Prediction-based Power-Performance Adaptation of Multithreaded Scientific Codes

Matthew Curtis-Maury, Christos D. Antonopoulos, Filip Blagojevic, and Dimitrios S. Nikolopoulos

Abstract*— Computing is currently at an inflection point, with the degree of on-chip thread-level parallelism doubling every one to two years. The number of cores has become one of the most important architectural parameters that characterize performance and power-efficiency of a modern microprocessor, and a computer system in general. Concurrency lends itself naturally to allowing a program to trade some of its performance for power savings, by regulating the number of active cores. Unfortunately, in several computing domains, users are unwilling to sacrifice performance to save power. Futhermore, the opportunities for saving power via other means, such as voltage and frequency scaling, may be limited in heavily optimized applications. In this paper, we present a prediction model for identifying energy-efficient operating points of concurrency in well-tuned multithreaded scientific applications, and a runtime system which uses live analysis of hardware event rates through the prediction model, to optimize applications dynamically. The runtime system throttles concurrency so that power consumption can be reduced and performance can be set at the knee of the scalability curve of each parallel execution phase. We present a dynamic, phase-aware performance prediction model (DPAPP), which combines multivariate regression techniques with runtime analysis of data collected from hardware event counters, to locate optimal operating points of concurrency. DPAPP is hardware-aware, in the sense that it takes into account the dimensions of parallelism in the architecture, using distinct predictors and hardware events for each dimension. It is also phase-aware. Using DPAPP, we develop a prediction-driven runtime optimization scheme, which drastically reduces the overhead of searching the optimization space for power-performance efficiency, while achieving near-optimal performance and power savings in real parallel applications.

***Index Terms*— Modeling and prediction, Application-aware adaptation, Energy-aware systems**

I. INTRODUCTION

Microprocessors are currently at an inflection point, where clock rates and instruction-level parallelism have been replaced by the number of execution cores, as the key metric that characterizes the performance and drives the marketability of a computer system. Moore's law is now interpreted as "the number of cores on a microprocessor is expected to double every one to two years", and hardware vendors race for the most cores that can be packaged on a single chip [21], [25].

In the new landscape of highly parallel microprocessors and system architectures, system software appears to be largely unprepared for the transition. The programming effort required for parallelizing and optimizing code practically remains an unresolved issue, even among research communities that have been investigating this problem for decades. At the same time, power dissipation is now a major consideration for system software optimization on parallel architectures [10]–[13]. The introduction of many simple cores on a microprocessor has been largely motivated by the poor power-efficiency of microarchitectural components that attempt to improve performance at the cost of hardware complexity and reliability [3]. Concurrency not only improves power efficiency, but also helps system software steer power and performance simultaneously. The conventional wisdom holds that when concurrency is increased, performance is improved, but with an associated increase in power consumption. Conversely, when concurrency is decreased, power consumption is reduced, at a cost for performance.

While there are many situations where it is desirable to trade performance for reduced power consumption, in the domain of high-performance scientific computing, performance remains the primary target and energy may be a second tier concern for end users. Applications written for high-end computing systems create a challenge for energy-aware system software, which needs to identify opportunities to reduce power consumption with a non-negative impact on performance. In well-tuned, heavily optimized scientific applications, idle periods or long memory latencies, two known opportunities for performance-aware power reduction via dynamic voltage and frequency scaling [23], [29], may not arise as frequently, or not be as long as needed to enable substantial power reduction. Programmers will usually do their best to eliminate idling and minimize memory access latency via load balancing and extensive data caching optimizations respectively.

On the other hand, there are certain cases where inherent program characteristics –such as limited algorithmic concurrency, fine computational granularity, and frequent synchronization– and architectural properties –such as capacity limitations of shared resources– limit the scalability and the maximum degree of exploitable concurrency in an application, resulting in an observed performance *loss* through the use of *more* parallelism. In these cases, power and performance can be simultaneously improved by *throttling* concurrency.

To motivate the work presented in this paper, Figure 1 shows a breakdown of the parallel execution time of three applications from the NAS Benchmarks Suite [18] into phases. The breakdowns were obtained during execution of the benchmarks on a quad-processor server with Intel Xeon processors using Hyperthreading technology. Each chart depicts the $(processors, Hyperthreads/processor)$ configuration that minimizes the execution time of each phase. The fastest configuration is identified experimentally, by executing each target phase in all possible hardware configurations of the system. LU-HP-B, SP-A and MG-B execute optimally with at least one Hyperthread per processor deactivated, thus saving power while simultaneously improving performance, during 95%, 84% and 81% of their parallel execution times respectively. LU-HP-B and SP-A execute with at least one entire processor deactivated during 40% or more

Matthew Curtis-Maury, Filip Blagojevic, and Dimitrios S. Nikolopoulos are with the Center for High End Computing Systems at Virginia Tech.

Christos D. Antonopoulos is with the Department of Computer Engineering and Communications at the University of Thessaly.

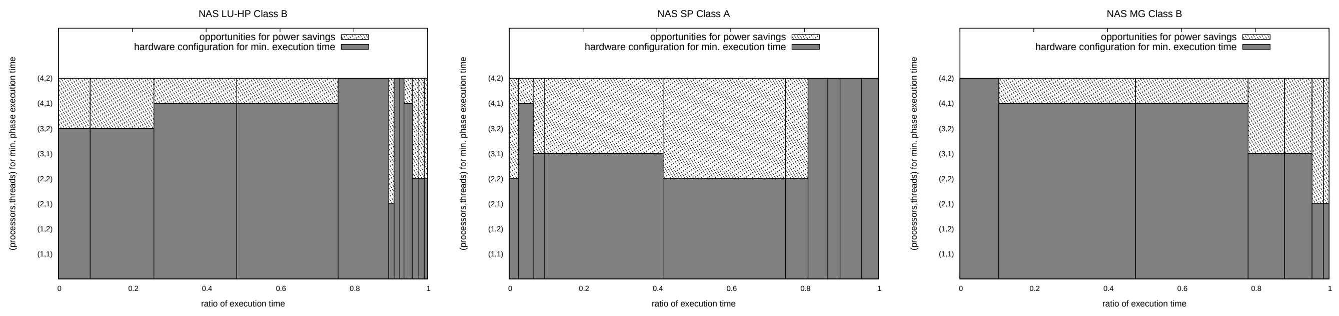

Fig. 1. Breakdown of parallel execution time of three applications from the NAS Benchmarks Suite [18], on a four-processor server with Intel Hyperthreading processors. Each phase is represented with a gray parallelogram. The length of the phase and the hardware configuration (#processors, #hyperthreads/processor) that minimize the execution time of the phase correspond to the width and height of each parallelogram respectively. Lightly shaded areas illustrate opportunities for concurrency throttling.

of the optimal execution time. Concurrency throttling has the advantage that it can be applied to well-tuned, compute-intensive phases of an application, that otherwise provide limited opportunities for power optimization. Concurrency throttling tends to drastically reduce power consumption. Whenever it also reduces execution time, it achieves a multiplicative reduction on the total expended energy.

Despite its appeal, concurrency throttling is an opportunity which may present itself to varying degrees across different programs, across different phases of the same program, or even across different inputs fed to the same program. Identifying concurrency throttling opportunities statically is hard, because it requires fine-grain analysis of the dynamic behavior of parallel code across and within parallel execution phases. Besides the problem of identification and quantification of the opportunities, applying concurrency throttling directly in applications requires exposure of the programmer to architectural details, such as the number and physical layout of processors. This tactic is widely considered as one of the factors that make parallel programming exceptionally difficult [9]. Given the complexity along with the inherent drawbacks of delegating concurrency throttling decisions to the user or to a static analysis tool, runtime systems appear to be ideal candidates for the identification and exploitation of concurrency throttling opportunities.

This paper presents a dynamic program concurrency controller, which seeks the optimal operating point of concurrency in multithreaded programs, at the granularity of program phases. In contrast to concurrency control schemes based on live empirical search of operating points, our controller relies on a dynamic, phase-aware performance prediction (DPAPP) model. The model predicts the optimal operating point of concurrency on different configurations of processors, cores, and threads, here on referred to simply as *hardware configurations*. The key contribution of the DPAPP model is that it enables drastic reduction of the overhead associated with searching the optimization space for concurrency throttling, henceforth facilitating rapid and efficient program adaptation. DPAPP uses live input from hardware event counters, collected while executing program phases on operating points of maximal concurrency. We use a multivariate regression process for selecting the critical hardware events that best predict performance, and for training the DPAPP model in assessing the scaling effects that changing hardware configurations have on overall program throughput. The DPAPP training process derives distinct predictors for thread-level, core-level and processor-level parallelism, to account for the presence of multidimensional parallelism and variance in the impact of sharing of resources between parallel processing units within and across chip boundaries. We use the DPAPP model to steer a runtime concurrency controller, which succeeds in identifying phases where power consumption can be conserved while sustaining or improving performance. We demonstrate the effectiveness of our controller using the full NAS Benchmarks suite, on a shared-memory multiprocessor composed of SMT processors.

The rest of this paper is organized as follows. In Section II, we discuss background and related work. Section III introduces our model for dynamic, phase-aware performance prediction of parallel applications. Section IV presents our control scheme for dynamic, power-aware and performance-aware concurrency adaptation of multithreaded codes. We present a detailed discussion of our experimental methodology and results in Section V. Finally, we conclude this paper in Section VI.

II. Related Work

Much previous research has been performed on optimizing the execution of programs using feedback from hardware event counters (HECs), however it has predominantly been offline, profile-guided in nature. For example, NUMA multiprocessor page placement using hardware assistance [31], CPO (Continuous Program Optimization) from IBM which includes management of variable page-size systems [4], and case studies of specific applications [2]. In contrast, little work has been done on runtime optimization utilizing hardware counters as the program executes. Existing examples include HEC-based SMT job schedulers [32] and the ADORE runtime optimization system [30]. Our work falls into the category of online dynamic optimization with feedback from hardware counters, however it targets energy consumption, in addition to performance.

Performance prediction of parallel programs has been studied in great depth, however the majority of research is targeted at offline prediction. Due to space limitations, we cannot here fully discuss previous work in this mature area. Work most similar to ours includes offline research on partial execution-based prediction [37] and statistical simulation of superscalar processors using IPC predictions based on very short code samples [8]. Minimizing design space evaluation time for processor development has spurred much research on predicting the performance effects of altering various microarchitectural parameters, including regression-based [26] and machine learning-based approaches [16], [19]. To our knowledge, no prior work has considered online predictors of parallel execution performance on shared-memory architectures, using runtime input on IPC and hardware event counts.

High-performance, power-aware computing has recently become an important topic of research. Efforts range from power-scalable and power-efficient clusters [10], [11] to runtime systems providing support for dynamic frequency and voltage scaling for parallel applications [12], [23]. Our work is most closely related to the latter, as both attempt to identify opportunities at runtime to achieve power savings without sacrificing performance. Our work differs in that we target shared-memory rather than distributed memory multiprocessors. Additionally, DVFS still faces hard technological contraints before being applied per-thread in multicore chips [27], while concurrency throttling is immediately deployable on all forms of emerging processors composed of multiple cores and hardware threads. Finally, it should be pointed out that DVFS and concurrency throttling are not necessarily at odds with each other as they may be applied in a synergistic fashion to achieve still greater energy-efficiency [28].

Concurrency control has been previously applied for optimization of multithreaded codes on shared memory multiprocessors. Specifically, concurrency control can enable adaptive execution in multiprogramming environments [1], [35], [38]. Further, standalone programs can benefit from concurrency control across different phases with potentially different execution and scalability characteristics [15], [39]. In most cases, concurrency control is applied in a given phase by the programmer, the runtime system, or the compiler. Compiler-based control is generally performed using a simple threshold-based strategy and the parallel code region is either sequentialized or run with a programmer-specified fixed number of threads [15], [20], [36]. Programmmers have long had the ability to manually specify concurrency levels, however few runtime systems provide the functionality to autonomically manage these decisions from within. Our work provides such a system, offering fully autonomic concurrency control based on performance predictions of each configuration.

Recent work has considered applying concurrency control and DVFS on single chip multiprocessors, with decisions utilizing search algorithms of the configuration space [28]. This research shares many motivations with our work, mainly maintaining performance while reducing power consumption, however the suggested solutions to the problem differ significantly. First, we do not explore the potential of DVFS, but we rather introduce a solution that works on architectures independently of their support for DVFS functionality. Second, our approach is implemented on a real system, rather than simulated, verifying that our technique works in practice with all overheads considered. Third, we utilize performance prediction rather than empirical searches of the configuration space to reduce the number of test executions necessary to perform adaptation. Further, we show that the overhead of search based techniques hinders the performance of short-lived codes, particularly when compared to prediction. Additionally, our approach targets multiprocessor systems where the combined energy consumption of the processors plays a much larger role than in uniprocessor systems such as that evaluated in [28].

III. Dynamic Phase-Aware Performance Prediction

The goal of dynamic phase-aware performance prediction (DPAPP for short) is to predict the performance of a multithreaded, compute-intensive region of code in a program –which we hereafter refer to as a *phase*– across varying configurations of the processing units on a parallel architecture [5]. We use the term *processing units* as an umbrella term covering hardware threads, scalar or superscalar processor cores, and single- or multiple-chip uni- or multi-processors. As a base hardware substrate, we consider shared-memory multiprocessors with three distinct types of processing units, namely multi-core processors, cores within multi-core processors and threads within multi-threaded cores. We refer to each of these types of processing units as a *dimension of parallelism* in the system. More formally, we define a *dimension of parallelism* as a set of homogeneous processing units that share a given level of the memory hierarchy, which is also shared by processing units nested in lower dimensions of parallelism, but not shared by processing units in higher dimensions of parallelism. In principle, each dimension of parallelism shares a distinct set of execution and memory resources, and therefore exhibits distinct scalability properties. The dimensions of parallelism that we consider are representative of current commercial multiprocessors [21], [25]. Our DPAPP technique considers phases that are identified as parallel loops, as these structures encapsulate the bulk of parallel code in real scientific applications.

Our DPAPP model works by predicting the cumulative *useful* Instructions Per Cycle ($uIPC$) of multithreaded phases. $uIPC$ is defined as the sum of IPCs of the threads used to execute a phase, excluding instructions and cycles expended for synchronization and parallelization. Ignoring parallelization and synchronization overheads makes $uIPC$ inversely proportional to the execution time of a fixed number of instructions on a given hardware configuration. Note that although $uIPC$ ignores instructions for triggering and synchronizing threads, it still considers the effects of interference between threads on shared hardware resources during concurrent execution. The objective of DPAPP is to identify phases where concurrency can be reduced during the execution of useful application computation, with a non-negative impact on performance. The use of $uIPC$ as a prediction target focuses the optimization process on lengthy, compute-intensive parts of applications, where power optimization opportunities may be limited with means other than concurrency throttling.

A. DPAPP Outline

DPAPP makes distinct predictions on the optimal number of processing units to use at each dimension of parallelism in the system. For ease of presentation, we first describe the operation of DPAPP for a given dimension of parallelism d. We defer the discussion of how DPAPP predicts across dimensions of parallelism until Section III-E.

DPAPP takes input from live samples of hardware event counters. HECs are sampled at the beginning and end of each phase, while the phase is executed on the configuration that activates all processing units at dimension d. The set of hardware events sampled are specific to d and are selected using a formal statistical process, according to their contribution to $uIPC$. We refer to these events as *critical events*. Samples of critical event rates are fed to a model that estimates $uIPC$ per phase, per configuration, for all feasible configurations of processing units at dimension d. Intuitively, DPAPP attempts to predict how the rate of retirement of useful instructions, $uIPC$, will change in a phase when the number of processing units used to execute the phase changes. To make this prediction, DPAPP uses a multivariate regression model, which correlates observed event rates on a sample configuration and observed $uIPC$ values on all feasible hardware configurations during training runs. The model outputs a set of scaling factors for $uIPC$ and the critical hardware events, for each feasible hardware configuration. These outputs are used as constant coefficients during production runs, to predict optimal operating points of concurrency for each phase in the code. We describe the model in more detail in Section III-B and the process for training the model in Section III-C. The process for selecting critical events is discussed in Section III-D.

The objective of DPAPP is to produce performance predictions and adapt the code dynamically, as the program executes. Recall that a primary motivation behind DPAPP is the avoidance of the overhead of experimentally searching through hardware configurations to find optimal operating points for phases in the program. To minimize the prediction overhead and to achieve effective code adaptation as early as possible during execution, DPAPP samples HECs for a minimal number of phase traversals. Following phase traversals used for sampling hardware event rates, the runtime system selects the predicted optimal operating point of concurrency for each phase. To further tame overhead, DPAPP models performance as a linear function of event rates, so as to produce rapid predictions across a potentially large number of hardware configurations. By contrast, an exhaustive search algorithm would have to test $\prod_{d=1}^{D} p_d$ phase traversals, where p_d is the number of processing units in dimension d and D the number of dimensions of parallelism. A heuristic search algorithm would also have $\prod_{d=1}^{D} p_d$ worst-case complexity.

B. $uIPC$ Prediction Model

The DPAPP predictor estimates the $uIPC$ of a phase on a target configuration t (denoted as $\overline{uIPC(t)}$) using input from execution of the phase on a sampled test configuration s. The input from the sampled execution includes the actual $uIPC$ of the sampled configuration ($uIPC(s)$) and a set of n hardware event per cycle rates, $(e_1(s), ..., e_n(s))$. Each event rate $e_i(s), i = 1 \ldots n$ is the number of occurrences of event i divided by the number of elapsed clock cycles during the execution of the phase in test configuration s.

Although in theory, the DPAPP predictor can use any feasible configuration as a sample configuration, we heuristically chose to use the configuration where all processing units at the given dimension of parallelism are active. Intuitively, $uIPC$ and the critical event rates sampled in this configuration encapsulate the cumulative impact of hardware components on scaling, at maximum system capacity at the given dimension of parallelism.

We model $\overline{uIPC(t)}$ of the target configuration, as a linear function of $uIPC(s)$ of the source configuration, as:

$$\overline{uIPC(t)} = uIPC(s) \cdot \alpha(t, e_1(s), ..., e_n(s)) + \beta(t) \tag{1}$$

for a set of n critical hardware events, which may function either as enhancers, or as impediments of scalability. The selection of the events in this set is discussed further in Section III-D. Notice that both the scaling factor α and the residual β of the linear function are specific to and dependent on the target hardware configuration t. In other words, each target configuration t exerts its own scaling impact on $uIPC(s)$, which can be positive or negative. To gauge how individual critical events affect scalability, the linear scaling factor is in turn modeled as a linear combination of hardware event rates observed during the sampled configuration s:

$$\alpha(t, e_1(s), ..., e_n(s)) = \sum_{i=1}^{n} (x_i(t) \cdot e_i(s) + y_i(t)) + z(t) \tag{2}$$

The linear model of event rates stems from the empirical observation that a change in the configuration used to execute a program phase will result in changes – either upwards or downwards – of critical hardware event rates, reflecting the contention or effective hardware utilization at each level of parallelism. These event rates are linearly related – positively or negatively – with the $uIPC$. This relation is captured in Equation 2 with positive or negative event coefficients respectively. Our model attempts to estimate these coefficients using multivariate regression, discussed further in Section III-C.

Combining equations 1 and 2, the estimated $\overline{uIPC}$ for a target configuration t can be calculated as:

$$\overline{uIPC(t)} = uIPC(s) \cdot \sum_{i=1}^{n} (x_i(t) \cdot e_i(s)) + uIPC(s) \cdot \gamma(t) + \beta(t) \tag{3}$$

where $\gamma(t)$ is defined as $\sum_{i=1}^{n} (y_i(t)) + z(t)$. Accurate estimation of $\overline{uIPC}$ for a target configuration t is thus dependent on the proper approximation of the coefficients $x_i(t)$, $\gamma(t)$ and the residual $\beta(t)$. Note that the coefficients scale both the event rates and $uIPC$ of the sampled configuration s.

$\overline{uIPC}(t)$ values for all possible configurations are used directly for prediction of the optimal operating concurrency for each phase, at the given dimension of parallelism. We truncate $uIPC$ predictions that exceed the cumulative maximum capacity ($uIPC_{max}$) of all processing units at the given dimension of parallelism, to $uIPC_{max}$, which is derived experimentally for any given processor using microbenchmarks. Furthermore, we assume that there is no superlinear speedup across configurations of a phase, although this case does appear in real codes. In practice, phases with superlinear speedup have their optimal operating point of concurrency at the maximum number of processing units and offer no opportunity for concurrency throttling.

C. Offline Training and Estimation of Coefficients

We use multivariate linear regression on the multithreaded phases of a set of training benchmarks to determine the values of the coefficients in Equation 1. Although more advanced machine learning techniques could be deployed for prediction, the number of cycles invested in making predictions at runtime is a primary concern for DPAPP, therefore we opt for the simplest linear prediction model. Specifically, training benchmarks are executed under all feasible hardware configurations, at all dimensions of parallelism, while recording per-phase $uIPC$ and the critical hardware events used for prediction (see Section III-D). The training benchmarks are selected empirically so as to include phases with variance in three characteristics: scalability ranging from poor to perfect; granularity of parallel computation, ranging from fine to coarse; and ratio of computation to memory accesses, ranging from low to high. In our experimental evaluation, we use two parallel benchmarks (MM5, a mesoscale weather modeling code and the Unstructured Adaptive application from the NAS Benchmarks) with 119 phases in total. In this work, we define phases to be OpenMP parallel regions enclosing parallelized loops. The training benchmarks achieve excellent coverage of diverse phase characteristics.

Our multivariate regression analysis uses the events collected under the selected sample configuration s multiplied by the $uIPC$ of the sampled configuration, i.e. $e_i(s) \cdot uIPC(s)$, and the actual $uIPC$ alone ($uIPC(s)$) as *independent variables,* to predict the $uIPC(t)$ of each target configuration t as the *dependent variable*. We use the product of e_i and $uIPC$ of the sampled configuration for coefficient derivation because our model uses multiplicative effects of events on the observed $uIPC$ rather than additive ones, in accordance with Equation 3. This process estimates the necessary coefficients for each event in function $\alpha(t)$. Regression analysis is performed separately to predict $uIPC$ for each target configuration t, therefore we derive independent sets of coefficients and independent scaling factors for each target configuration. For a system with p_d units in dimension d of parallelism, $1, \ldots, D$, multivariate regression analysis derives a total of $\sum_{i=1}^{D} p_d$ sets of coefficients.

D. Rigorous Event Set Selection for $uIPC$ Prediction

The accuracy of DPAPP is heavily dependent upon the selection of an effective set of critical events for predicting performance and scalability along each dimension of parallelism. The events should accurately reflect, in a statistical sense, performance and scalability bottlenecks in the system. We have previously considered empirical selection of events that represent known performance-critical components of microprocessors [5]. In this paper, we present a rigorous statistical technique, which automates the event selection process and makes it reproducible and generally applicable to any target architecture.

Modern processors generally provide very large sets of events that can be recorded. Furthermore, a microprocessor can typically record multiple events at the same time. For example, Intel Pentium 4's provide 40 events which can be further differentiated by specifying bitmasks to each event, and up to 18 events can be recorded at once. The IBM Power5 provides 500 events and permits up to 6 to be recorded simultaneously. The number of legal sets of events that can be recorded simultaneously on these architectures is far too large for it to be feasible to exhaustively test each set of events as input for prediction. Moreover, while the most effective prediction possible would likely result from the use of all (or at least most) available events, there is an architectural limit on how many events can be recorded simultaneously, and often there are further restrictions on which events can be recorded at the same time.

Rather than exhaustively looking at each possible combination of events, our predictor training tool independently looks at the contribution of each event to $uIPC$. To gauge each event's significance, we initially use multivariate regression on data from the set of training benchmarks to predict $uIPC(t)$ for each target configuration, using all events that are available for monitoring on the processor. We model $uIPC$ as in Equation 3, with the exception that we use a set of N events where $N >> n$.

Following the initial $uIPC$ modeling phase, we prune all events that have zero or negligible occurrence rates. We then consider the contribution of each event to the resulting $uIPC(t)$ prediction, as a percentage of $uIPC(t)$. The contribution of each event is calculated by multiplying the event rate by its coefficient and by $uIPC(s)$ and dividing the result by $uIPC(t)$. We average the contributions of each event across all feasible configurations and all phases in the training runs, and rank the events in descending order of contribution. The actual number of events selected for prediction (n) is processor-dependent. We set n to be the maximum number of events that the hardware performance monitor of the processor can count simultaneously, without time-multiplexing of event registers. This selection criterion minimizes the overhead of monitoring hardware events for prediction. Note that on architectures where dependencies between events prevent simultaneous monitoring of specific sets of events, some critical contributing events may still be left out of the predictor due to conflicts with other, more heavily contributing events.

E. Prediction on Architectures with Multiple Dimensions of Parallelism

On architectures with multiple dimensions of parallelism, resource sharing varies considerably across dimensions. For example, physical processors in an SMP share only the off-chip interconnection network and DRAM. Cores within a processor typically share an on-chip interconnection network and the outermost levels of the on-chip cache. Threads on a single core share most resources of the execution core, including pipelines, branch predictors, TLB and L1 cache. Contention for these shared resources is largely responsible for performance and scalability.

To capture the implications of multidimensional parallelism, DPAPP uses a distinct set of critical events and derives a distinct set of scaling factors for each dimension of parallelism in the system. DPAPP repeats the processes outlined in Section III-B and Section III-D, to obtain prediction event sets and coefficients for each dimension of parallelism. At actuation time, DPAPP makes predictions along each of the dimensions of parallelism and combines these predictions to yield a power-efficient concurrency operating point for each phase in the program.

F. Predictor Optimization

The accuracy of DPAPP is significantly improved by classifying code phases according to their observed $uIPC$ during the execution of the sample configuration. The justification for such an extension is twofold. First, grouping phases based on $uIPC$ allows training and prediction to occur separately for phases with different scalability slopes. As such, the division between buckets is selected such that it divides different degrees of scalability. Second, it is intuitive that the effects of events will vary depending on the original instruction throughput of each phase. Dividing the phases into buckets and creating separate $\alpha(t)$ scaling functions for each class of phases gives the predictor the opportunity to make more fine-grain and accurate predictions. At runtime, the observed $uIPC$ on the sample configuration determines which set of coefficients will be used for prediction. We use this optimization in our implementation of DPAPP.

IV. Concurrency Control for Performance and Power Optimization

In this section we present our phase-aware concurrency control algorithm for a 2-layer shared-memory multiprocessor, such as a multi-chip multi-processor with multi-core processors. We then discuss the power and energy reduction potential of the algorithm and extensions to the algorithm that take account for inter-phase interference.

```
1:  {**Input**: phase identifier, sampling rate}
2:  {**Output**: predicted optimal operating concurrency, $c_{max}$}
3:  {Assumes 2-dimensional multiprocessor with $P_0 \cdot P_1$ processors.}
4:  {Each tuple $\{p_0, p_1\}$ represents a hardware configuration.}
5:  $S \leftarrow sampling\_rate$; $c_{max} \leftarrow \{P_0, P_1\}$; $uIPC_{max} \leftarrow 0$;
6:  **for all** $i, 1 \leq i \leq S$ **do**
7:      $c_{max,i} \leftarrow \{P_0, \frac{i}{S} \cdot P_1\}$;
8:      sample $uIPC(c_{max,i})$;
9:      sample event rates of $c_{max,i}$;
10:     $uIPC_{max,i} \leftarrow uIPC(c_{max,i})$;
11:     **for all** $j, 1 \leq j \leq P_0$ **do**
12:         $c \leftarrow \{j, \frac{i}{S} \cdot P_1\}$;
13:         predict $\overline{uIPC(c)}$;
14:         **if** $\overline{uIPC(c)} > uIPC_{max,i}$ **then**
15:             $uIPC_{max,i} \leftarrow \overline{uIPC(c)}$; $c_{max,i} \leftarrow c$;
16:         **end if**
17:     **end for**
18:     **if** $(uIPC_{max,i} > uIPC_{max})$ **then**
19:         $c_{max} \leftarrow c_{max,i}$; $uIPC_{max} \leftarrow uIPC_{max,i}$;
20:     **end if**
21: **end for**
```

Fig. 2. DPAPP-driven concurrency controller algorithm for an architecture with 2-dimensional parallelism.

A. DPAPP Concurrency Controller

Scientific codes are dominated by iterative execution of phases, and DPAPP exploits this property to sample hardware event rates in the first few phase traversals and set the concurrency of each phase to the predicted optimal operating point, early during execution of the program. The live search of the optimization space for operating points of concurrency can also be performed by timing phases at different configurations and running search heuristics such as greedy hill-climbing [6], [28] or simulated annealing [24]. However, as the number of feasible hardware configurations increases with the introduction of more cores and threads per processor, direct search methods may spend most of the execution time sampling suboptimal configurations, rather than optimizing the program. This disadvantage manifests itself in codes where dominant multithreaded phases are traversed only a few times. Even if direct search methods are used for off-line auto-tuning by repetitive executions of the entire program [9], searching the program optimization space for any input on any feasible configuration of processing units may be prohibitive. DPAPP prunes the search space for concurrency optimization to a constant number of samples.

Figure 2 illustrates a DPAPP-driven concurrency control algorithm for a multiprocessor with two dimensions of parallelism. The DPAPP concurrency controller estimates optimal operating points of concurrency, using samples of critical hardware event rates from live executions of program phases. The controller is dynamic, in the sense that it adapts the program as it executes, with no prior knowledge of program characteristics. The DPAPP technique is used by the controller so that predictions of optimal concurrency points are derived from a small number of samples of hardware event rates. The number of samples is a tunable parameter of the control algorithm. In our prototype, we use a sample rate of $S = 2$ taken along the innermost dimension of parallelism, i.e. threads within a processor. The algorithm in Figure 2 generalizes to more than two dimensions by repeating the loop in lines (11)–(17) for each dimension beyond the second, while saving the current predicted optimal configuration in a temporary variable.

Once predictions for a phase are obtained, all subsequent traversals of a phase are executed at the predicted optimal operating point of concurrency, which is set by the controller. The DPAPP concurrency control algorithm has two parameters, the sampling rate and the dimension of parallelism along which the initial samples are taken. The second parameter is fixed at the training phase of the DPAPP predictor, during which all possible orderings of dimensions of parallelism can be tested. The sampling rate S is chosen so as to control the overhead of sampling. The sampling rate corresponds to the number of times each phase needs to be executed before deriving a prediction for the optimal operating point. Our choice, $S = 2$, provides the minimum number of samples needed to capture the effects of using more than one core or thread on a multi-core or multi-threaded processor.

Certain assumptions are necessary to implement our concurrency controller, and we outline those in the following. First, we rely on the capability of the runtime system to change the number of threads used to execute a phase of parallel code at runtime. This capability is available in OpenMP, at the granularity of parallel loops and parallel regions. However, changing the number of threads at runtime may not be possible in some applications due to data initialization which depends on the number of threads used. This pattern is uncommon and is trivial to modify. Second, the phases of an application must be executed at least S times, to allow for sampling. Finally, the execution properties of each phase between executions must remain relatively stable. In practice, this is the case in both regular and irregular codes.

B. Energy Savings Possibilities

Energy savings using adaptive concurrency throttling come through two avenues. First, by reducing execution time, because the energy consumed is reduced proportionally. Second, through the deactivation of processing units, which reduces power consumption. The power consumption of a processing unit is dependent upon its level of utilization, as clock-gating limits the power dissipation of functional units when they are idle. Further, a processor can be transitioned to a lower power mode when it is not being used. For example, on Intel Pentium 4 processors, the *hlt* instruction transitions the processor to a low power mode, where power consumption is reduced from approximately 9W when idle to 2W when halted. While we do not manually control the transitioning between power states of the processors from within the runtime system, the operating system does so when the processor remains inactive for some time period. We have experimentally verified that in Linux 2.6 kernels, processors are actually transitioned to the halted state by the operating system during 90% of the time during which they have been left idle. Manually transitioning processors would result in minimal additional power savings, so we do not consider this direction further in this work. Moreover, instructions that transfer the processor to the halted state are usually privileged and can be executed only by the operating system.

C. Cross-Phase Decision Making

The processes of prediction, decision making, and adaptation are not performed at whole-program granularity, rather, each phase of an application is analyzed independently. This allows phases with different execution properties in the same application to execute with their own, locally optimal hardware configurations. Since many programs have behavior that varies across phases [34], overall performance can be improved compared to using a single configuration for the entire program. However, a non-negligable performance penalty may be paid as a result of changing the hardware configuration across adjacent phases at runtime. This performance penalty stems primarily from migration of working sets of threads between caches [22]. To avoid negative inter-phase interference, we consider variants of our adaption scheme that are aware of this interference.

We have developed two schemes for cross-phase prediction. The first of these schemes simply finds the configuration that is best for the majority of the application's phases, and applies this to all phases, regardless of their locally optimal configuration. This scheme avoids cache interference entirely, at the expense of using a single configuration for all phases and missing fine-grain optimization opportunities. The second approach is an extension to the first, where phases are allowed to temporarily replace the global optimal configuration with their local optimal configuration, only if IPC improvement beyond a preset threshold is predicted by using the local decision. Using this technique, interference will only be tolerated when the phase in question is expected to make up for it in performance gain through the use of an alternative configuration.

V. Evaluation

In this section we perform an evaluation of both the performance prediction model and the adaptive concurrency control technique presented in previous sections. In the next subsection we present the experimental setup that we used in our evaluation. Following, we present the results of event selection for prediction and the resulting accuracy of the predictor. Finally, we compare the power and performance results of the *Performance Prediction-based Adaptive Concurrency Controller* (PPACC) with those attained by runtime auto-tuning techniques based on empirical search and by off-line auto-tuning techniques based on static execution with no concurrency control.

A. Experimental Setup

We performed all of our experimental evaluations on a Dell PowerEdge 6650 server composed of four Intel Hyperthreaded Xeon processors with 1GB of main memory. Each processor is a 1.4 GHz, 2-way SMT equipped with an 8-KB L1 data cache, a 12-KB trace cache, a 256-KB L2 cache, and a 512-KB L3 cache. The Linux kernel used was version 2.6.15.

Experiments were performed with 10 benchmarks that are representative of scientific and engineering applications typically requiring high performance. Nine of the benchmarks originate from the OpenMP version of the NAS Parallel Benchmarks suite, version 3.1 [18]. We use three different problem sizes, available in the NAS distribution (W, A, B). MM5 is an OpenMP implementation of a mesoscale weather prediction model [14]. The benchmarks include a wide variety of program properties, and in particular, widely varying $uIPC$ scalability across execution phases. Therefore, they are challenging targets for prediction. The benchmark suite includes several benchmarks with a small number of iterations (CG, FT, IS, MG), in which empirical search strategies may suffer due to a large percentage of total execution time being spent in exploration, as well as benchmarks with a large number of iterations (BT, LU, LU-HP, SP, UA, MM5), where search strategies stand to have their search overheads better amortized. Results for FT are not included for class size B, because its working set does not fit in the available memory of our hardware platform.

Table I lists the benchmarks along with some pertinent information about their structure. The number of iterations, phases, and percentage of time spent in parallel regions shown are for class size A. The table also outlines the percentage of execution time during which at least one processor can be deactivated with non-negative impact on performance (i.e. the program runs optimally with at most 3 processors) and percentage of execution time during which one Hyperthread per processor can be deactivated with non-negative impact on performance (i.e. the program run optimally with at most one Hyperthread per processor), averaged over all three class sizes. This information is taken from static executions on all feasible hardware configurations.

B. Performance Prediction Evaluation

In order to evaluate our performance prediction model, we selected two benchmarks for training, specifically UA (compiled to class size A) and MM5. These benchmarks were selected because the phases they contain have widely varying execution properties, including IPC, scalability, and locality. Further, they contain enough phases to serve as a standalone training set. These applications were used in the event selection process as well as the predictor training. Predictions were made for the remaining benchmarks, i.e. all remaining NAS benchmarks with class sizes W, A, and B.

1) Event Selection: Selection of an effective set of events to use for performance prediction requires data for all of the available hardware counters on each of the test configurations for all of the training benchmark phases. Further, the $uIPC$ values of all phases on each hardware configuration are necessary as well. There are 40 events on Pentium 4 processors that can be recorded using only a single register each, with further differentiation within each event through the use of bitmask parameters specifying, for example, to record L2 cache misses, hits, or accesses. There is also an event to count memory accesses which requires two counter registers. We select one bitmask for each event representing the hardware parameter most likely to have the largest effect on performance, leaving 41 events to consider. Of these, 13 had rates near zero, and were thus removed as described in section III-D. The performance monitoring unit of the Pentium 4 with Hyperthreading technology shares the 18 counter registers between the two co-executing threads, leaving 9 counters available for each thread. The 28 events that survived pruning provide a total of 99,372 possible architecturally legal sets of events that can be recorded on the 9 performance counter registers per thread.

Regression analysis was performed on the data from each phase to find the events that contributed the most to the resulting IPC prediction. Table II displays the set of events that was selected for each prediction on our platform. In this discussion, configuration $(nproc, nthr/proc)$ denotes a configuration with $nproc$ processors and $nthr/proc$ threads per processor. It should be pointed out that events with large contributions have been excluded due to conflicts with more dominant events. That is, the inclusion of one highly contributing event often eliminates other contributing events that interfere with it. All that can be done in these cases

Benchmark	BT	CG	FT	IS	LU	LU-HP	MG	SP	*UA*	*MM5*
Iterations	200	15	6	10	250	250	4	400	200	180
Phases	5	5	5	1	3	11	6	9	49	70
% Time in Phases	99.5	91.6	91.2	79.7	99.9	99.7	86.3	99.6	99.8	95.5
% Time Disable CPU	1.9	33.3	0.1	100.0	0.0	15.1	6.0	35.1	59.3	7.7
% Time Disable SMT	99.1	66.6	93.0	100.0	0.0	50.8	53.5	32.9	33.1	70.0

TABLE I
THE SET OF BENCHMARKS WE USED TO EVALUATE ONLINE PERFORMANCE PREDICTORS FOR POWER-PERFORMANCE ADAPTATION, ALONG WITH THEIR MAIN PHASE CHARACTERISTICS.

is to select the event with the largest contribution and ignore the conflicting events. Specifically, three of the top five events on this architecture cannot be included because they conflict with the top two events. This suggests that on architectures where there are reduced or no dependencies between events, our prediction approach will likely achieve higher accuracy.

Predictor	**(4,2)→(*,2)**	**(4,1)→(*,1)**
Event0	Cycles Active	Cycles Active
Event1	L2 Cache Misses	L2 Cache Misses
Event2	Branches Retired	Branches Retired
Event3	UOP Queue Writes	TC Deliver Mode
Event4	Memory Cancels	Memory Cancels
Event5	Packed SP UOPs	Packed DP UOPs
Event6	Memory Accesses (1)	Machine Clears
Event7	Memory Accesses (2)	Stall Cycles
Event8	Instructions Retired	Instructions Retired

TABLE II
THE INTEL PENTIUM 4 HARDWARE EVENTS SELECTED FOR EACH PREDICTION TYPE. THE SECOND AND THIRD COLUMNS SHOW THE EVENTS FOR PREDICTING THE OPTIMAL CONFIGURATION WITH 2 AND 1 HYPERTHREADS ACTIVATED PER PROCESSOR RESPECTIVELY.

2) Prediction Accuracy: We perform our evaluation of the accuracy of the online performance predictor using eight of our ten benchmarks, excluding the two benchmarks used for training the predictor. We consider the absolute prediction error and the configuration prediction error for each benchmark. We calculate the absolute prediction error as $|uIPC_{pred} - uIPC_{obs}|/uIPC_{obs}$, where $uIPC_{obs}$ is the observed IPC of useful instructions. On our experimental platform, there are six predictions made for each phase. Specifically, we predict for configurations with one and two Hyperthreads per processor on one, two, and three processors. The average prediction error for each phase is taken across all target configuration predictions. Configuration prediction accuracy compares the predicted optimal configuration for each phase with the *local static optimal configuration*. The local static optimal configuration is obtained as follows: We execute the benchmarks with each of the eight possible hardware configurations statically, i.e. with no concurrency control between phases. For each phase, we designate as optimal the one configuration out of the eight possible that minimizes the execution time of the phase. Note that this definition of optimal configuration ignores inter-phase interference and that a local static optimal configuration may or may not be the global, program-wide static optimal configuration. Configuration prediction accuracy illustrates how often the predictor identifies the local static optimal configuration. It should be noted that $uIPC$ prediction is particularly challenging on our experimental platform, because often, $uIPC$ changes due to Hyperthreading cannot be approximated as a linear function of the number of processors and threads used. However, we should also note that the litmus test for our predictor is not $uIPC$ prediction accuracy but configuration prediction accuracy. As long as the predictor correctly predicts the optimal configuration for each phase, a potentially high $uIPC$ prediction error can be disregarded.

As discussed in Section III-F, we utilize phase classification before making predictions. Specifically, we divided phases into buckets with $uIPC$ greater than or equal to 1.0 and those less than 1.0 during the sample configuration. This division is not arbitrary, rather, it provides an approximate value to separate phases with low scalability characteristics versus those that scale well, in general, on this architecture. During prediction, each phase uses the coefficients derived from the $uIPC$ bucket corresponding to its observed $uIPC$ during the sample configuration.

The $uIPC$ prediction accuracy can be seen in the leftmost graph of Figure 3. This graph gives the cumulative distribution function of prediction error, that is, the percent of phases that experience error below each threshold with threshold samples taken every 5%. The overall average absolute prediction error is 18.6%. We note that 24% of all predictions have less than 5% error and 43% of all predictions have less than 10% error. On the other hand, only 4% of the predictions show error larger than 50%. Although our performance prediction model is purposefully simple to minimize the overhead of applying it at runtime, its results compare favorably with other reported statistical techniques for predicting IPC [7].

In terms of prediction of the optimal configuration for each phase, the middle chart of Figure 3 shows the percent of phases for which each possible ranking of configuration was selected. This value is calculated by sorting the configurations by IPC for each phase and identifying which entry was selected by the predictor. For example, a value of 1 indicates that the best configuration was selected and 2 indicates that the second best configuration was selected, etc. This graph shows that in 64% of cases the single best configuration is identified by the predictor. An additional 19% of phases have the second best possible configuration selected.

As a result of the high configuration prediction accuracy, the performance loss in mispredicted regions is usually quite low. The rightmost chart of Figure 3 shows the weighted performance loss observed for each benchmark during mispredicted phases. This value is calculated as $\sum_{i=1}^{N_B} w_i \cdot D_i$, 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 specific region accounts for, and D_i is the absolute performance penalty suffered by the

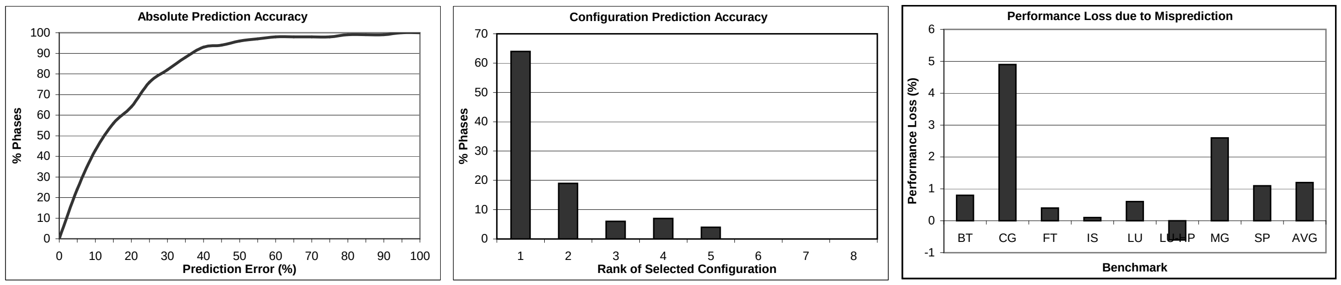

Fig. 3. The left chart illustrates the CDF of prediction error. The middle chart illustrates the percent of phases for which each rank of configuration was selected. The rank of the selected configuration is taken from the list of configurations sorted by their IPCs on static executions of each phase, a value of 1 indicates that the optimal configuration was selected. The right chart shows the performance loss (>0) or gain (<0) resulting from configuration misprediction.

mispredicted region i. The average penalty across benchmarks is only 1.2%. The explanation for the negative performance loss (performance gain) of LU-HP is that by not changing configurations to the optimal in all cases, the cache effects of altering configurations are reduced. These results show that our model is capable of identifying the optimal configuration most of the time, and when it does not it still manages to find a competitive configuration to use, with minimal performance penalty. Note that in some cases, a misprediction may derive a more energy-efficient hardware configuration for a phase, where power may be reduced with negligible impact on execution time.

C. Adaptive Concurrency Control Evaluation

To measure the power consumption of the benchmarks under various hardware configurations we utilize a power measurement methodology based on hardware event counters [17] that has proven to be highly accurate. This methodology works by first partitioning the processor into components and then determining the maximum power consumption of each component based on the die area it consumes. The runtime power consumption of each component is the maximum power adjusted by an activity factor. The latter is estimated by looking at corresponding hardware event counters. This amount is added to a non-gated clock power associated with each component that grows non-linearly with activity. Finally, the power consumptions of all components are summed along with a constant base idle power. It should be noted that we focus only on processor power consumption. For the well-tuned scientific applications we consider in this paper, processor power is the dominant portion of the total system power consumption [33].

1) Motivating Examples: Figure 4 depicts the execution times and energy consumption of each benchmark under class size A for each static configuration. Static configurations use a single configuration for the entire execution. These graphs show that on our experimental platform, very little additional performance gain is seen through adding additional processors once two processors are active. Particularly interesting is the IS benchmark, which sees its best performance using a single thread on only one processor. Further, sometimes there is a large gain through using the second execution context on each processor, and sometimes a substantial loss. For these reasons, adaptation of the number of processors and execution contexts stands to improve both execution time and power consumption. It can be observed that while performance levels out, the energy consumption increases at rather steep rates with more processors.

The reader may note that the observed scalability bottlenecks are an artifact of hardware bottlenecks, such as limited memory bandwidth. While this statement is correct, it also reflects a property of a large number of real systems, including state-of-the-art platforms that outdate our experimental system. For example, we performed experiments with the NAS benchmarks on a newly released quad-core Intel Xeion processor (QX 6700) which have shown that applications still tend not to scale well on even the latest hardware. In particular, several of the benchmarks fail to scale beyond two cores, with maximum speedups saturating well below 2 (see Figure 5). As a result, opportunities for concurrency throttling still exist even in the newest hardware platforms.

As further evidence of the importance of phase-level adaptation, Figure 6 displays the IPCs for each phase of the LU-HP benchmark at class size B under each static configuration normalized by the IPC of (1,1). It is evident from the chart that a single application can have optimal configurations varying greatly between phases. LU-HP in particular experiences five different optimal configurations across different phases, specifically (2,1), (2,2), (3,2), (4,1), and (4,2). Therefore, using a technique to execute each phase at its local optimal operating point stands to improve performance. In cases where the optimal configuration occurs on fewer than the available number of processing elements, power savings can occur during the execution of these phases. The goal of our adaptation approach is to exploit these properties with no *a priori* knowledge of the codes and achieve both power and performance benefits.

Before discussing the online adaptive strategies and their results, we focus on two offline approaches to adaptation. The first of these, *static optimal*, uses the single program-wide static configuration that results in the lowest execution time. The static optimal configuration for an entire program differs in general from the static optimal configurations of phases in a program. The second approach is *phase optimal* and uses the local static optimal configuration, not considering cross-phase effects, as defined earlier. Due to interference occurring by changing the configurations in *phase optimal*, the mean execution time of the benchmarks is 1.0% higher than *static optimal*. For this reason, we limit our following evaluation to comparing adaptive strategies to *static optimal*.

The two offline approaches that we consider have the disadvantage that the optimal configuration may change with different input sizes. For example, IS executes statically optimally on (3,1) for class size W, but (1,1) and (2,1) for class sizes A and B respectively. For individual phases, the optimal configuration

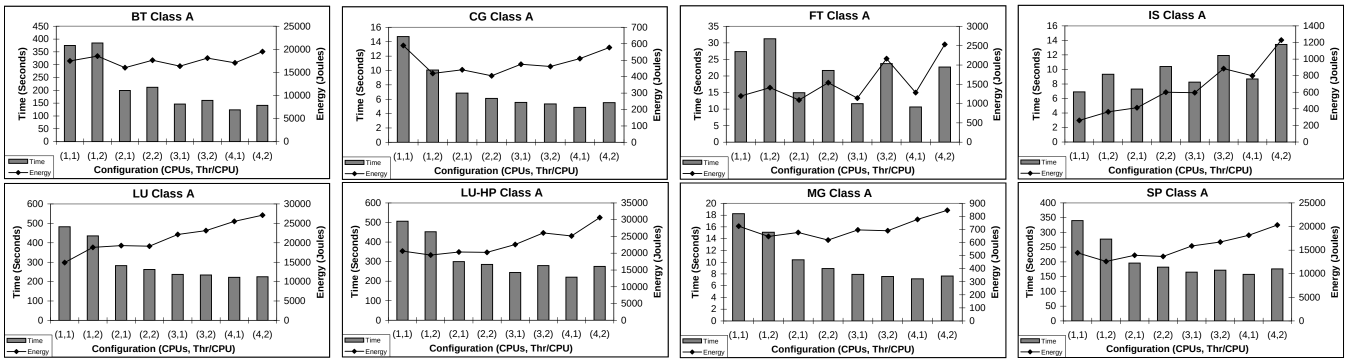

Fig. 4. The execution times and energy consumption of each static configuration.

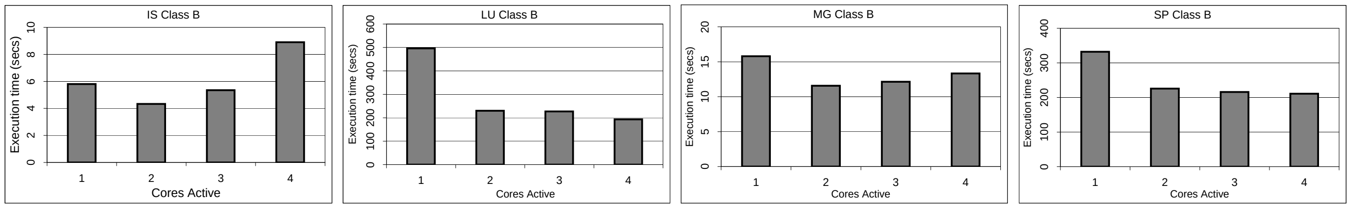

Fig. 5. Scalability characteristics of 4 of the NAS benchmarks on a state-of-the-art quad-core Intel processor.

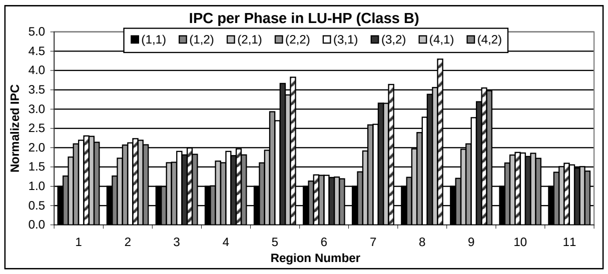

Fig. 6. IPCs for each phase of the LU-HP benchmark under each static configuration, normalized by the IPC on (1,1).

varies by problem size as well. Specifically, only 52.5% of the program phases in our benchmarks experience the same optimal configuration regardless of input size. This means that use of these static techniques requires offline analysis that is specific to the application and the input size. By contrast, the online adaptive approaches adapt autonomically at runtime for the current application execution and require no application/input-size specific offline analysis.

2) Empirical Search-based Strategies: For purposes of comparison, we have implemented two alternative dynamic adaptation strategies based on empirical search of the configuration space at runtime. The first of these is the most straightforward form of adaptation, exhaustive search, where each possible configuration is tested and the one that provides the lowest execution time is selected for each phase. Figure 7 illustrates the normalized arithmetic means of three metrics: execution time, average power consumption during execution, and energy consumption. These metrics are derived for each benchmark under different execution strategies. Each metric is first normalized to the corresponding metric of the (4,2) configuration for the specific benchmark, which exploits all available execution contexts on our experimental platform. We then calculate the arithmetic means of the normalized metrics, for each benchmark.

Occasionally, the power consumption actually *increases* through the use of adaptation. This result is counterintuitive since adaptation is always expected to either keep the number of processors and Hyperthreads used constant, or reduce it. Recall that our starting assumption for adaptation is that deactivating threads inside a processor reduces power. This is actually true in the majority of phases, specifically 79% of all phases. However, in certain cases, the use of Hyperthreading introduces long stall times in the processor, due to contention for shared resources, and therefore long periods of processor inactivity, during which power consumption is low. By contrast, deactivating Hyperthreading in these situations, reduces stall times, increases processor utilization and therefore increases the average power consumption. However, the increase in power consumption does not translate into an increase in overall energy, due to the reduction in execution time experienced in these cases. Therefore, the overall result is positive for the applications where the anomaly with Hyperthreading occurs.

The average execution time of all benchmarks over all problem sizes using exhaustive search was reduced by 10.9% compared to statically using all available processors and execution contexts on the system. Power is reduced by 9.7% as well, resulting in a 19.5% reduction in total energy consumption. However, this approach incurs high overhead in the exploration phase, due to its testing of each configuration. Exhaustive search needs to execute 8 iterations of each phase to reach a decision. This overhead shows up when the results are compared to using the optimal static number of threads for the entire program execution, where exhaustive search is outperformed by 16.1% overall and by 31.6% in benchmarks with a small number of iterations (MG, CG, FT, IS). However, for applications with many iterations (BT, SP, LU, LU-HP), exhaustive search is able to come within 1.1% of the static optimal in terms of performance, while reducing power consumption by 3.3%, because the search overhead can be amortized over a large number of iterations.

The second empirical search technique that we implemented is

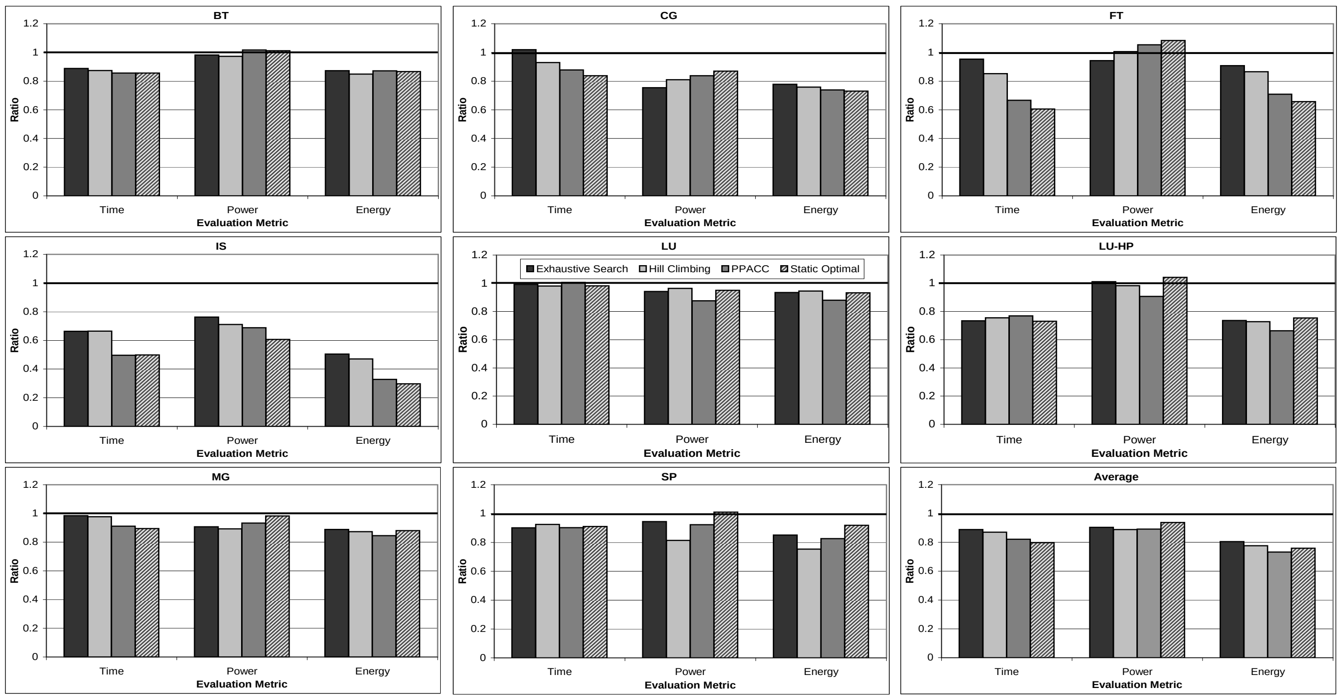

Fig. 7. Performance of the adaptation strategies in terms of execution time (first group of bars), power (second group of bars), and energy (third group of bars) normalized with respect to the (4,2) static configuration for each benchmark, averaged over all class sizes.

a heuristic search algorithm, which we have previously devised to reduce the overhead of exhaustive search [6]. This algorithm works by applying a hill-climbing heuristic search to find the optimal number of processing elements to use at each dimension of parallelism, one dimension at a time. The algorithm begins by executing the phase on all available processors with all Hyperthreads active. Then, the number of processors is successively reduced until an increase in execution time is observed. The lowest number of processors that results in a decrease in execution time is used for the corresponding phase. This process is then repeated on the decided upon number of processors to determine the number of Hyperthreads to use on each processor.

Using hill climbing reduces the number of required test iterations for each phase to 5 in the worst case for our experimental platform, and only 3 in the best case, since our platform has two layers of parallelism. This overhead reduction allows the hill climbing algorithm to achieve improved performance compared to exhaustive search because a larger percentage of the iterations will be executed with the decided upon optimal configuration, rather than testing additional suboptimal configurations. Specifically, compared to exhaustive search, hill climbing achieves an 1.6% improvement in execution time overall and a 3.9% improvement for applications with few iterations, with a minor 0.5% increase in execution time for the applications with many iterations. The slight performance drop in applications with many iterations can be attributed to occasionally, but infrequently, selecting slightly worse configurations than exhaustive search. Power consumption is reduced by 1.7% and energy consumption by 3.6% on average, compared to exhaustive search. Compared to *static optimal*, hill climbing reduces the performance loss to 26.5% for applications with a small number of iterations, and to 13.9% overall. The energy consumption is also reduced by 22.4% compared to using all available execution contexts. These results show that hill climbing is able to reach good configuration decisions, while requiring fewer exploration iterations, thus introducing lower overhead. However, the search overhead is still clearly a factor for applications with few iterations.

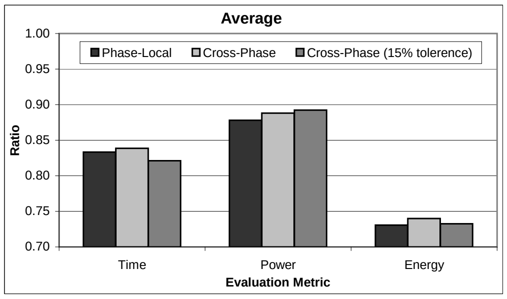

Fig. 8. Execution time, power, and energy effects of utilizing the three different prediction based adaptation strategies, with all numbers normalized with respect to the (4,2) execution.

3) Performance Prediction-based Adaptation: Through the use of performance prediction, the number of iterations required for adaptation can be further reduced using the algorithm presented in Section IV, to only two iterations in the case of our experimental platform, thereby minimizing overhead. Further, performance prediction reduces the effects due to changing configurations during the exploration process that can lead to suboptimal decisions by the direct-search strategies. On the downside, $uIPC$ predictions need significantly more processor cycles than direct comparisons of the execution times of phases.

We first compare a strategy whereby the predicted optimal configuration for each phase is used blindly, to strategies that consider cross-phase analysis to make decisions. The best strategy is selected for use with PPACC, and is compared to the offline and direct-search approaches already presented. First, we evaluate our approaches to minimize the harming effects of using the local optimal configuration for each phase, which occur if changes in the configuration of adjacent phases result in redistribution of working sets between caches [22]. We compare the results to the greedy local optimal approach to find the best prediction-based adaptation approach. Our experimental results, shown in Figure 8, indicate that simply attempting to avoid cache interference is not inherently effective. Using an approach whereby the configuration selected as the best for the majority of execution time (i.e. the dominant configuration) is enforced for all phases produces a slowdown of 1.5% compared to the local optimal approach, with an additional 0.9% energy consumption. This happens because, in many cases, the benefits of executing a phase with its local optimal configuration outweigh the performance loss suffered as a result of cross-phase interference.

Given the advantage of local adaptation over global enforcement of the dominant configuration, along with the fact that changing configurations too liberally hurts performance, we developed an intermediate adaptation scheme that uses a global dominant policy for most phases, with the exception of phases expected to experience substantial performance gains by using its own local optimal configuration. In particular, using this approach, the global decision is enforced unless a given phase expects at least a 15% performance gain, which we experimentally verified to be enough to outweigh the cache effects of changing configurations. When compared to phase-local adaptation, cross-phase decision making with exceptions attains an 1.3% average performance improvement. An increase in power consumption of 2% is also observed, however the energy consumption is unchanged, making cross-phase with exceptions the best prediction-based adaptation strategy. These results show that concurrency throttling modules must consider the effects of changing configurations in adjacent phases in conjunction with the local predictions for each phase, when making decisions.

Using cross-phase decisions while allowing exceptions, results in an average 17.9% performance improvement over statically using all available execution contexts, further improving performance upon exhaustive search by 8.3% and upon hill climbing by 6.8%. Additionally, the average performance loss compared to the statically optimal configuration is reduced to only 2.5% overall and 1.3% for applications with many iterations, showing that a flexible cross-phase decision policy is able to make performance-effective decisions. More importantly, the results for applications with a small number of iterations are within 3.7% of the statically optimal configuration, compared to 31.6% and 26.5% for exhaustive search and hill climbing respectively, because of the significantly reduced exploration overhead. Our experimental platform has only 8 feasible hardware configurations and the performance advantage of PPACC over the empirical search approaches is expected to grow in the future as the available number of processors, cores, and threads in a system rises.

The power-related results for PPACC are just as substantial as those for performance. Energy consumption is the product of power consumption and execution time, and concurrency control attempts to reduce both, decreasing energy consumption by a still larger margin. We observe 10.8% and 26.7% reductions in power and energy consumption, respectively, compared to using all execution contexts. When compared to using the static optimal configuration, a 2.9% average reduction in power is seen and a 0.9% reduction in energy. This result may seem surprising, however it can be explained by the fact that the static optimal uses only a single configuration for the entire program execution, rather than further decreasing the number of active processors for individual phases below the global optimal level.

PPACC also sees a 1.1% reduction and a 0.8% increase in power consumption compared to exhaustive search and hill climbing respectively. Further tracing of this result shows that PPACC executes the benchmarks with an average of 3.13 processors, while exhaustive search executes with 3.20 and hill-climbing with 3.02 processors. However, PPACC reduces total energy consumption by 10.2% and 6.3% because of its performance advantages. These results indicate that prediction-based adaptation is able to make effective decisions, both in terms of improving execution time and in terms of reducing energy consumption.

Overall, prediction-based adaptation outperforms or matches the performance of direct-search based adaptation on all fronts. Additionally, it does not require the application/input-size specific offline analysis, while still achieving results very close to static optimal for performance and surprisingly, but justifiably, better results for power and energy. Performance prediction-based adaptation as utilized in PPACC thus proves to be a highly effective strategy for improving the performance and energy consumption of parallel applications.

VI. Conclusions

The performance and power characteristics of applications running on emerging computing systems composed of multiple multithreaded and multicore processors demand consideration of throttling concurrency when scalability bottlenecks result in no performance gain, or a performance loss, from using additional processors. In this paper, we have presented an approach to adaptive concurrency control that uses information collected at runtime to predict the performance of an application across various hardware configurations. Hardware event counters are collected to provide insight into the interaction of the hardware and software, allowing the predictor to characterize the performance and scalability of a given program phase. Over a range of benchmarks with a variety of execution characteristics, the accuracy of the predictor in terms of locating the optimal configuration to execute benchmarks, on a per-phase basis, was shown to be high.

We have presented an autonomic runtime system that employs the described performance predictor and have shown that adaptive concurrency control can be performed with performance- and energy-effective decisions being made, while keeping the overhead at manageable levels. We improve upon an approach that selects the optimal predicted configuration for each phase by making the predictor aware of the decisions made for other phases, thereby allowing it to consider cross-phase cache effects in the decisions for each phase, resulting in improved performance. The described system is shown to outperform adaptation strategies based on empirical searches of the configuration space due to reduced exploration overhead and a decision process that is not mislead by effects resulting from changes in configurations during the training process, as are the search strategies. Finally, the approach is shown to be significantly more effective than simply using all available execution contexts for all phases, in terms of performance, power, and energy consumption. It yields performance results comparable to offline-derived application/input-size specific decisions, and improvements in power and energy, without requiring additional application/input-size specific analysis.

Acknowledgment

This research is supported by the National Science Foundation (Grant CCR-0346867), the U.S. Department of Energy (Grant DE-FG02-06ER25751) and Virginia Tech.

References

[1] T. Anderson, B. Bershad, E. Lazowska, and H. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[2] U. Andersoon and P. Mucci. Analysis and Optimization of Yee Bench using Hardware Performance Counters. In *Proc. of the ParCo 2005 Conference*, Malaga, Spain, September 2005.

[3] Shekhar Y. Borkar. Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. *IEEE Micro*, 25(6):10–16, September 2005.

[4] C. Cascaval, E. Duesterwald, P. Sweeney, and R. Wisniewski. Multiple Page Size Modeling and Optimization. In *Proc. of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 339–349, Saint Louis, MO, September 2005.

[5] M. Curtis-Maury, J. Dzierwa, C. Antonopoulos, and D. Nikolopoulos. Online Power-Performance Adaptation of Multithreaded Programs using Hardware Event-Based Prediction. In *Proc. of the 20th ACM International Conference on Supercomputing*, Queensland, Australia, June 2006.

[6] 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.

[7] 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*, Monterrey, Mexico, January 2001.

[8] L. Eeckhout, S. Nussbaum, J. Smith, and K. De Bosschere. Statistical Simulation: Adding Efficiency to the Computer Designer's Toolbox. *IEEE Micro*, 23(5):26–38, September 2003.

[9] Krste Asanovic et at. The landscape of parallel computing research: A view from berkeley. Technical report ucb/eecs-2006-183, EECS Department, University of California at Berkeley, December 2006.

[10] N. Adiga 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, November 2002.

[11] W. Feng and C. Hsu. The Origin and Evolution of Green Destiny. In *Proc. of IEEE Cool Chips VII: An International Symposium on Low Power and High Speed Chips*, Yokohama, Japan, April 2004.

[12] V. Freeh, D. Lowenthal, F. Pan, and N. Kappiah. Using Multiple Energy Gears in MPI Programs on a Power-Scalable Cluster. In *Proceedings of the 2005 ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP'05)*, June 2005.

[13] 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, April 2005.

[14] 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.

[15] M. Hall and M. Martonosi. Adaptive Parallelism in Compiler-Parallelized Code. Stanford, California, August 1997.

[16] E. Ipek, S. McKee, B. de Supinski, M. Schulz, and R. Caruana. Efficiently Exploring Architectural Design Spaces via Predictive Modeling. In *Proc. of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, June 2006.

[17] 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*, San Diego, CA, November 2003.

[18] 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.

[19] P. Joseph, K. Vaswani, and M. Thazhuthaveetil. Efficiently Exploring Architectural Design Spaces via Predictive Modeling. In *Proc. of the 39th International Symposium on Microarchitecture*, December 2006.

[20] 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*, Chicago, IL, June 2005.

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

[22] 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, March 2003.

[23] 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, November 2005.

[24] S. Kirkpatrick, C. Gelatt, and M. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, 1983.

[25] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE MICRO*, 25(2):21–29, March/April 2005.

[26] B. Lee and D. Brooks. Accurate and Efficient Regression Modelling for Microarchitectural Performance and Power Prediction. In *Proc. of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, June 2006.

[27] 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, March 2005.

[28] 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.

[29] 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 Symposim*, Denver, CO, April 2005.

[30] J. Lu, H. Chen, P. Yew, and W. Hsu. Design and Implementation of a Lightweight Dynamic Optimization System. *The Journal of Instruction-Level Parallelism*, 6:1–24, 2004.

[31] J. Marathe and F. Mueller. Hardwware Profile-Guided Automatic Page Placement for ccNUMA Systems. In *Proc. of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 90–99, New York, NY, March 2006.

[32] A. Settle, J. Kihm, A. Janiszewski, and D. Connors. Architectural Support for Enhanced SMT Job Scheduling. In *Proc. of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT'04)*, pages 63–73, Antibes, France, September 2004.

[33] S. Sharma, C. Hsu, and W. Feng. Making a Case for a Green500 List. In *Proc. of the Workshop on High-Performance, Power-Aware Computing*, Rhodes, Greece, April 2006.

[34] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. In *Proc. of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[35] A. Tucker and A. Gupta. Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors. In *Proc. of the 12th ACM Symposium on Operating Systems Principles (SOSP'89)*, pages 159–166, Litchfield Park, Arizona, December 1989.

[36] M. Voss and R. Eigenmann. Reducing Parallel Overheads through Dynamic Serialization. In *Proc. of the 13th International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (IPPS/SPDP)*, pages 88–92, San Juan, Puerto Rico, April 1999.

[37] L. Yang, X. Ma, and F. Mueller. Cross-Platform Performance Prediction of Parallel Applications using Partial Execution. In *Proc. of the IEEE/ACM Supercomputing'2005: High Performance Networking and Computing Conference*, Seattle, WA, November 2005.

[38] K. Yue and D. Lilja. An Effective Processor Allocation Strategy for Multiprogrammed Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 8(12):1246–1258, December 1997.

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