

Online Power-Performance Adaptation of Multithreaded Programs using Hardware Event-Based Prediction

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

With high-end systems featuring multicore/multithreaded processors and high component density, power-aware high-performance multithreading libraries become a critical element of the system software stack. Online power and performance adaptation of multithreaded code from within user-level runtime libraries is a relatively new and unexplored area of research. We present a user-level library framework for nearly optimal online adaptation of multithreaded codes for low-power, high-performance execution. Our framework operates by regulating concurrency and changing the processors/threads configuration as the program executes. It is innovative in that it uses fast, runtime performance prediction derived from hardware event-driven profiling, to select thread granularities that achieve nearly optimal energy-efficiency points. The use of predictors substantially reduces the runtime cost of granularity control and program adaptation. Our framework achieves performance and ED^2 (energy-delay-squared) levels which are: i) comparable to or better than those of oracle-derived offline predictors; ii) significantly better than those of online predictors using exhaustive or localized linear search. The complete prediction and adaptation framework is implemented on a real multi-SMT system with Intel Hyperthreaded processors and embeds adaptation capabilities in OpenMP programs.

Categories and Subject Descriptors

C.1.4 [Computer Systems Organization]: Processor Architectures—*Parallel Architectures*; D.4.1 [Operating Systems]: Process Management—*Threads*; D.4.8 [Operating Systems]: Performance—*Modeling and prediction*

General Terms

Management, Measurement, Performance

Keywords

online adaptation, power-aware computing, performance prediction, hardware performance counters

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICSO6, June 28–30, Cairns, Queensland, Australia.

Copyright (c) 2006 ACM 1-59593-282-8/06/0006 ...\$5.00.

1. INTRODUCTION

Multicore and multithreaded processors have a prominent role in high-end computing. Thread-level parallelism has long been recognized as the means to sustain performance scaling and overcome the limitations of conventional techniques for exploiting instruction-level parallelism [13]. Although multicore and multithreaded architectures have proven their potential and have been adopted in several commercial products [8, 9, 10], the programming technologies used for these architectures date from almost two decades ago. Software is still parallelized using POSIX threads or similar low-level substrates, regardless of whether the target architecture is an SMP, a ccNUMA, a CMP, an SMT or a layered architecture such as an SMP of SMTs or a CMP of SMTs.

Although well established, existing multithreading libraries lack essential capabilities for tapping into the performance and power advantages of emerging CMPs, SMTs and layered architectures built from these components. Effective execution of programs on multiprocessors with multiple layers of multithreading requires rethinking of the design of threading libraries, for several compelling reasons:

- Threads interact with different layers of parallel execution hardware – such as execution contexts in an SMT processor, cores in a multicore processor or processors in an SMP system – in complex and non-trivial ways. Interaction between threads and hardware and inter-thread interference may also differ substantially from one layer to another. Threads interfere in both positive and negative manners, depending on a large number of cross-cutting factors involving the program such as granularity, locality, control structure, instruction-level parallelism and the architecture, such as cache/TLB size and organization, available bandwidth, branch predictor design, instruction cache design, etc. Unfortunately, multithreading libraries overlook the interaction between threads and hardware as well as thread interference at different parallel execution layers. As a result they lack the sophistication to effectively map parallel computation to layered parallel architectures, or even the expressiveness to allow the programmer to do so.
- Multithreading libraries currently lack power-aware execution capabilities. If negative interference between threads at certain execution layers of the system could be detected by a library, the library would be able to simultaneously improve performance and reduce power consumption by throttling concurrency and deactivating threads, execution cores or entire processors. With steep increases in processor power consumption, rising energy cost and more system maintenance and reliability problems arising due to power and thermal considerations, energy conservation becomes equally as important as high per-

formance. The potential of energy conservation through multithreading libraries is, however, not well explored.

These observations suggest that multithreading libraries can no longer consider concurrency as an one-dimensional entity. They need to regulate concurrency at multiple parallel execution layers, using potentially different strategies at each layer. Moreover, multithreading libraries should provide a versatile software infrastructure for high-performance and energy-aware execution of parallel programs on layered multiprocessors. This paper addresses these issues by proposing a novel, transparent, self-managed scheme for online performance and power adaptation in the context of multithreading libraries. Specifically, this paper makes the following contributions:

- We present a runtime performance prediction model based on *dynamically collected and dynamically analyzed* profiles of parallel execution phases. The profiles are collected from hardware event counters. The prediction model uses a few snapshots – more precisely as many as the number of parallel execution layers in the architecture – of normalized event rates, for each phase of multithreaded execution in the program. The event rates are forwarded to an online predictor, which derives a transfer function to predict the IPC of the given phase under different system configurations (number of processors, number of threads per processor). The predictor makes an instantaneous decision on the seemingly optimal (from a performance, energy, or energy-performance perspective) number of threads to activate at each layer of the architecture.

In addition to providing the necessary input for adaptation, our prediction model minimizes runtime overhead, without necessarily sacrificing accuracy in finding optimal or nearly optimal operating points in the program. In this respect, our prediction model substantially improves previous work on adaptive thread-level concurrency control, which relied heavily on exhaustive or semi-exhaustive searches and timing of program configurations. Furthermore, our prediction model is the first targeted explicitly at IPC prediction across system configurations and is significantly more accurate than other proposed models based on hardware event-based predictors when applied in this context.

- We present a comprehensive framework for autonomic power-performance adaptation of multithreaded programs, including a multithreaded hardware event profiling component, an on-the-fly performance characterization and prediction runtime system using feedback from the profiling component, a rapid predictor, and the necessary thread control mechanisms to implement adaptation at runtime.
- We deploy and evaluate our framework on OpenMP programs running on a real layered multiprocessor (a system with multiple Intel Hyperthreaded processors) and show that it achieves performance, energy consumption and ED^2 which are comparable to, and often better than, those achieved by the best static execution scheme. The best static scheme can only be known through an oracle or exhaustive experimentation. Our scheme also outperforms by significant margins other runtime program adaptation schemes that use direct search approaches, in terms of performance, energy and energy-delay metrics.

The rest of this paper is organized as follows. Section 2 discusses related work in the area of performance and power adaptation on parallel architectures. In section 3, we present the complete power-performance adaptation framework. Section 4 presents our experimental analysis and results. Finally, section 5 concludes the paper.

2. RELATED WORK

2.1 Runtime Loop-Level Adaptation for High Performance

Researchers have used live timing analysis of loops in iterative parallel codes to improve performance by controlling the number of worker threads and the loop scheduler used by the runtime system on a loop by loop basis. Recent relevant research efforts using Intel’s Hyperthreaded processors as a test bed have been presented by Zhang and Voss [19] and by Jung et al. [7]. These schemes are not power-aware, since they consistently activate all processors in the system, even in cases when activating less processors could be done without harming performance. Furthermore, live timing analysis of loop executions requires substantial runtime overhead and may force short-lived programs or programs with fine-grain phases to execute for significant parts of their lifetime with sub-optimal configurations. Our work overcomes this problem using instantaneous and direct performance and power predictions.

The classification tree approach for prediction presented in [19] partially mitigates the runtime overhead but suffers from two drawbacks which make it unsuitable for our purposes. The first is that classification trees have difficulties accurately predicting continuous performance metrics, in particular when multiple, numerically close predictions need to be compared to derive a decision on the optimal operating point for a given phase of the program. Second, classification based on absolute values of events is inferior as a performance or power predictor, since the absolute number of occurrences of an event does not necessarily reflect the contribution of the event to either execution time or power consumption.

Jung et al. [7] use fixed thresholds to decide between using one or two hyperthreads per processor and between executing loops sequentially, or using all processors. Their adaptation scheme is based on the threshold value of a single metric, namely L2 cache misses per instruction. Our work shows that performance adaptation with a single classifier is inadequate for locating power and performance-efficient program execution points across the spectrum of possible processor/thread configurations. For example, the parallel loop at line 43 of file `12norm.f` from the LU-HP benchmark of the NAS Benchmark suite, has a miss rate per instruction per thread of 0.016 or higher. The threshold used in [7] for switching off Hyperthreading was 0.01 on the same processor that we use for our experiments. However, the specific loop consistently obtains performance improvement of at least 30% per processor, when Hyperthreading is activated.

2.2 Event-Driven Accounting of Energy

Power modeling using hardware event counters (also known as hardware performance monitoring counters), is a relatively recently explored area. The characteristic of hardware counters which makes them appealing or estimating runtime power consumption is that they reflect the activity levels of different components of the microprocessor over time. Combining these activity levels with power consumption estimates for each individual component leads to simple, yet accurate power estimation methodologies. Such methodologies have been presented by Weissel and Bellosa [18], and Isci and Martonosi [5]. The model proposed in [5] is prohibitively costly for runtime power estimation and optimization. It requires four complete program executions with different, rotating counter configurations, in order to collect the necessary input information. Therefore, although our work uses this model for offline power and energy consumption measurements of program executions, we use a simpler and faster relativistic power model to estimate the impact of concurrency control on power consumption at runtime.

2.3 Compiler-Driven and Architectural Mechanisms for Power Control on CMPs

The optimizing compiler and computer architecture communities have recently studied the problem of power-efficient program execution on chip multiprocessors from various perspectives, using predominantly simulated chip multi-processors.

Liu et al. [12] have exploited idle time during barriers to stretch computation via DVFS and conserve energy without harming performance. Our contribution, which is based on granularity control, is complementary and orthogonal to DVFS-based schemes.

Li and Martínez [11], presented some runtime search algorithms for DVFS and processor control, to find power-efficient program execution points on chip multiprocessors. The authors used a combination of hill climbing, binary search and linear search to find locally optimal numbers of processors and voltage/frequency levels, given specific performance targets. Reduced search overhead via prediction is one important difference in our work. Furthermore, their scheme searches program configurations with a fixed performance target, while our scheme searches for an *initially unknown* optimal (from a performance or power-performance perspective) operating point. In other words, our scheme does not require prior knowledge of application performance characteristics.

2.4 Symbiotic Job Scheduling on SMTs/CMPs

Several researchers have investigated the problem of symbiotic job scheduling on SMTs, CMPs and layered designs (e.g. CMPs of SMTs), using approaches similar to our work.

Moseley et al. [15] presented two methods, based on classification trees and linear regression respectively, to predict the IPC of sequential programs when executed in pairs on a Hyperthreaded processor. Their methods use as input a sample of the rates of some selected hardware events during an execution interval (25 million cycles), to predict the IPC of the same program during the next execution interval. The predicted IPC values are used in turn to derive decisions for co-scheduling high IPC with low IPC threads on each SMT, thereby achieving better thread symbiosis [17]. Our work has a different objective, since it targets power-efficient execution of parallel programs, rather than efficient execution of workloads of multiple sequential programs. In terms of the prediction model used, the scheme presented in [15] uses uniform random sampling of the program to delineate phases, whereas our scheme identifies phases at the boundaries of recurring parallel regions. A third important deviation is that we employ a different performance prediction model, using event rates as multiplicative factors for IPC, instead of linear components of IPC [17].

DeVuyst, Kumar and Tullsen have recently proposed various symbiotic job schedulers for CMPs of SMTs [2]. Their schedulers, similarly to our framework, use runtime performance projections. More specifically, they use direct measurements of performance and energy, such as IPC, and $PowerperCycle/IPC$ or $PowerperCycle/IPC^2$. These measurements are used as feedback to their decision making process which can optimize for performance, energy, or both, by shuffling threads across cores and moving threads away from underutilized cores. Our runtime system uses similar metrics as the *output* of our predictor, however we sample only two occurrences of each program phase and the input to our predictor consists of multiple event rates. Using input from multiple hardware event counters is necessary in our framework, since standalone values of IPC are not necessarily strong indicators of scalability across configurations with a variable number of processors or threads. Although the scheme in [2] shares the objective of optimizing performance and energy on layered multiprocessors with our work, it targets multiprogram workloads of sequential pro-

grams and cannot be applied for the adaptation of parallel codes.

3. POWER-EFFICIENT PROGRAM EXECUTION USING INSTANTANEOUS PERFORMANCE PREDICTION

Earlier efforts on adaptive parallel program execution techniques on multiprocessors – including chip multiprocessors, SMTs, and multi-SMTs – have heavily relied upon runtime searching of program configurations, using different numbers of threads/processors, and/or different voltage/frequency levels. An enabling factor for using direct search methods is that many applications, particularly in the scientific and engineering computing domains, have dominant periodic phases. The applications execute multiple iterations of discrete parallel execution phases. Some of these iterations can thus be leveraged for a direct search of the best program configuration for any given phase.

Unfortunately, there are several disadvantages in using direct search approaches for runtime adaptation. On systems with a large number of processors, cores per processor, threads per core or voltage/frequency levels, direct search can span many iterations through the execution life of the application, during which the application is executed with mostly sub-optimal configurations, incurring significant performance and power penalties. Furthermore, many important codes have only few main iterations, or have fine-grain, short-lived parallel execution phases, during which direct search is often infeasible. Although direct search may be improved with heuristics such as hill climbing [1, 11], with microprocessor technology moving to multicore designs with more cores and higher degrees of multithreading in each core, direct search methods may prove inadequate and inefficient.

Our contribution in this context is a comprehensive, self-managed runtime adaptation scheme, based on predictions of the optimally exploitable number of execution contexts at each architectural layer of the system, namely processors, cores per processor and threads per core.

3.1 Basic Performance Prediction Strategy

In the following discussion, we assume for simplicity a two-level multiprocessor architecture, with multiple SMT processors. Our scheme performs a two-level prediction for selecting how many threads per processor and how many processors to use in each parallel execution phase. The prediction scheme uses input from two test executions of the phase on all available processors (*all_procs*), one with multithreading activated ($SMT_{active} = 1$), and a second with multithreading deactivated ($SMT_{active} = 0$) on each processor¹. More specifically, the input consists of the two observed IPCs ($IPC_{obs(all_procs, SMT_{active})}$) and a set of additional hardware performance metrics ($m_1(all_procs, SMT_{active}), \dots, m_n(all_procs, SMT_{active})$) observed at runtime, during each of the two test executions. These performance metrics are attained from performance monitoring counters, a resource available on all modern microprocessors. We note here that we assume the test executions of each phase are on the critical path of the program and are not off-line experiments.

For our purposes, a phase is defined as a parallel region. Our predictor is not dependent on the number of processors or the number of threads or execution cores available on each processor. Instead, it applies as many prediction steps as the number of architectural layers available in the machine. In a three-level multiprocessor (multiple processors, each a CMP, with multiple threads in each core),

¹On an SMP with 4 SMT processors, supporting 2 threads each, the test configurations would be ($4procs, 2thr/proc$) and ($4procs, 1thr/proc$).

our predictor would obtain input from 3 executions, one targeting each of the discrete architectural layers of the system.

For each possible configuration ($nproc, nthr/proc$), we predict performance in terms of the estimated cumulative IPC (IPC_{est}) across all processors/threads executing the application:

$$IPC_{est}(nproc, nthr/proc) = IPC_{obs}(all_procs, SMTActive) \cdot H_{(nproc, nthr/proc)}(m_1(all_procs, SMTActive), \dots, m_n(all_procs, SMTActive)) + e_{(nproc, nthr/proc)} \quad (1)$$

Equation 1 estimates the IPC for the target ($nproc, nthr/proc$) configuration using a transfer function $H_{(nproc, nthr/proc)}()$ to scale the observed IPC at the corresponding test execution². The estimated IPC is then corrected by a constant residual, $e_{(nproc, nthr/proc)}$. The transfer function is a linear combination of the hardware metrics $m_1(all_procs, SMTActive), \dots, m_n(all_procs, SMTActive)$ observed during the corresponding test execution. It should be noted that the model applies a different $H()$ and e for each target configuration.

Our performance predictor is based on the assumption that the relative difference in the IPC between parallel executions under different processor / threads per processor configurations is equivalent to the relative performance difference attained by those configurations. This assumption is valid in the context of data-parallel loops, since the total computational load – and thus the total number of instructions – is independent of the way the loop is executed. Our technique excludes the overhead of synchronization instructions from IPC measurements and estimations. Spinning at synchronization points results in many instructions that are unrelated to the main computation, which obfuscates predictions by distorting instruction count and instruction stream characteristics. However, synchronization instructions are taken into account during the evaluation of this technique.

Our framework supports OpenMP programs and leverages the OpenMP mechanisms for controlling the number of threads used on a region by region basis. It includes our own customized library for monitoring performance and for binding threads on specific processors and hardware contexts within processors. Our model requires $N + 1$ invocations (including a warmup iteration) of each parallel phase to predict IPC for all possible configurations on an N – level layered multiprocessor. Our predictor directly selects the most “efficient” configuration, based on the IPC predictions. The criterion for efficiency is adjustable and can be tuned to give priority to performance, power or a combination of power and performance.

3.2 Predictor Details

3.2.1 Selecting Hardware Events for Prediction

The hardware events we use for prediction quantify basic performance-limiting events, both within and across processors. They are all normalized with respect to the duration (in clock cycles) of each parallel region. More specifically, we monitor the *rate of bus accesses*, as an indicator of memory access intensity and contention for accessing the shared bus, the *rate of L2 cache misses*, as an indicator of locality and potential interference for the shared cache by threads executing on the same SMT processor, the *percentage of cycles in which the processor’s trace cache is in deliver mode*, as an indirect indicator of the instruction cache hit ratio specifically for the Intel P4, the *rate of branch instructions*, as an indicator of the granularity – or indirectly the length – of basic blocks

²Depending on whether the target execution has simultaneous multithreading activated or not.

in the target parallel region, the *rate of mispredicted branches*, as an indication of the frequency of pipeline flushes and discarding of speculative computation, and *retired instructions per cycle (IPC)*, which is used as the basis for the calculation of the estimated IPCs of the other configurations.

Our choice of hardware events can also be justified in quantitative terms. The Hyperthreaded Pentium processor support counting of up to 18 individual hardware events, shared between the two hyperthreads. Not all combinations of events are permissible on the counters. An optimal exhaustive event selection strategy would try all permissible combinations of events on the 9 counters corresponding to each thread and select the combination which is the most accurate in predicting IPC across configurations. Such an exhaustive search is impractical. Furthermore, only a few events affect performance significantly, and thus should be used for performance predictions. Therefore, we ignored events with negligible occurrence rates. Besides the 6 events mentioned above, other events that seem to contribute significantly to performance are memory hierarchy events, such as L1 misses and TLB misses. Unfortunately, L1 and TLB misses cannot be measured in addition to the aforementioned set of events, due to conflicts for counters / configuration registers. Furthermore, L1 and TLB misses are, intuitively, less significant contributors to performance and scalability than L2 cache misses and bus accesses on our experimental platform, in which the limited memory bandwidth is a primary concern.

Finally, we performed a simple sensitivity analysis to investigate whether each of the 6 selected events contributes to prediction accuracy. Our results (not shown due to space considerations) indicated that each of the 6 events contributes to an improvement of prediction accuracy, and the prediction accuracy may improve by as much as 44%, as the event set size is grown incrementally from 1 to 6 events. Certain events, such as bus accesses per cycle are more critical than others in terms of affecting prediction accuracy.

3.2.2 Estimation of the Transfer Functions

As we mentioned in section 3.1, the transfer functions used to derive an IPC prediction for a specific ($nproc, nthr/proc$) configuration are linear combinations of performance metrics monitored during a test execution ($all_procs, SMTActive$) of the target code region. Each transfer function $H_{(nproc, nthr/proc)}$ can be written as:

$$H_{(nproc, nthr/proc)}(m_1(all_procs, SMTActive), \dots, m_n(all_procs, SMTActive)) = \sum_{i=1}^n (a_i(nproc, nthr/proc) \cdot m_i(all_procs, SMTActive) + b_i(nproc, nthr/proc)) + c_{(nproc, nthr/proc)} \quad (2)$$

Combining equations 1 and 2, the IPC estimation for a specific ($nproc, nthr/proc$) configuration is calculated – starting from IPC and hardware metrics observations from the test execution of the ($all_procs, SMTActive$) configuration – as:

$$IPC_{est}(nproc, nthr/proc) = \sum_{i=1}^n (a_i(nproc, nthr/proc) \cdot m_i(all_procs, SMTActive) - IPC_{obs}(all_procs, SMTActive)) + d_{(nproc, nthr/proc)} - IPC_{obs}(all_procs, SMTActive) + e_{(nproc, nthr/proc)} \quad (3)$$

where $d_{(nproc, nthr/proc)} = c_{(nproc, nthr/proc)} + \sum_{i=1}^n b_i(nproc, nthr/proc)$. As a result, the estimation of the transfer function for a specific

$(nproc, nthr/proc)$ configuration is equivalent to the estimation of the coefficients $a_{i(nproc, nthr/proc)}$ and $d_{(nproc, nthr/proc)}$, as well as of the residual $e_{(nproc, nthr/proc)}$.

The coefficients and the residual are calculated by applying linear regression off-line, using the least squares method, on a training population of parallel regions. After selecting a subset of our benchmarks for training, we executed each benchmark in the training set statically with a fixed $(nproc, nthr/proc)$ combination throughout each execution, and repeated the same executions for all valid combinations of $nproc$ and $nthr/proc$ available on the system. The resulting cumulative IPC has been recorded for each parallel region in the training set. For the configurations that correspond to test executions, we have also recorded the values of the 5 additional metrics ($m_{i(all_procs, SMTactive)}$) for the events listed in section 3.2.1. The set of benchmarks used in the training process is entirely disjoint from the set of benchmarks used in our evaluation.

During the training for each target configuration, the observed IPC for the specific configuration plays the role of the *dependent variable*. The *independent variables* are the products $m_{i(all_procs, SMTactive)} \cdot IPC_{(all_procs, SMTactive)}$, as well as the $IPC_{(all_procs, SMTactive)}$ alone. The application of linear regression on the training population results in estimations for the coefficients $a_{i(nproc, nthr/proc)}$, $d_{(nproc, nthr/proc)}$ and the residual $e_{(nproc, nthr/proc)}$. These off-line calculated values are used in equation 3 to derive – at run-time – IPC estimations for all possible configurations of a target parallel region, given the performance data collected during the two³ test executions of the specific region.

To further refine our prediction model, we divided the IPC observations in two buckets, using the IPC value of 1.0 as a threshold, and applied our regression process independently for the samples in each bucket. The rationale behind this decision was to derive different scaling coefficients of IPCs for scalable and non-scalable parallel regions. The threshold value of 1.0 was selected from empirical observation of IPC scaling in our test cases. More educated threshold selection strategies will be investigated in future work.

3.2.3 Adaptation Criteria

The performance predictor is flexible and can be used to provide input to adaptive strategies. Our adaptive infrastructure integrates a multitude of strategies, targeting different performance/energy optimization criteria.

The simplest strategy is a pure performance-oriented one. The adaptive policy selects the configuration that results in the highest cumulative predicted IPC across all possible $(nproc, nthr/proc)$ configurations. Although the strategy is not directly energy aware, it has the potential of improving energy consumption by reducing application execution time, since parallel regions may execute faster with a less than maximum degree of concurrency. Further, through the performance oriented deactivation of processors, less power is consumed as well. A slight variation of this policy allows some performance penalty, to a user-specified extent, if this can potentially predict a configuration that uses fewer processors and thus consumes less power.

More sophisticated adaptation strategies take into account energy consumption and try to optimize either pure-energy or energy/performance combining criteria. As we explained in section 2.2, existing models for power consumption estimation through performance counters [5] are not practical for use in an online power adaptation strategy, since they require four whole executions of the target code region in order to collect all necessary metrics. As a work-around, we use static, scaling coefficients to approximate the

Config	(1,1)	(1,2)	(2,1)	(2,2)	(3,1)	(3,2)	(4,1)	(4,2)
Power	0.32	0.34	0.55	0.59	0.79	0.81	1.0	1.01

Table 1: Power scaling coefficients used to approximate the expected cumulative power consumption of all processors of our experimental platform, under different configurations. Power is normalized with respect to the consumption in the (4 processors / 1 thread per processor) configuration.

expected power consumption in different configurations, with respect to the power consumption of a base case. The coefficients are calculated by applying the power estimation model to the same population of training regions used for the estimation of transfer functions for IPC scaling (section 3.2.2). Table 1 summarizes the static power coefficients derived for our experimental platform.

A pure, energy-driven adaptation strategy estimates the relative energy consumption under different configurations by multiplying the estimated CPI (1/IPC) for each configuration by the respective power scaling coefficient. The strategy then executes each region with the configuration that results in the lowest energy consumption. Besides energy, metrics such as $energy \times delay$ (ED) or $energy \times delay^2$ (ED²) are also popular in the context of high performance computing systems [14], since they take into account both the energy savings and the potential performance penalty. We implemented an ED- and an ED²-driven strategy, in which we estimate the relative ED and ED² among different configurations by multiplying the power scaling coefficient for each configuration with the estimated CPI² or CPI³ respectively.

3.2.4 Implementation Issues

Our power/performance-centric adaptation infrastructure is implemented – without limitations in its applicability – in the context of a back-end runtime library for applications parallelized using OpenMP. It uses the boundaries of parallel regions as phase markers⁴. At the same time, it intercepts entries and exits to/from the body of each parallelized loop. The interception is performed right after the work distribution code and right before the barrier-type synchronization that are usually introduced by the OpenMP runtime at the beginning and at the end of loops respectively, in order to avoid distorting the measured performance data by code unrelated to the main computation performed in each loop.

As explained earlier, a set of performance metrics is monitored during the two test configurations executed at the first two invocations of each parallel region. The performance-related information is read from the performance monitoring counters of the processor. We use PACMAN (*PerformAnce Counters MANager*), our custom performance monitoring library, to configure performance monitoring hardware and read the data. PACMAN, in turn, uses the Perfctr kernel-level driver [16] to gain access to the counters and their configuration registers. We use per-thread performance monitoring, instead of system-wide performance monitoring. System-wide monitoring supports a safe sampling rate of only 10 Hz, since the values returned by Perfctr are updated approximately every 100 msec, at scheduling points or upon entry to / exit from the kernel. Per-thread monitoring directly reads the values of the counters on the processor on which the thread executes. As a result, it allows frequent sampling of the counters and is suitable for monitoring very fine program regions.

Pentium 4 processors share performance monitoring hardware

⁴The boundaries of loops would be more accurate phase markers, however the OpenMP specification prohibits varying the number of processors inside a parallel region.

³In the case of a 2-level architecture.

Bench.	BT	CG	FT	IS	LU	LU-HP	MG	SP	UA	MM5
Iter.	200	15	6	10	250	250	4	400	200	180
Regions	10	14	8	5	10	20	13	15	59	189
Re-exec. regions	5	5	5	1	3	11	6	9	49	70

Table 2: Benchmarks used throughout our experimental evaluation. UA and MM5 were used for training.

Benchmark	IPC Error (%)
BT	9.34
CG	18.33
FT	25.07
IS	127.33
LU	27.91
LU-HP	15.32
MG	11.14
SP	8.46
All	19.5

Table 3: The accuracy of the IPC predictor, measured as the weighted percentage of difference between predicted and actual IPC across all parallel regions in all valid static configurations.

between the threads simultaneously using the two execution contexts of the processor. As a result, conflicts may occur if both threads attempt to use the same counter- or configuration-register. In order to eliminate conflicts, Perfctr allows threads that use performance monitoring to execute only on the first execution context of each processor. We have removed that limitation from Perfctr and delegated to PACMAN the assignment of non-conflicting performance monitoring configurations for threads executing on the same processor, as well as the binding of threads to specific processors and execution contexts within each processor. It should be pointed out that the binding scheme also targets the minimization of cache distortion and the optimal exploitation of already cached data whenever configurations are changed, either across different parallel regions, or during the initial test executions of each parallel region.

Whenever a processor remains idle, the operating system has the option to put it at its deepest power-saving mode. For Intel Pentium 4 processors, for example, this is achieved by executing the privileged `hlt` instruction. The overhead of transferring a Pentium 4 processor to the halted state and back is less than 1000 cycles [19], whereas the power consumption of the processor is reduced by approximately 7W (from 9W when idling to 2W in halted state). Although our adaptive strategies do not directly put processors in halted state, they leave processors idle whenever possible, allowing the operating system to demote them to halted state. In fact, our experimental results indicate that the processors are halted during 89% of their idling time in the course of our adaptive program executions.

4. EXPERIMENTAL RESULTS

4.1 Experimental Setting

To evaluate our power-performance adaptation framework, we used nine OpenMP benchmarks from the NAS Parallel Benchmarks suite [6] (version 3.1) and an OpenMP implementation of a mesoscale weather prediction model, MM5 [4]. The NAS benchmarks used are BT, SP, LU, LU-HP, CG, FT, MG, IS and UA. For the results presented in this section, we used the parallel regions in

UA and MM5 as our training set and the rest of the NAS benchmarks as our test set. UA and MM5 combined contain a fairly large number of parallel regions (248 in total), with a wide variance in absolute performance (IPC), scalability, granularity, locality and other performance and power-critical characteristics. In general, our training set selection strategy was to select minimal sets of applications with a good coverage of regions that stand at the opposite ends of the performance and scalability spectra. Here we use one out of many possible training sets from our benchmark collection that meet our selection criterion.

We experimented on a Dell PowerEdge server, composed of 4 Hyperthreaded Intel Xeon MP 1.4GHz processors, with 1GB of main memory. Each processor has a 512KB level-3 cache, a 256KB level-2 cache, a 8KB level-1 data cache and a 12KB level-1 instruction trace cache. The system runs Linux kernel version 2.4.25. We compiled the benchmarks with the Intel FORTRAN compiler (version 9.0). We ran the NAS benchmarks with the problem size set to class A, which is large enough to yield realistic results, while being small enough to ensure that the working sets of all applications fit entirely in main memory. The benchmarks used for testing our runtime system and our prediction model contain applications with as few as 4 and as many as 400 outermost loop iterations (see Table 2), as well as a varying number of recurring parallel regions, ranging from 1 in IS to 11 in LU-HP. Recurring parallel regions are the ones offering opportunities for power-performance adaptation.

4.2 Results

4.2.1 IPC Predictor Accuracy

Table 3 shows a direct evaluation of the accuracy of the IPC prediction model described in Section 3. The IPC predictor is evaluated as follows. For each parallel region, and for each of the 8 possible configurations to execute the region, we record the observed IPC and the IPC predicted by our model. We calculate the normalized error (as a percentage) and we weigh the error with the weight of the region in the parallel code. The weight of the region is the percentage of parallel execution time attributed to the region. Finally, we sum the weighted errors to derive a single IPC error rate for each benchmark. Table 3 summarizes these results. We observe that the weighted IPC error is under 20% in five out of our eight test cases and is unacceptably high only in IS. Note however that although the IPC error is very high in IS, our model correctly predicts the optimal configuration to execute the single parallel execution phase of the benchmark. Overall, the IPC prediction accuracy is good, considering the simplicity of our prediction model, and its performance compares very favorably to the performance of other statistical approaches for IPC prediction reported in the literature [3].

Another point of comparison for our IPC prediction model are previously proposed models in which IPC is predicted as a linear combination of individual hardware event rates and partial products of event rates [15], rather than with a transfer function. We replicated the model presented in [15], in which both individual events rates and all intermediate products of event rates are used as independent variables to predict IPC, the dependent variable in the regression model. Using the same training set as used for our approach, the IPC prediction error of that model ranged between 84% and 700% across all configurations of our benchmarks. This clearly shows that our model is more effective at predicting IPC values across system configurations than that proposed in [15].

Though a useful indicator, the absolute IPC prediction error itself is not the most important metric of effectiveness for our prediction model. Our predictor will only be effective if it selects operat-

	% Parallel execution time with optimal prediction	Weighted performance improvement/loss in mispredicted regions
BT	64.10	-0.83
CG	0.01	+11.27
FT	51.86	+18.49
LU	99.39	-0.05
LU-HP	94.45	-0.08
MG	88.03	+4.63
SP	27.88	-1.28
IS	100	0.00
AVG	65.72	+4.02

Table 4: Configuration prediction accuracy of our model. The second column shows the percentage of parallel execution time during which our model predicts correctly the best static optimal configuration. The third column shows the weighted performance improvement or loss incurred in the regions where our predictor mispredicts the optimal configuration.

ing points which are identical to or near (in terms of performance and power consumption) the optimal operating points that would be selected by an oracle. In other words, *optimal configuration prediction accuracy* is a more important metric than absolute IPC prediction accuracy for our predictor.

Table 4 provides insight into the configuration prediction accuracy of our model. The table shows two indicators of configuration prediction accuracy: The fraction of time during which the adaptive version of each benchmark executes with the statically optimal configuration (as observed offline for each region) predicted correctly by our predictor; and the weighted performance loss (or gain) in the remainder of each benchmark, while our predictor mispredicts the optimal configuration. We calculate this weighted performance difference as $\sum_{i=1}^{N_B} w_i \times D_i$, where N_B is the number of mispredicted regions in benchmark B , w_i is the weight of each mispredicted region expressed as the percentage of the total parallel execution time of B that the region accounts for, and D_i is the performance improvement (positive) or loss (negative) in mispredicted region i , compared to the performance of the best static configuration for the same region.

As can be seen from the second column in Table 4, our predictor successfully predicts and executes regions with the optimal program configuration (in terms of both number of processors and the threads per processor) during 65.7% of the execution time of our benchmarks, on average. In 3 benchmarks, LU, LU-HP and IS, prediction accuracy in terms of deriving the optimal program configuration is excellent. Prediction accuracy is also very good in MG. In the 3 applications where optimal predictions are used during 94% or more of the execution time (LU, LU-HP, IS), the penalty of mispredictions is marginal, ranging from 0 performance loss (IS) to a maximum performance loss of 0.08% (LU-HP). In MG, there is actually a performance gain from misprediction of one region, which is explained in detail in the following paragraphs.

The four programs in which the predictor is suboptimal in terms of deriving the best configuration for each region (BT, CG, FT and SP) merit further discussion. Detailed examination of the results indicates that mispredictions incur a very small performance penalty in BT and SP (0.8% and 1.3% respectively). In CG and FT, mispredictions incur a seemingly counter-intuitive, and non-negligible performance improvement. In BT and SP, our predictor always predicts the second best static configuration, and the difference in performance between the two top configurations is marginal.

The counterintuitive result in CG and FT is explained as follows. CG has 3 consecutive mispredicted parallel regions. Let

us name these regions R_1, R_2, R_3 . R_2 accounts for 96% of the execution time. In those 3 regions our predictor predicts configuration (4,2) i.e. 4 processors, 2 hyperthreads per processor as the optimal, whereas the statically optimal configuration is (4,1). However, the statically optimal configuration of the parallel region immediately preceding the 3 mispredicted regions (we name it R_0) is (4,2). Our predictor retains (4,2) as the chosen configuration across the 4 consecutive regions R_0, \dots, R_3 , whereas an optimal static execution would switch the configuration from (4,2) in R_0 to (4,1) in R_1, \dots, R_3 . This switch would incur a non-negligible performance penalty due to cache interference and changes in the working sets of the 4 threads that stay alive after the switch from (4,2) to (4,1) in R_1 . By not changing the configuration, our predictor actually preserves cache locality and executes the dominant mispredicted region R_2 about 12% faster than the statically optimal configuration. Since R_2 covers more than 96% of the program, the overall performance of the adaptive execution with the predictor is better than that of the best static execution. The exact same phenomenon is observed in FT for 3 consecutive regions that account for more than 80% of the parallel execution time, and in MG for 3 consecutive regions which account for more than 70% of the parallel execution time. This positive result is not attributed to the design of our predictor, but to a side-effect of keeping configurations intact across phases, thus avoiding cross-phase negative interference.

To summarize, we observe that our predictor matches optimal oracle-derived predictions almost perfectly in a variety of benchmarks, and in the cases where it mispredicts, the performance penalty is either negligible, or non-existent, since the predictor often dictates that the system should maintain a stable configuration across critical program phases and preserve cache performance.

4.2.2 Performance and Energy Impact

Figure 1 depicts the execution times attained by different static configurations and dynamic adaptation strategies for the 8 benchmarks from the NAS suite. It is clear from the diagrams that, in many cases, providing more processors / threads to the application does not necessarily improve execution time. CG, for example, performs optimally with 3 threads, distributed across 3 different physical processors. Activating more processors or threads results in performance degradation. At the same time, the activation of the 4th processor obviously increases power and energy consumption as well. Dynamic adaptation policies look for such optimal points in the configuration space of applications in order to optimize either performance or an energy/performance combining metric.

Figure 2 depicts the performance of dynamic, adaptive strategies, with respect to *energy* (E), *energy* \times *delay* (ED) and *energy* \times *delay*² (ED^2), normalized over the static execution with 8 threads. The latter would normally be the natural configuration choice for a system with 4 processors and 2 Hyperthreads per processor. The rightmost column in each bar group corresponds to the optimal static configuration (a single configuration used for the duration of program execution) with respect to each metric. It should be noted that even in cases when all 4 processors are used by an adaptation strategy (see LU in Table 5), energy savings and performance improvement can often be obtained by deactivating the second Hyperthread on each processor.

The simplest adaptive strategy is the one that optimizes execution time for each region, based on exhaustive search of all possible configurations (*exh* in the diagrams). Exhaustive search outperforms the execution time of the static configuration using eight threads in all but one case (CG), by an average of 9%. It is also better in terms of ED^2 for all applications (32% on average). At the same time, it consumes – on average – 21% less energy than the 8

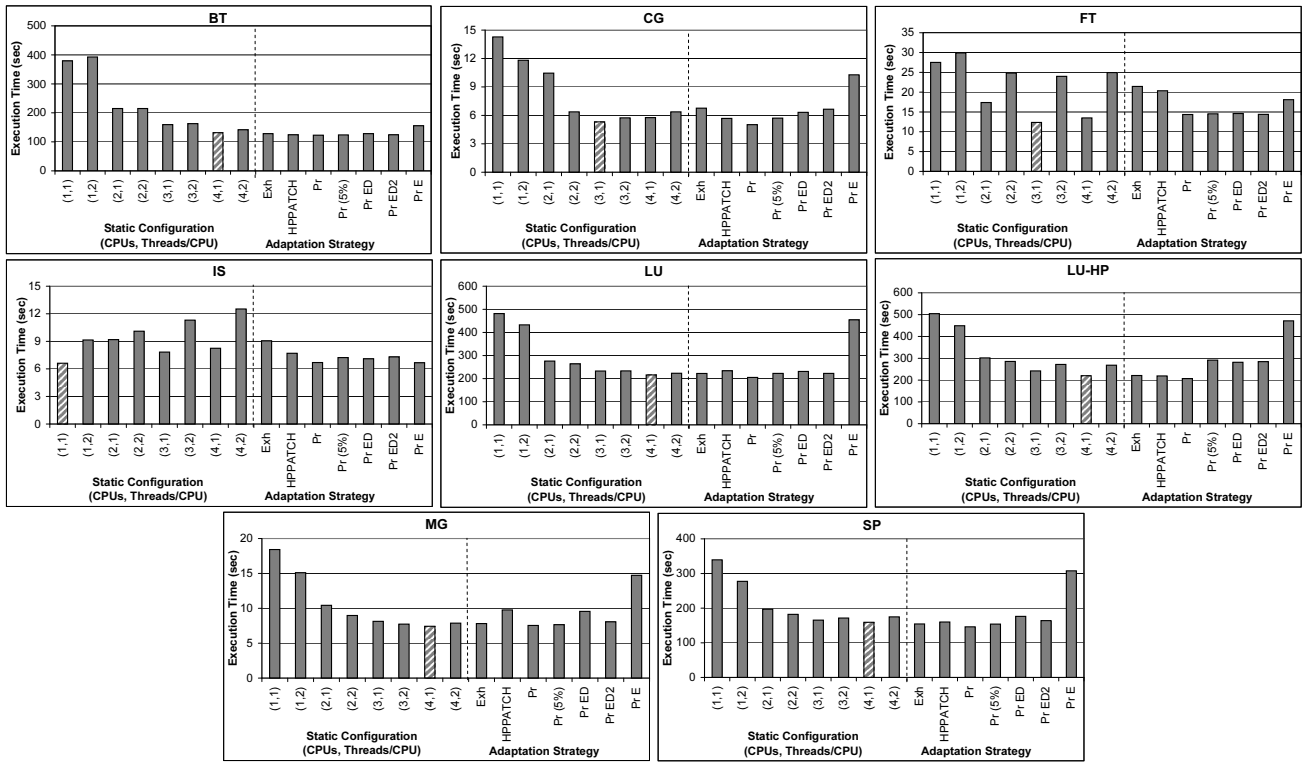


Figure 1: Execution time of the benchmarks under the 8 different static configurations (left side) and the 7 dynamic adaptation strategies (right side). The best performing static configuration for each application has been marked with striped bars.

threads configuration. However, exhaustive search suffers from excessive overhead during the search phase. The effect of the search phase overhead is more profound in applications with few outermost iterations. As a result, exhaustive search proves on average 18% slower than the execution time optimal static configuration.

In earlier work we presented HPPATCH [1], a hill-climbing search-based adaptive heuristic. HPPATCH performs a localized search of the configuration space. It starts from the (4,2) static configuration and looks for a configuration that results in a local minimum in terms of execution time. HPPATCH has the potential to decrease the total number of search iterations, and thus the associated overhead. At the same time, however, it may get trapped in a sub-optimal local minimum and fail to identify the best configuration. When compared to statically using eight threads, HPPATCH is 10% faster and uses 18% less energy on average. It is however 15% slower than the optimal, execution time-wise, static configuration. Even with the reduced number of search iterations, the initialization overhead is still non-negligible in applications with few outermost loop iterations, such as FT, IS and MG. Excluding these applications, HPPATCH is able to match – on average – the execution time of the optimal static configuration.

The performance adaptation strategy based on IPC prediction (*Pr* in diagrams) reduces the length of the search phase to just two iterations. When compared to the static 8 threads configuration, IPC prediction-based adaptivity is, on average, 22% faster. This corresponds to 14% and 13% improvements over exhaustive search and HPPATCH respectively. Even compared to the execution time optimal static configuration, IPC prediction-based adaptivity is faster by 2%, since a statically optimal configuration of an entire program may still be sub-optimal for certain phases of the program. In fact, the only static configurations that outperform predicted IPC-based

adaptation are for FT, IS and MG. This dynamic adaptation strategy performs well with respect to energy/performance-centric metrics, too. It consumes 26% less energy, compared to the static execution with 8 threads. The adaptive approach locates opportunities to execute with fewer processors and thereby consume less power, while – at the same time – reducing execution time as well. As a result, it also improves ED^2 by 49%, with respect to the 8 threads static execution and is within 11% of the ED^2 -optimal static configuration. Even in applications with few iterations (FT, IS and MG), the IPC prediction-based adaptive policy minimizes the overhead of the search phase and results in execution times within 6% of the optimal static configurations for these applications.

The next type of adaptation approach that we considered tolerates some degree of performance loss if this would allow using a configuration that consumes less power. We experimented with acceptable performance loss of 5%, 10% and 25%. In practice, activating a configuration which tolerates even a 5% lower IPC (*Pr* (5%) in the diagrams) often proved to result in a significant execution time increase. As a result, the potential energy savings from the use of fewer processors are outweighed by the execution time penalty. In fact, for all metrics, IPC prediction-based adaptivity without performance loss tolerance outperforms – on average – that with performance tolerance.

The adaptive strategy that directly targets the optimization of energy consumption (*Pr E* in diagrams) proves unsuitable in the context of a high performance computing environment. Despite the fact that it results in a 34% average reduction of energy consumption compared with the static, 8 threads execution, it still consumes 23% more energy than the energy-optimal static configuration. At the same time, it was on average 42% slower than a static execution with 8 threads. Table 5 reveals that the energy optimization strategy

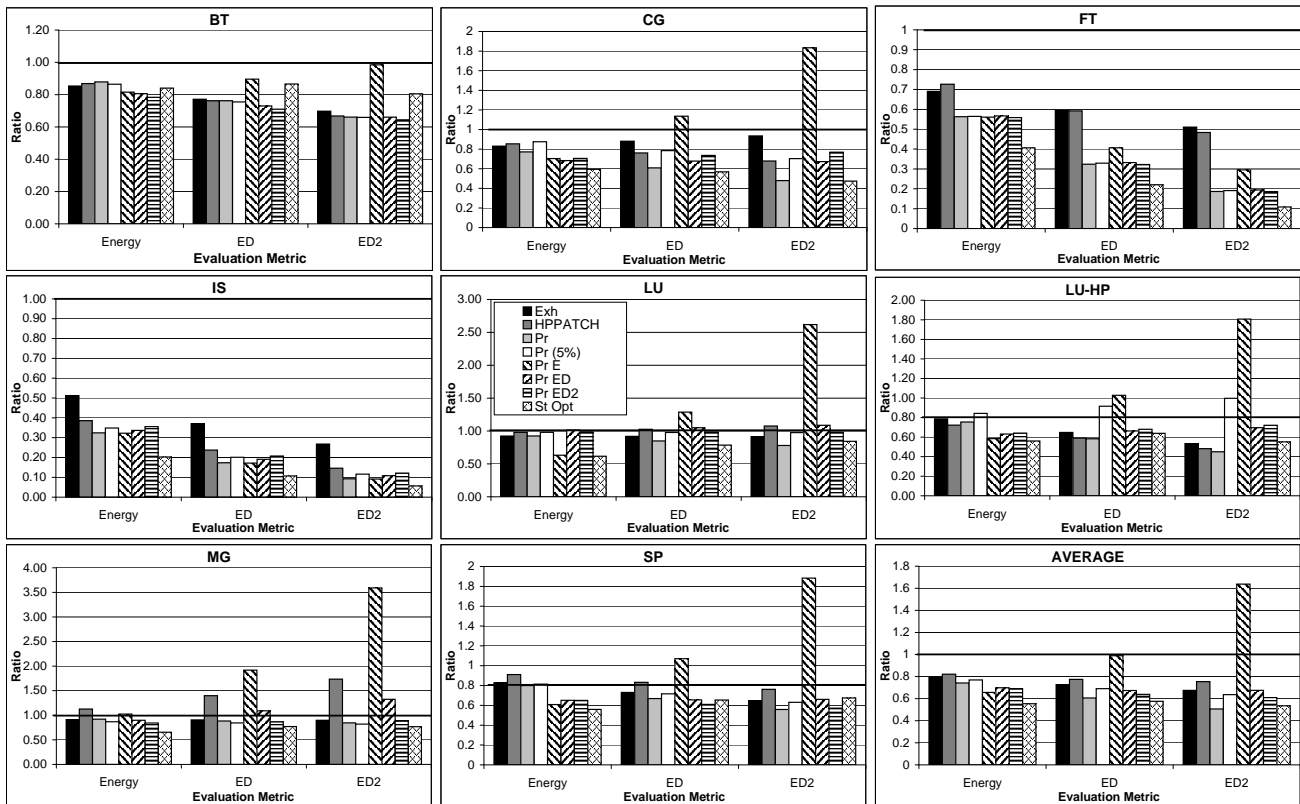


Figure 2: Performance of the dynamic adaptation strategies in terms of energy (first group of bars), energy \times delay (second group of bars) and energy \times delay² (third group of bars). Each group of bars has been normalized with respect to the performance of the (4 processors/2 threads per processor) static configuration for the respective metric. The rightmost bar in each group corresponds to the optimal static configuration for the respective metric.

Bench	BT	CG	FT	IS	LU	LU-HP	MG	SP
Exh	3.84	3.01	3.30	2.33	3.93	3.81	3.69	3.63
HPPATCH	3.99	3.82	3.76	1.83	4.00	3.51	3.57	3.99
Pr	3.94	3.98	3.90	1.61	4.00	3.86	3.16	3.65
Pr (5%)	3.94	3.99	3.81	1.62	4.00	3.01	3.43	3.55
Pr E	2.43	1.47	2.94	1.60	1.02	1.02	1.95	1.01
Pr ED	3.19	2.46	2.92	1.62	4.00	2.07	2.77	2.09
Pr ED ²	3.14	2.46	2.83	1.61	4.00	2.09	3.03	2.29

Table 5: Average number of processors used during the adaptive executions of the benchmarks, for each of the seven tested adaptation strategies.

favors executions with few processors, often downgrading parallel applications to sequential execution. This results in an unacceptable performance penalty, as the reader can observe in Figure 1.

The last two adaptive strategies focus on the minimization of the ED and ED² metrics (*Pr ED* and *Pr ED²* in the diagrams respectively). The average ED and ED² improvements over the static execution with 8 threads are in the range of 33% and 39% respectively. However, the two strategies are 27% and 34% worse than the ED and ED² optimal static configurations. At the same time, the ED and ED² attained by the pure IPC-based adaptation strategy are better than the ones attained by strategies specifically targeting the optimization of ED and ED² by 9% and 16% respectively. ED- and ED²-centric strategies tend to deactivate processors in order to reduce power consumption. This trend is clear in table 5. However,

they frequently select too few processors, thus increasing execution time on average by 18% and 14%. This happens due to the fact that any error in CPI prediction is amplified when ED or ED² are calculated because the CPI prediction is raised to the second and third powers respectively, as discussed in section 3.2.3. For example, if the CPI prediction error is a mere 10%, then the resulting prediction error for ED will be 21%, and 33% in the case of ED². It should also be taken into account that energy estimations are based on static energy scaling coefficients. We expect the performance of ED and ED²-centric adaptation strategies to significantly benefit from more accurate, performance counters-driven, run-time power estimators, which are in the focus of our future work.

Overall, the performance-centric adaptive strategy driven by IPC predictions outperforms the other adaptive approaches in all metrics (execution time, E, ED, ED²). Moreover it is significantly more efficient compared with static executions which activate all execution resources of the architecture and proves comparable to or better than optimal, oracle-chosen static configurations. The adaptive strategy is based on a fast, accurate online IPC predictor which reduces the configuration search phase to just 2 iterations and at the same time yields optimal or almost optimal configuration predictions for each parallel region of the program.

5. CONCLUSIONS AND FUTURE WORK

Chip multithreading provides a hardware environment in which software can be optimized for multiple performance and energy objectives. Multithreading and concurrency function as knobs which

can be used to tune programs to run at power-efficient and performance-efficient operating points. This paper makes contributions towards effectively utilizing the concurrency knob of chip multithreading and layered multiprocessor architectures to achieve near optimal energy-efficiency. We presented a very fast and effective – in finding nearly optimal operating points – online performance prediction model in which performance, power and combined metrics are predicted with as few as two snapshots of hardware event counters, on a phase by phase basis. Our model overcomes the limitations of and outperforms earlier schemes which relied on exhaustive or heuristic searches of optimal operating points via direct timing analysis of phases. Furthermore, our prediction model is significantly more accurate than other similar models based on snapshots of hardware event counters.

In the future, we plan to extend our framework in two directions. The first is the integration of a more educated power prediction model, which will replace our simple relativistic model. The current model projects power across system configurations without taking into account program characteristics. We plan to use our hardware profiler for predicting power online, from snapshots of hardware event counters taken during phases. An interesting exercise in this context will be to investigate whether both power and performance can be predicted using the same set of hardware events. Our second target is to improve our framework in the handling of program phases with irregularities, including time-dependent and non-deterministic behavior. Although our current prediction model can be used effectively to capture a variety of phase characteristics, it operates under the assumption that phases exhibit invariant behavior across invocations. Capturing and modeling variability in phase characteristics will be a first step towards deploying our predictor in irregular codes.

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.

6. REFERENCES

- [1] M. Curtis-Maury, J. Dzierwa, C. Antonopoulos, and D. Nikolopoulos. Online Strategies for High-Performance Power-Aware Thread Execution on Emerging Multiprocessors. In *Proc. of the Second Workshop on High-Performance Power-Aware Computing*, Rhodes, Greece, April 2006.
- [2] M. DeVuyst, R. Kumar, and D. Tullsen. Exploiting Unbalanced Thread Scheduling for Energy and Performance on a CMP of SMT Processors. In *Proc. of the 20th IEEE/ACM International Parallel and Distributed Processing Symposium*, Rhodes, Greece, April 2006.
- [3] L. Eeckhout and K. De Bosschere. Statistical Simulation of Superscalar Architectures using Commercial Workloads. In *Proc. of the Fourth Workshop on Computer Architecture Evaluation using Commercial Workloads (in conjunction with HPCA-7)*, Monterrey, Mexico, January 2001.
- [4] G. A. Grell, J. Dudhia, and D. R. Stauffer. A Description of the Fifth-Generation Penn State/NCAR Mesoscale Model (MM5). NCAR Technical Note NCAR/TN-398 + STR, National Center For Atmospheric Research (NCAR), June 1995.
- [5] C. Isci and M. Martonosi. Runtime Power Monitoring in High-End Processors: Methodology and Empirical Data. In *Proc. of the 36th ACM/IEEE Annual International Symposium on Microarchitecture*, pages 93–104, San Diego, CA, November 2003.
- [6] H. Jin, M. Frumkin, and J. Yan. The OpenMP Implementation of NAS Parallel Benchmarks and its Performance. Technical report nas-99-011, NASA Ames Research Center, October 1999.
- [7] C. Jung, D. Lim, J. Lee, and S. Han. Adaptive Execution Techniques for SMT Multiprocessor Architectures. In *Proc. of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 236–246, Chicago, IL, June 2005.
- [8] R. Kalla, B. Sinharoy, and J. Tendler. IBM POWER5 Chip: A Dual-Core Multithreaded Processor. *IEEE Micro*, 24(2):40–47, March 2004.
- [9] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE MICRO*, 25(2):21–29, March/April 2005.
- [10] D. Koufaty and D. Marr. Hyperthreading Technology in the Netburst Microarchitecture. *IEEE Micro*, 23(2):56–65, March 2003.
- [11] J. Li and J. Martínez. Dynamic Power-Performance Adaptation of Parallel Computation on Chip Multiprocessors. In *Proc. of the 12th International Symposium on High-Performance Computer Architecture*, Austin, TX, February 2006.
- [12] C. Liu, A. Sivasubramaniam, M. Kandemir, and M. Irwin. Exploiting Barriers to Optimize Power Consumption on CMPs. In *Proc. of the 19th International Parallel and Distributed Processing Symposium*, Denver, CO, April 2005.
- [13] J. Lo, J. Emer, H. Levy, R. Stamm, D. Tullsen, and S. Eggers. Converting Thread-Level Parallelism to Instruction-Level Parallelism via Simultaneous Multithreading. *ACM Transactions on Computer Systems*, 15(3):322–353, August 1997.
- [14] 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.
- [15] T. Moseley, J. Kim, D. Connors, and D. Grunwald. Methods for Modelling Resource Contention on Simultaneous Multithreaded Processors. In *Proc. of the 2005 International Conference on Computer Design*, pages 373–380, San Jose, CA, October 2005.
- [16] M. Pettersson. A Linux/x86 Performance Counters Driver. <http://user.it.uu.se/~mikpe/linux/perfctr/>.
- [17] A. Snively and D. Tullsen. Symbiotic Job Scheduling for a Simultaneous Multithreading Processor. In *Proc. of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'IX)*, pages 234–244, Cambridge, Massachusetts, November 2000.
- [18] 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, October 2002.
- [19] 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, April 2005.