

The use of adaptive strategies is typically associated with a certain degree of overhead for two reasons: First, during the search phase, loops are executed with suboptimal configurations. The exhaustive search strategy is penalized more by the search phase, since all configurations have to be tested, down to the sequential execution, before a decision can be made. HPPATCH, on the other hand, usually converges to a decision much faster. The second source of overhead relates to the potential execution of consecutive loops with different configurations. Programmers of scientific codes tend to optimize their applications for cache performance, under the assumption that the application is going to be executed with a fixed number and placement of threads. Adaptive strategies often change the number of threads in consecutive loops, thus distorting cache locality [6]. In order to evaluate the extent of the performance degradation the distortion of cache locality is responsible for, we executed LU-HP under all 8 static configurations. We then identified the optimal configuration for each loop and recorded the loop execution time under that configuration. Next, we executed the application once more, using the off-line identified optimal configuration for each loop. We found that the loops suffered a slowdown of 19% on average, with respect to their optimal execution times recorded in the static configurations. This overhead can be attributed solely to the dynamic variation of the degree of parallelism and thread placement across loop boundaries.

The results of increasing the degree of performance tolerance (not depicted due to space limitations) show, as expected, an increase in the execution times of applications. Slower configurations are favored, in order to deactivate processors and reduce CPU power consumption. This strategy seems to be beneficial for the average CPU energy consumption of applications, when they are executed under the exhaustive search adaptive scheme. It results in an average of 5.9% energy savings, compared to the adaptive execution with no performance tolerance. For the HPPATCH adaptive heuristic, however, the energy consumption increases on average by 1.2% for executions with performance tolerance higher than 0%. The exhaustive search scheme can better exploit opportunities offered by higher performance tolerances, since it has a broader search space than HPPATCH. In the case of the ED^2 metric, however, the experimental results are clearly against configurations tolerating performance loss. Performance tolerance higher than 0% results in an average of 2.6% and 23.1% higher (worse) values of the ED^2 metric for the exhaustive search and the HPPATCH adaptive strategies respectively. Even the minimal CPU energy reductions – in the case of exhaustive search – cannot outweigh the execution time increase induced by tolerating performance loss.

Figure 3 provides more insight into the CPU energy requirements of applications under the two different adaptive strategies when no performance loss is tolerated. The results of a comparative evaluation of the exhaustive search against the HPPATCH adaptive strategy are clearly in favor of HPPATCH. HPPATCH results, on average, in 5% faster execution, 7.6% less CPU energy consumption and 7.3% better ED^2. Although, theoretically, HPPATCH may converge to suboptimal configurations, in practice it usually converges to the correct decisions with just a fraction of the search cost and cache distortion.

Table 2 summarizes the performance results for HPPATCH. Compared to a static execution with 8 threads, HPPATCH without performance loss tolerance results, on average, in 16.9% less CPU energy consumption and 29.2% better ED^2. The adaptive strategy manages to identify configurations for each individual loop of the application that allow the deactivation of physical processors without penalizing perfor-

	Time	Energy	ED^2
8 Threads	8.0	16.9	29.2
Time Opt	-4.5	5.2	-5.4
Energy Opt	8.2	-3.1	11.9
ED^2 Opt	-2.8	1.9	-5.4

Table 2. Improvement (%, higher is better) in time, energy and ED^2 using HPPATCH, compared to using 8 threads and the execution time, energy and ED^2 optimal static configurations.

BT	CG	FT	LU	LU-HP	MG	SP	UA
3.52	2.81	2.67	4.00	2.29	2.62	2.66	3.67

Table 3. Average weighted number of physical processors used by each application when executed under the HPPATCH adaptive strategy without performance loss tolerance.

mance. Table 3 reports the average weighted number of physical processors used by each application. The value has been calculated by weighting the number of processors selected in each loop by the percentage of contribution of the specific loop to the total execution time of the application. Even in the case of LU, which uses 4 processors throughout its execution life, HPPATCH identifies the opportunity of deactivating the second execution context of each processor in the most time consuming loop of the application, thus resulting in both execution time improvement and CPU power / energy savings.

Even compared to the fastest and the most ED^2 efficient static configurations, HPPATCH experiences ED^2 performance that is a mere 5.4% worse. On the other hand, HPPATCH results in 5.2% and 1.9% energy consumption savings compared with the aforementioned static strategies, respectively. HPPATCH also consumes only 3.1% more energy than the energy optimal static configuration, while executing 8.2% faster and with 11.9% better ED^2 performance. These results show that HPPATCH is, on average, competitive with even the best static configuration, regardless of the metric being used, without requiring *a priori* knowledge of which configuration to use for each application.

5 Related Work

High-performance power-aware execution of MPI programs on distributed memory platforms has recently been investigated [2, 7]. These works have identified opportunities to save energy without impacting execution time in MPI programs by exploiting idle periods during communication and collective operations. The authors have proposed several automated methods for saving energy during idle periods by reducing the frequency and the voltage supply of processors. These methods operate on clusters with processors that allow DVFS (dynamic frequency and voltage scaling).

Our work differs from research on power and performance-efficient MPI execution in that it improves energy and performance efficiency on shared-memory multiprocessors and SMT/CMP architectures. Our scheme, similarly to the just-in-time DVFS technique presented in [7], performs timing analysis across outer iterations of parallel codes, however it applies energy optimizations at a different granularity, namely that of phases enclosed by parallel loop boundaries, rather than entire outer iterations. Furthermore, our work uses a different approach for energy savings, that is, the deactivation of processors and execution contexts.

Our study of power and performance implications of tuning thread granularity in parallel codes shares similar objectives with a study of the power-performance trade-offs of chip multiprocessors presented in [11]. The authors investigate the implications of both DVFS and thread granularity on performance using a single, simulated CMP. Our work differs in several respects. It is conducted on a real multi-SMT system and the power-performance optimizations proposed are applicable at runtime via live timing analysis, rather than derived statically from a power-performance model [11]. Furthermore, our approach looks for power-performance optimization opportunities by granularity control in program phases, rather than by fixing the granularity of the entire program.

Runtime analysis of loops for granularity control towards energy savings has been considered in [6], where the authors proposed compiler support for generating multiversion loops to control thread granularity at runtime. The work in [6] performs thread control at the granularity of loop iterations instead of entire loops, which is the case in our scheme. No results were reported in [6] and the work, as with [11], targeted individual chip multiprocessors rather than systems with multiple physical processors.

The use of hardware event counters for the estimation of energy consumption on microprocessors has been explored in [3, 20]. The CPU power estimation model used in this study was adopted from the Pentium 4 model presented in [3]. The work presented in [20] is the first to use event counters at runtime to derive an estimation of power which can be used to dynamically scale voltage and frequency from within the OS scheduler during system operation.

6 Conclusions

Using physical experimentation with a real system and a realistic power model based on hardware event counters, we have evaluated the CPU energy / performance tradeoffs that arise by tuning thread granularity during the execution of parallel applications on multi-SMT systems. Our study shows that the execution time-wise optimal static configuration often leaves idle processors and execution contexts. Because of the diminishing returns of using more processors for performance, it is natural to consider deactivating processors in order to improve energy efficiency.

We have presented an adaptive approach, HPPATCH,

which takes advantage of the limitations in the exploitable parallelism of parallel applications to leave execution contexts or entire physical processors idle during certain program phases. HPPATCH works by searching for a number of threads and processors which optimizes execution time for each program phase. In so doing, it provides the potential to reduce not only execution time, but also CPU power consumption by reducing the number of processors actively consuming power. In our experiments, HPPATCH was able to reduce execution time by 8%, CPU energy consumption by 17% and $Energy * Delay^2$ by 29% on average over a range of parallel applications compared to using all available execution contexts, which would be the natural choice of a performance-conscious programmer. Furthermore, HPPATCH's localized search method proves to be significantly more effective than exhaustive runtime searching of program configurations and a viable approach to autonomous optimization of parallel programs on dense parallel architectures.

Acknowledgments

This research is supported by the National Science Foundation (Grants CCR-0346867 and ACI-0312980), the U.S. Department of Energy (Grant DE-FG02-05ER2568) and an equipment grant from the College of William and Mary.

References

[1] N. Adiga and et al. An Overview of the BlueGene/L Supercomputer. In *Proc. of the IEEE/ACM Supercomputing'2002: High Performance Networking and Computing Conference*, Baltimore, MD, Nov. 2002.

[2] R. Ge, X. Feng, and K. Cameron. Improvement of Power-Performance Efficiency for High-End Computing. In *Proc. of the 19th International Parallel and Distributed Processing Symposium*, Denver, CO, Apr. 2005.

[3] C. Isci and M. Martonosi. Runtime Power Monitoring in High-End Processors: Methodology and Empirical Data. In *Proc. of the 26th ACM/IEEE Annual International Symposium on Microarchitecture*, pages 93–104, San Diego, CA, Nov. 2003.

[4] H. Jin, M. Frumkin, and J. Yan. The OpenMP Implementation of the NAS Parallel Benchmarks and its Performance. Technical Report NAS-99-011, NASA Ames Research Center, Oct. 1999.

[5] R. Kalla, B. Sinharoy, and J. Tendler. IBM POWER5 Chip: A Dual-Core Multithreaded Processor. *IEEE Micro*, 24(2):40–47, Mar. 2004.

[6] M. Kandemir, W. Zhang, and M. Karakoy. Runtime Code Parallelization on Chip Multiprocessors. In *Proc. of the 2003 Design, Automation, and Test in Europe Conference*, pages 510–515, Munich, Germany, Mar. 2003.

[7] N. Kappiah, V. Freeh, and D. Lowenthal. Just In Time Dynamic Voltage Scaling: Exploiting Inter-Node Slack to Save Energy in MPI Programs. In *Proc. of IEEE/ACM Supercomputing'2005: High Performance Computing, Networking Storage, and Analysis Conference*, Seattle, WA, Nov. 2005.

[8] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-Way Multithreaded SPARC Processor. *IEEE Micro*, 25(2):21–29, Mar. 2005.

[9] R. Kumar, N. Jouppi, and D. Tullsen. Conjoined-Core Chip Multiprocessing. In *Proc. of the 37th International Symposium on Microarchitecture (MICRO-37)*, pages 195–206, Portland, OR, Dec. 2004.

[10] K.-J. Lee and K. Skadron. Using Performance Counters for Runtime Terperature Sensing in High-Performance Processors. In *Proc. of the 19th International Parallel and Distributed Processing Symposium*, Denver, CO, Apr. 2005.

[11] J. Li and J. Martínez. Power-performance implications of thread-level parallelism on chip multiprocessors. In *Proc. of the 2005 International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Austin, TX, Mar. 2005.

[12] Y. Li, D. Brooks, Z. Hu, and K. Skadron. Performance, Energy, and Thermal Considerations for SMT and CMP Architectures. In *Proc. of the 11th International Symposium on High-Performance Computer Architecture*, Washington, DC, 2005.

[13] M. Martonosi, D. Brooks, and P. Bose. Modeling and Analyzing CPU Power and Performance: Metrics, Methods, and Abstractions. In *SIGMETRICS 2001 / Performance 2001 - Tutorials*, 2001.

[14] C. McNairy and R. Bhatia. Montecito: A Dual-Core, Dual-Thread Itanium Processor. *IEEE Micro*, 25(2):10–20, Mar. 2005.

[15] National Research Council. *Getting up to Speed. The Future of Supercomputing*. National Academies Press, Washington DC, 2004.

[16] M. Pettersson. A Linux/x86 Performance Counter Driver. http://user.it.uu.se/∼mikpe/linux/perfctr/.

[17] T. Sherwood, E. Perelman, and B. Calder. Basic Block Distribution Analysis to find Periodic Behavior and Simulation Points in Applications. In *Proc. of the 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT'2001)*, pages 3–14, Barcelona, Spain, Sept. 2001.

[18] T. Takayanagi, J. Shin, B. Petrick, J. Su, and A. Leon. A Dual-Core 64b UltraSPARC Microprocessor for Dense Server Applications. In *Proc. of the 41st Conference on Design Automation (DAC'04)*, pages 673–677, San Diego, CA, June 2004.

[19] A. Tucker. *Efficient Scheduling on Multiprogrammed Shared-Memory Multiprocessors*. PhD thesis, Department of Computer Science, Stanford University, Nov. 1993.

[20] A. Weissel and F. Bellosa. Process Cruise Control: Event-Driven Clock Scaling for Dynamic Power Management. In *Proc. of the 2002 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 238–246, Grenoble, France, Oct. 2002.

[21] Y. Zhang and M. Voss. Runtime Empirical Selection of Loop Schedulers on Hyperthreaded SMPs. In *Proceedings of the 2005 IEEE International Parallel and Distributed Processing Symposium*, Denver, CO, Apr. 2005.

[22] A. Zmily and C. Kozyrakis. Energy-Efficient and High-Performance Instruction Fetch Using a Block-Aware ISA. In *Proc. of the 2005 International Symposium on Low Power Electronics and Design*, New York, NY, 2005.