Prediction Models for Multi-dimensional Power-Performance Optimization on Many Cores

Ankur Savailal Shah

Thesis submitted to the faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science and Applications

Committee Members:
Dimitrios S. Nikolopoulos, Chair
Kirk W. Cameron
Wu-chun Feng

April 18, 2008
Blacksburg, Virginia

Keywords: Multicore processors, high-performance computing, performance prediction, runtime adaptation, concurrency throttling, power-aware computing

Prediction Models for Multi-dimensional Power-Performance Optimization on Many Cores

Ankur Savailal Shah

(ABSTRACT)

Power has become a primary concern for HPC systems. Dynamic voltage and frequency scaling (DVFS) and dynamic concurrency throttling (DCT) are two software tools (or *knobs*) for reducing the dynamic power consumption of HPC systems. To date, few works have considered the synergistic integration of DVFS and DCT in performance-constrained systems, and, to the best of our knowledge, no prior research has developed application-aware simultaneous DVFS and DCT controllers in real systems and parallel programming frameworks. We present a multi-dimensional, online performance prediction framework, which we deploy to address the problem of simultaneous runtime optimization of DVFS, DCT, and thread placement on multi-core systems. We present results from an implementation of the prediction framework in a runtime system linked to the Intel OpenMP runtime environment and running on a real dual-processor quad-core system as well as a dual-processor dual-core system. We show that the prediction framework derives near-optimal settings of the three power-aware program adaptation knobs that we consider. Our overall runtime optimization framework achieves significant reductions in energy (12.27% mean) and ED^2 (29.6% mean), through *simultaneous* power savings (3.9% mean) and performance improvements (10.3% mean). Our prediction and adaptation framework outperforms earlier solutions that adapt only DVFS *or* DCT, as well as one that sequentially applies DCT *then* DVFS.

Further, our results indicate that prediction-based schemes for runtime adaptation compare favorably and typically improve upon heuristic search-based approaches in both performance and energy savings.

ACKNOWLEDGEMENTS

During my two years at Virginia Tech I have had the privilege to be a part of an academic community that is truly exceptional. While officially this is an individual effort, the work done for this project could in no way have been accomplished without the help of many people in this community. I would like to take this opportunity to thank all of them.

First and foremost, I am grateful to my family for believing in me and supporting me as I embarked on the path towards graduate studies.

I wish to express sincere gratitude to my advisor, Dr. Dimitrios S. Nikolopoulos for guiding me throughout the thesis. I thank him for giving me the opportunity to work on a wonderful thesis topic. The knowledge I gained from him in these two years will immensely help me in my career.

I also want to thank my committee members Dr. Kirk Cameron and Dr. Wu-chun Feng for their valuable comments on this thesis as well as for their encouragement and ideas during the projects I worked on for the *Computer Architecture* and *Advanced Networking* courses.

I would like to thank all the members of the PEARL research group, especially Mathew Curtis-Maury who provided me with outstanding ideas and assistance which drove the direction of this project. I am also thankful to Scott Schneider for giving me an opportunity to work on *streamflow*, which was definitely one of the most interesting things I did as a graduate student.

I would also like thank Dr. Ali Butt, for introducing me to research in computer science and making *PeerStripe* a fun project to work on.

And lastly, I would like to thank my friends Pavan, Ganesh, Hari, Beran, Chreston, Sanket, Jyotirmay, Harshil and Jai who have all been there for me through thick and thin while I worked towards this Masters Degree.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Multi-core processors trade parallelism for reduced power consumption through software control of the number of active cores and power-aware workload distribution between cores [29]. In an execution workload containing phases with limited scalability to many cores, controlling concurrency and workload distribution produces substantial energy savings with no performance penalty. Occasionally, throttling concurrency provides a performance gain by reducing contention between threads for shared resources such as memory bandwidth. Conserving cores at runtime is valuable for emerging many-core processors, which will integrate hundreds of cores [1]. Recent studies indicate that less than half of industrial-strength codes scale to hundreds of (conventional) processors and only a handful scale to thousands of processors [22]. In practice, most parallel codes in use today run on one to eight processors. Thus, conserving cores, either for power saving purposes, or for other purposes, such as consolidation or fault tolerance [26], are viable alternatives to uncontrolled parallelization.

Dynamic concurrency throttling (DCT) is a software-controlled mechanism, or *knob*, for runtime power-performance adaptation on systems with multi-core processors. Dynamic voltage and frequency scaling (DVFS) provides a second knob. While earlier research has significantly ad-

vanced the understanding of performance and power implications of DVFS [19, 41] and processor/core conservation [7], less emphasis has been placed on integrating DVFS and DCT in a *unified* software power-performance adaptation framework. In particular, these techniques have not yet been combined in the context of HPC systems and applications. Further, combined DVFS and DCT frameworks have not been evaluated on real software systems for multi-core hardware.

To the best of our knowledge, combined approaches for simultaneous DVFS and DCT have only been explored via hardware simulation, using empirical search methods [29]. These methods clearly demonstrated that runtime adaptation of concurrency and voltage/frequency in iterative parallel applications can achieve high-performance, power efficient execution. However, even with only two power-performance adaptation knobs available in software, the search space for adaptation can grow to unmanageable proportions. An M-core processor with L voltage/frequency levels presents a power-performance adaptation search space of size $O(L \cdot M)$. If we assume that the processor's cores are asymmetric, implying that performance depends on the placement of threads on cores as well as the number of cores used, the space grows to $O(L \cdot 2^M)$. We cannot reasonably search this space, even with a modest number of cores and power states, particularly at runtime.

1.2 Contributions

While prediction models for adaptation via concurrency throttling were introduced by Curtis-Maury, et al [8, 9], the current work makes several new contributions in the context of performance prediction for power-performance adaptation.

- *We present a software framework for multi-dimensional, software-controlled, and HPC-constrained power-performance adaptation on systems based on multi-core processors.*
- *We consider prediction models for DVFS and DCT simultaneously, effectively exploring a larger and significantly more challenging runtime optimization space.*

- *We draw comparisons between alternative power-performance adaptation methods and present effective strategies to synthesize multiple power-performance adaptation methods in software.*

- *We propose methods to generalize multi-dimensional prediction models using sampling of the target configuration space and we significantly improve prediction accuracy compared to previous work [8, 9], thus achieving better optimization with zero tolerance for performance loss.*

- *We implement a fine-grained and directed technique (phase mapper) to the application of the control knobs significantly reducing the adaptation overhead.*

- *We evaluate the framework against scientific application benchmarks, significantly improving the performance and energy consumption.*

The framework adds a new dimension to the ACTOR framework [8, 9] by integrating the effects of DVFS into the training and runtime models. The framework provides transparent runtime adaptation of HPC codes and requires only a trivial code instrumentation step. Its key component, a dynamic multi-dimensional performance predictor, statistically analyzes samples of hardware event rates collected from performance monitors and predicts the performance impact of switching to any DCT and DVFS levels available on the system (either combined or in isolation). We base the statistical analysis on a rigorous regression model that is trained from samples of the power-performance adaptation search space collected from real workloads. We apply the model to application execution phases individually since its low cost allows its use at runtime during a single run of each application, regardless of input. The model usually predicts the optimal system configuration to execute each phase (occasional small errors lead it to choose a near optimal configuration instead), thereby achieving performance gains and energy savings. We present a functioning software prototype of our framework for OpenMP applications and evaluate it through physical experimentation on a system with two quad-core Intel Xeon processors and a system with two dual-core Intel Xeon processors. The model derivation and training are automated and portable

across multi-core processors. Further, the associated software prototype is based on portable components, specifically PAPI and OpenMP.

Through derivation and evaluation of our prototype, this paper contributes answers to several important questions:

- Can statistical performance models based on hardware event counters accurately predict performance with multi-dimensional input parameters—more specifically, the number of cores, mapping of threads to cores, and processor voltage/frequency—across a large space of untested system configurations?
- Can prediction-based models achieve as good or better results than heuristic search methods that time the phases of the program on sample configurations for multi-dimensional power-performance adaptation?
- Do prediction-based model costs prevent their use for online power-performance adaptation, given multiple knobs?
- Can we prune the optimization space for prediction-based power-performance adaptation during model training and during model actuation to derive effective adaptation frameworks without prohibitive development and training costs?
- Which power-performance adaptation knob—DCT or DVFS—is more critical with respect to power-efficiency, assuming no tolerance for performance loss in an HPC environment?
- What are the synergistic effects of applying these knobs simultaneously, if any?

Our experimental results demonstrate that simultaneous phase-aware prediction of the performance impact of DVFS and DCT can achieve significant energy savings (12.27% mean for the NAS benchmarks). In nearly all applications we tested, the energy gains come with simultaneous power savings (3.9% mean) and performance improvement (10.3% mean). Even on machines where there aren't many opportunities for reducing concurrency to improve performance, our framework can use frequency scaling to obtain reductions in energy consumption by upto 15.3% while sustaining a marginal performance loss of 4.9%. Since our prediction schemes converge

rapidly to optimal or near-optimal system configurations for parallel execution phases, our results show they typically outperform exhaustive or heuristic search strategies. Prediction-based schemes scale better than search-based schemes when applied to runtime adaptation, which renders them more suitable for emerging many-core systems. We further show experimentally that a $Hybrid$ 2-dimensional DVFS-DCT predictor achieves both higher energy savings and performance gain, compared to a predictor that first predicts DCT and then DVFS, or either in isolation. Thus, users in performance-sensitive settings should apply the unified model of DVFS and DCT to sustain high performance and simultaneously obtain near maximum energy savings. Last, our results show that prediction-based power-performance adaptation schemes come very close to optimal static execution schemes, which can be derived only post-facto, after exhaustive experimentation.

1.3 Outline

The rest of this thesis document is organized as follows. Section 2 presents background and preliminary concepts. Section 3 presents our model of multi-dimensional power-performance prediction. Section 4 outlines the implementation of our prediction model in a real software prototype. Section 5 presents our experimental analysis from a system with two quad-core Intel Xeon processors and a system with two dual-core Intel Xeon processors. Section 6 discusses related work and Section 7 concludes the paper.

Chapter 2

Preliminaries

In this section, we discuss the definitions of terms used in the rest of the paper and provide the theoretical foundation for our performance prediction models. Since different mappings of a set of threads to cores may yield significant performance variation, we differentiate the number and the topology of the cores on each die, as well as the number and topology of the dies during performance prediction. For example, on a system with multiple Intel Xeon quad-core processors, where each processor has two sockets and each socket has two cores and a L2 cache bank which is accessible by both cores on the socket, we differentiate between three potential mappings of threads to cores: i) two threads running in the same socket and sharing a common L2 cache bank; ii) two threads placed on different sockets on the same die and executing with private L2 cache banks, using only half of the available memory bandwidth due to the use of a single die; and iii) two threads placed on different dies, without sharing common L2 cache space, with full memory bandwidth. Programs that use two threads on this processor have a choice between binding the threads in any of the three possible ways listed above. Depending on program properties the performance of these configurations may vary significantly. We use the term *concurrency configuration* to imply both a certain degree of software concurrency (number of concurrently executing threads among which computation is distributed) used in the program, and a certain mapping of threads to cores and dies.

We assume that each die of the system can independently be set by a privileged instruction

to execute at a voltage/frequency level chosen from a predetermined set defined by the architecture. We assume global voltage and frequency scaling for each die as a whole, as opposed to per core, since this technology is readily available on commercially available multi-core processors. We conduct physical experimentation, using hardware timers and power meters to measure performance and energy respectively.

We decompose parallel workloads into phases, where each phase executes parallel computation using a potentially variable number of threads and completes at a synchronization point, such as a barrier, or a critical section. While DVFS is entirely transparent and can be applied to any code region, we cannot apply DCT to arbitrary parallel code regions without violating correctness. In principle, codes written in a shared-memory model where parallel computation does not include code dependent on the identifiers of threads, are amenable to DCT without correctness considerations. The vast majority of OpenMP codes meet this requirement for processor-independence, as do the workloads that we use in this paper (NAS benchmarks). Ongoing research efforts are addressing DCT in other programming models, such as MPI [17].

Though not universally applicable, concurrency throttling has the advantage that it may actually improve performance while simultaneously reducing power consumption [9]. While concurrency is throttled, selected cores per die, or entire dies of the MCMP are put to a low-power idle state (such as the halt state on Intel processors), either by the runtime library or by the operating system. Halting cores and dies aggressively reduces power consumption, while also reducing contention for shared resources such as cache and memory bandwidth during memory-intensive phases or at synchronization points.

Our contribution involves a modeling/prediction component and a runtime actuation component. The first component predicts performance for each phase of parallel code under all feasible concurrency configurations and global voltage/frequency settings, with input from samples of hardware event counters collected at runtime. We use it at the boundaries of execution phases as the program executes. The predictor correlates hardware event counter samples, concurrency configurations (number and mapping of threads to cores), and voltage/frequency settings with whole

system instruction throughput. Without loss of generality, we derive predictions for the fixed optimality criterion of minimizing energy without increasing runtime, which best meets the requirements of HPC environments. We use our predictions in the actuation component on a per phase basis to minimize energy consumption under this rigid performance constraint. Since our predictions can be inaccurate, actuations may actually incur performance and energy loss. In practice, such losses are usually imperceptible, since the predictor often derives suboptimal DCT and DVFS settings that exhibit performance and power signatures which are very similar to those of the optimal settings.

Chapter 3

Empirical Model Derivation

In this section, we discuss the runtime performance predictors that estimate performance in response to changing the settings of two power-performance *knobs*, DCT and DVFS. Each combination of frequency and concurrency configuration (threads and specific core bindings) available on the system is referred to as a hardware configuration, The predictors use input from execution samples collected at runtime on specific configurations to predict the performance on other, untested configurations. Performance is estimated for each phase in terms of useful instructions per second, or $uIPC$, which is the IPC without the instructions used for parallelization or synchronization. The number of $uIPC$ is an accurate indicator of how a particular application interacts with the not just the processor but the entire machine including the memory hierarchy. By using $uIPC$ predictions, we exploit opportunities to save power primarily by scaling down the memory-bound parts of the actual computation to reduce contention and to exploit slack due to memory or parallelization stalls. The input from the sample configurations consists of the useful IPC ($uIPC_s$), as well as a set of n hardware event rates ($e_{(1..n,s)}$) observed for the particular phase on the sample configuration s, where each event rate $e_{(i,s)}$ is calculated as the number of occurrences of event i divided by the number of elapsed cycles during the execution of configuration s. The model predicts $uIPC$ on a given target configuration t, which we call $uIPC_t$.

3.1 Baseline Prediction Model

Our prediction model estimates the effects of the observed event rates and $uIPC_s$ that produce the resulting value of $uIPC_t$. The event rates capture the utilization of particular hardware resources that represent scalability bottlenecks, thereby providing insight into the likely impact of hardware utilization and contention on scalability. Although the model can include multiple sample configurations, we begin by describing the simplest case of a single sample and build up the model from there. We model $uIPC_t$ scalability as a linear function as follows:

$$uIPC_t = uIPC_s \cdot \alpha_t(e_{(1..n,s)}) + \epsilon_t \tag{3.1}$$

From Equation 3.1, it can be seen that both the function α_t and the constant term ϵ_t are dependent upon the particular target configuration. That is, each target configuration t is represented through different coefficients that capture the varying effects of hardware utilization at different degrees of concurrency, different mappings of threads to cores, or different voltage/frequency levels. In effect, $\alpha()$ scales up or down the observed $uIPC_s$ on the sample configuration based on the observed values of the event rates on the same configuration, to attain $uIPC_t$ on any of the target configurations. The observed event rates determine how much we scale $uIPC_s$ as a linear combination of the sample configuration event rates as depicted in Equation 3.2:

$$\alpha_t(e_{(1..n,s)}) = \sum_{i=1}^{n}(x_{(t,i)} \cdot e_{(i,s)} + y_{(t,i)}) + z_t \tag{3.2}$$

The model's intuition is that changes in event rates indicate varying resource utilization and contention, resulting in either positive or negative effects on $uIPC_s$, which can be represented in the model through positive or negative coefficients. While the relationship between event rates and $uIPC$ may not be strictly linear, it has been observed that it can be well represented using a linear model [9, 24, 32]. estimate the specific coefficients through multivariate linear regression as dis-

cussed further in Section 3.3. By using such an empirical model, we greatly simplify the retraining process required for new architectures since we automatically infer the model from a set of training samples rather than through a detailed architectural description. We combine Equations 3.1 and 3.2, to derive the following equation for $uIPC$ on a particular target configuration t using a single sample configuration:

$$uIPC_t = uIPC_s \cdot \sum_{i=1}^{n}(x_{(i,t)} \cdot e_{(i,s)}) + uIPC_s \cdot \gamma_t + \epsilon_t \tag{3.3}$$

Therefore, estimating the value of $uIPC_t$ is equivalent to the proper approximation of the coefficients $x_{(i,t)}$, the constant term ϵ_t, and γ_t. The γ_t variable is the sum of a collection of terms from α_t that represent a coefficient for $uIPC_s$ itself, independent of the values of $(e_{(1..n,s)})$, and defined as $\sum_{i=1}^{n}(y_{(t,i)}) + z_t$.

3.2 Model Extensions

While the baseline prediction model can be effective for DCT [8, 9], we refine it to improve model accuracy and extend the model to predict performance with multi-dimensional input. Our first extension models $uIPC_t$ as a linear combination of multiple sample configurations from the configuration space. In the context of DVFS and DCT, each sample configuration uses a different number of threads bound to different execution units in the machine, at potentially different voltage and frequency levels. Thus, each sample configuration provides some additional insight into execution on other, untested configurations. The use of multiple samples allows the model to "learn" more about each program phase's execution properties that determine performance on alternative configurations. The actual selection of the samples can be statistical (e.g., uniform), or empirical, i.e., using some architectural insight such as the number of cores per die, or the number of cores sharing an L2 cache on each socket. Equation 3.4 presents the model extended to two samples, with an additional term λ to capture interaction between samples which we describe next.

$$uIPC_t = uIPC_{s1} \cdot \alpha_{(t,s1)}(e_{(1..n,s1)}) + uIPC_{s2} \cdot \alpha_{(t,s2)}(e_{(1..n,s2)}) + \lambda_t(e_{(1..n,S)}) + \epsilon_t \tag{3.4}$$

An additional benefit of using multiple samples is that the relationship between each configuration can be analyzed. Statistically, this is done by including an interaction term in the model that considers the product of two or more events as another term in the linear model. For simplicity, we limit possible interactions to between the same event across multiple configurations, meaning that the product of $uIPC$ on each sample configuration will be used as well as that of each particular event. Doing so allows the model to consider the interplay between multiple configurations. Specifically, we define the interaction term for a model using two samples as shown in Equation 3.5.

$$\lambda_t(1..n, S) = \sum_{i=1}^{n}(\mu_{(t,i)} \cdot e_{(i,s1)} \cdot e_{(i,s2)}) + \mu_{(t,IPC)} \cdot uIPC_{s1} \cdot uIPC_{s2} + \iota_t \tag{3.5}$$

The interaction term λ_t takes a linear combination of the products of each event across configurations, as well as that of $uIPC$. In Equation 3.5, μ is the target configuration-specific coefficient for each event pair and ι is the event rate independent term in the model.

On architectures with a particularly large number of possible configurations with complex performance dependencies, it may be necessary to increase the number of sample configurations. Our model can be extended up to an arbitrary collection of samples, as follows:

$$uIPC_t = \sum_{i=1}^{|S|}(uIPC_i \cdot \alpha_{(t,i)}(e_{(1..n,i)})) + \lambda_t(e_{(1..n,S)}) + \epsilon_t \tag{3.6}$$

While the use of more samples is expected to increase model accuracy, there is also an associated increase in sampling overhead. We address the selection of S in terms of specific configurations as well as its size in Section 3.6.

The term γ_t can be further generalized to account for the interaction between events across $|S|$ samples as follows:

$$\begin{aligned}\lambda_t(e_{(1..n,S)}) = \sum_{i=1}^{n}(\sum_{j=1}^{|S|-1}(\sum_{k=j+1}^{|S|}(\mu_{(t,i,j,k)} \cdot e_{(i,j)} \cdot e_{(i,k)}))) + \\ \sum_{j=1}^{|S|-1}(\sum_{k=j+1}^{|S|}(\mu_{(t,j,k,IPC)} \cdot uIPC_j \cdot uIPC_k)) + \iota_t\end{aligned} \tag{3.7}$$

To further improve model accuracy, we apply variance stabilization techniques in the form of a square-root transformation to the data to reduce the correlation between the residuals and the fitted values, as is done by Lee, et al. [27]. That is, we take the square-root of each term, as well as the response variable, before applying the model. This process results in a more accurate model by reducing model error for the largest and smallest fitted values and causing residuals to more closely follow a normal distribution.

3.3 Offline Model Training

To approximate the coefficients in our model, we use multivariate linear regression on phases from a set of training benchmarks. Each training benchmark's phases are executed on all considered configurations, while recording the $uIPC$ and a predefined collection of event rates at all target configurations. These values are then plugged into the model and multiple linear regression is used

to learn the patterns in the effects of event rates on the resulting $uIPC$ on the target configuration, with each phase's data serving as a training point. Specifically, the $uIPC$, the product of IPC and each event rate, and the interaction terms on the sample configurations serve as *independent variables* and the $uIPC$ on each target configuration serves as the *dependent variable*, in accordance with the above equations. We develop a model separately for each target configuration, with sets of coefficients being derived independently. The set of training benchmarks is selected so as to include variation in properties such as scalability and memory-boundedness. This allows the model to incorporate information for a wide range of possible event rates that could be experienced online.

Testing all sample and target configurations offline for training purposes may become a time consuming process on architectures with many processing elements and/or many layers of parallelism. To combat this, we prune the target configuration space, using insight on the target system architecture. Specifically, we eliminate symmetric cases in thread binding as well as unbalanced bindings of threads. For example the exact processor binding when only a single thread is used is not significant to performance. We also assume that the voltage/frequency of all dies in the system is set simultaneously to the same setting, to better support parallel codes and avoid load imbalance during parallel execution phases. On emerging architectures that feature hundreds of cores, it may become necessary to further reduce the search space during model training to limit offline overhead, for example by uniform sampling of the system configurations used for training. Another possibility is to only allow concurrency configurations that are symmetric, which would limit the search space by a constant factor, possibly reducing performance or energy gains, in some cases, but likely only by a small amount. At current multi-core system scales, the training process using a fully automated system for our approach takes on the order of hours, and scales up linearly with the number of possible configurations. However, since the offline training need only be performed a single time for a given machine, it may feasible to allow any configuration and simply pay the overhead.

We can train the models to predict the performance effects of concurrency throttling and fre-

quency change independently (DCT and $DVFS$ models), or across simultaneous changes across both dimensions ($Sequential$ and $Hybrid$ models). Each of these three models we use in our approaches has a distinct training set. Lets take the case of the hypothetical machine with m cores and n frequency levels. For such a machine, the DCT model uses the training data sampled at m different configurations with all cores running at frequency n_{max} (maximum frequency supported in the machine). As a result, it can only predict the $uIPC$ of configurations at which it sampled data i.e (m, n_{max}). We derive a separate $DVFS$ model for each concurrency level supported on the machine. We have m different $DVFS$ models, each of which uses the training data sampled at n different frequency levels, but only at mth concurrency i.e. m_p. Thus, each $DVFS$ model, can make the $uIPC$ predictions restricted to m_p, n different configurations.

Once these two, one dimensional models are derived, we can use them in combination to obtain the predictions across both the dimensions. This approach is referred to as $Sequential$ model. It effectively needs the training set to be sampled on $m * n$ different configurations. But, it can only use the models in a staggered fashion. It first decides on the optimal concurrency using the DCT model, say m_p and then uses the $DVFS$ model at m_pth concurrency to decide on the frequency level. The duration of the training process remains the same for the $Hybrid$ model. This model, again, requires that the training set be sampled at each point in the configuration space, i.e., it needs $m * n$ different samples. The difference being that in the $Hybrid$ model, we derive $m * n$ different models, each for a particular combination of concurrency and frequency. Hence, it can simultaneously decide on both the dimensions.

3.4 Predicting Across Multiple Dimensions

Like we mentioned earlier, we can apply our model to predict the performance effects of DCT and DVFS independently or across combined changes across both the power knobs. To predict for simultaneous changes, we collect samples at points along the two-dimensional space by varying the configuration along each prediction dimension. While we could predict along one dimension

at a time by selecting the optimal configuration in each dimension sequentially, predicting along both dimensions simultaneously avoids blind-spots in the predictions. The former strategy only predicts along the second dimension at the decided optimal level of the first dimension, whereas the second strategy is more likely to find the globally optimal configuration along both dimensions since it considers all combinations of both dimensions. We can generalize the model to predict performance in a configuration space of higher dimensions, and we can prune the space through uniform or other sampling schemes to reduce model training overhead.

3.5 Event Selection Process

The model requires feedback from hardware event rates in order to accurately predict performance across configurations. Therefore, it is necessary to identify particular event rates which result in high prediction accuracy. Unfortunately, it is not always obvious which events will be most closely associated with performance when power knob settings are changed. Due to the vast number of available counters on modern architectures, it is not possible to exhaustively search all possible combinations to identify an effective set. To select the events to use with our model, we use correlation analysis to determine which event rates on the sample configuration are most strongly correlated with the target IPCs. Then we select the top n events from the sorted list. We determine the number of events to use, n, based on how many event registers are available on the target architecture. It may happen that two or more selected events are highly correlated with each other as well. If this occurs, then it is necessary to remove all but one of the events because they provide no additional information for the linear regression, while their use may detract from the accuracy of the model through over-fitting. Further, removing redundant events allows for other events to take their place in the limited set of hardware event counter registers. The number of events to use, n, is determined by considering the number of hardware event registers available on the particular architecture, as well as the overhead required to perform additional iterations to collect more counters. For example, on a machine that can record four events concurrently, perhaps a

model would use four events, while if fewer registers are available, then multiple iterations may be necessary to record enough event rates. In this process, we follow no hard rule, rather, we use experimentation to strike a balance between minimizing overhead for collecting counter samples and maximizing accuracy in prediction. The event selection process is statistical and automated, therefore portable across multi-core architectures.

3.6 Selecting Sample Configurations

Every configuration used as a sample provides some additional data on what the performance will be on a different configuration. Although we could uniformly sample the configurations to reduce training and runtime search overhead, intuitively some configurations reveal specific architectural bottlenecks to scalability and performance. That is, certain configurations provide further insight into utilization of shared caches and memory bandwidth, and, thus, are stronger predictors than others. We therefore consider architectural properties while selecting the configurations that will best serve the prediction model.

When predicting along a single dimension (i.e., concurrency or DVFS-level), we can use a single sample configuration at the maximum concurrency or frequency available [8, 9]. When predicting along multiple dimensions, our experimental evidence suggests that effective samples are drawn by sampling at points along each dimension. In more detail, we first sample at the maximum concurrency and frequency and then select additional samples, guided by architectural intuition, to improve coverage along each dimension. Each additional sample can simultaneously test new points along multiple dimensions. For example, along the concurrency dimension of a four core system, four frequency levels, the first sample could use all four cores at full frequency and the second sample could use two cores at a different frequency level (thereby providing insight into changes of both the concurrency and frequency dimensions). While using the lowest frequency level may provide the most information in conjunction with the maximum frequency samples, it also comes with the highest potential overhead and hence it is not recommended. This technique

allows us to limit the number of samples while still providing significant input for the predictor along each dimension.

Chapter 4

Design and Implementation

4.1 The Runtime Adaptation Library

We have implemented the multi-dimensional prediction model as a runtime library which performs an online adaptation of concurrency and cpu frequency. We target parallel applications from the HPC domain with iterative structure, such that each program phase is executed many times. We make most of these iterative constructs by accumulating the necessary values of hardware counters which subsequently drives the model. We derive the regression coefficients through the training process and use them as part of the modeling subsystem within the runtime library. The library then utilizes these regression coefficients and the event rates in order to make online predictions about performance of targeted configurations.

Our library is designed to predict the most optimal configuration in terms of power and performance for an OpenMP application and it is implemented using portable components. The applications themselves need not be modified at all, except for a trivial piece of wrap-around code surrounding the OpenMP parallel regions as shown in the Figure 4.1. The calls to the control library are highlighted in the figure. They consist of the `start_region()` and `stop_region()` calls which serve as a marker to indicate the beginning and end of a parallel region. `start_region()` also allows us to throttle and bind threads to cores as well as to make appropriate changes in the

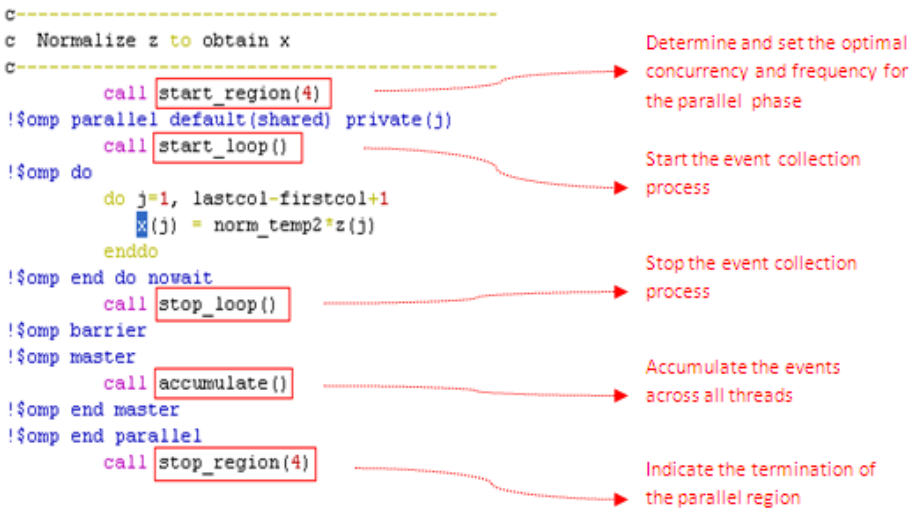

Figure 4.1: Code Instrumentation.

frequency as predicted by the model. We mark the beginning and end of counter collection process using the `start_loop()` and `stop_loop()` while `accumulate()` serves to collect and add the events across all the active processor cores. Since a single parallel region in itself might contain a lot a segregated regions performing distinct computations, we allow multiple sequences of `start_loop()`, `stop_loop()` and `accumulate()` calls to occur within an instance of `start_region()` and `stop_region()` calls. The runtime system is subdivided into the following major components:

- Collection of Events
- Prediction Models
- Phase Mapper
- DVFS/Concurrency Throttling

4.1.1 Collection of Events

We use PAPI [3] to collect the required events for the prediction models. The events are automatically sampled across multiple configurations in order to improve the accuracy. The number of iterations required to collect all the necessary counters varies from one approach to other but does not go beyond 3 iterations in our models. Each iteration might make numerous calls to the

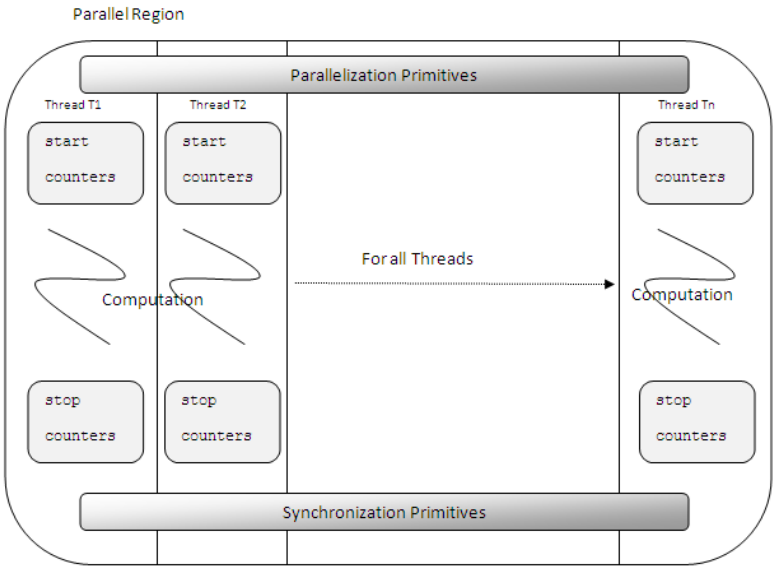

Figure 4.2: Collecting events across different threads in the Parallel Phase.

same parallel phase and hence in order to obtain a configuration which works for all of them, we collect the events across all the calls to the same phase in a single iteration. We use $rdtsc$ [18] to get the measure of time. In order to get a more accurate measure of the number of ticks per core, we bind the threads to cores and then calculate the difference in the number of ticks measured at the beginning and just before the end of the parallel phase for each of the threads. Figure 4.2 shows how these values are computed within `start_loop()` and `stop_loop()` functions we talked about earlier. We then select the maximum value of this difference across all threads as the measure of the time required to execute that phase. All the event rates are then calculated based on this value.

4.1.2 Prediction Models

The prediction models use the coefficients obtained during the training phase, to determine the optimal configuration for a particular parallel phase amongst the entire gamut of configurations supported by the processor. The configuration can be optimal in terms of performance or power or both as specified by the user. The default model always selects the configuration which is predicted to have the maximum $uIPC$ i.e. the configuration which has the maximum performance.

But we can also have a model which selects a particular configuration taking into into account the estimated increase/decrease in the power consumed if we switch from the current configuration to the predicted one. Since, existing approaches [21] incur a non-trivial overhead in order to estimate power at runtime, we use a relatively simple approach which statically derives power coefficients which indicate the relative increase/decrease in the power consumed when switching from a particular base configuration to any possible target configuration. In absence of tools which

Table 4.1: Power Coefficients for $Kimi$

Frequency(Ghz)	Threads				
	4	3	2s	2p	1
2.4	1	0.96	0.96	1.05	0.84
2.1	0.88	0.88	0.86	0.96	0.78
1.8	0.81	0.80	0.77	0.88	0.71
1.6	0.74	0.73	0.70	0.80	0.66

support measurements of energy consumed at the granularity of a parallel phase (some of them last for some microseconds only), we obtained these coefficients using the average drop in power we observe for the the same set of training applications when switching between various concurrency/DVFS configurations. Table 4.1 shows the power coefficients (W) for *Kimi*, a machine with two 2.4 GHZ Intel(R) Dual-Core CPUs, for a total of four processors and four DVFS levels ranging from 1.6 Ghz to 2.4 Ghz. 2s and 2p refer to the case when 2 threads are sharing the L2 cache (2s) and when the 2 threads have their own private cache (2p) respectively. Each cell in the table refers to the relative increase or decrease in the power consumption from the base case wherein all 4 cores are active and the processor is running at maximum frequency i.e. 2.4 GHz. We can then implement a policy which estimates energy per instruction as $((1/IPC) * W)$ or energy delay product *(EDP)* as $((1/(IPC)^2) * W)$. Thus, it takes into account both, the *delay* i.e. the performance penalty observed as well as the reduction in power consumed while selecting the target configuration. The ED metric can be expressed as shown in Equation 4.1:

$$(1/(IPC_{2.4,4}/IPC_x)^2) * W \tag{4.1}$$

Here $IPC_{2.4,4}$ is the IPC on 4 threads with 2.4 GHz, IPC_x is the target IPC predicted by IPC prediction equations, W is power coefficient for IPC_x. Using approaches similar to the *ED* model, we can easily implement policies which focus more on performance, giving ED^2 model or more on energy so that we have E^2D model [12] or and so on. While $uIPC$ serves as an accurate metric for comparisons of performance across different configurations during DCT, its value becomes invalid when the very duration of a cycle changes when we throttle the frequency of a core using DVFS in one of the hybrid approaches. In order to provide for an accurate comparison, we normalize the IPC at each frequency to the $IPnS$ or instructions per nanosecond. This $IPnS$ value is then used to estimate the performance. For example, a value of 1.4 at a frequency 2.4 Ghz is not equal to the value of 1.4 at 1.6 GHz. Their normalized values, $1.4 * 2.4$ and $1.4 * 1.6$, are then used for performance comparison.

4.1.3 Phase Mapper

After the predictor selects the most optimal configuration for a particular phase, we need to decide whether it is indeed worthwhile to change to that configuration. Our experiments have shown that it takes in the order of thousands of cycles in order to switch concurrency and change cpu frequency. If we apply these controls for every phase without omissions, it results in a non-trivial degradation of performance. This is because a program may have phases that are of too fine granularity to benefit from adaptation using either DVFS or DCT. The overhead involved simply exceeds the benefits obtained in this case. We have empirically identified a threshold of one million cycles, below which we use the currently active configuration when entering a phase. In practice, most application phases are much longer than the selected threshold; however short phases do exist and may distort performance significantly, if their locally optimal configurations differ from the optimal configurations of adjacent dominant phases.

While profiling the benchmarks we found that a parallel phase might be called repeatedly during the same iteration with different durations. The Figure 4.3 shows how the duration of a phase varies within an iteration of the LU-HP, MG and SP . As seen, the phase for MG is being

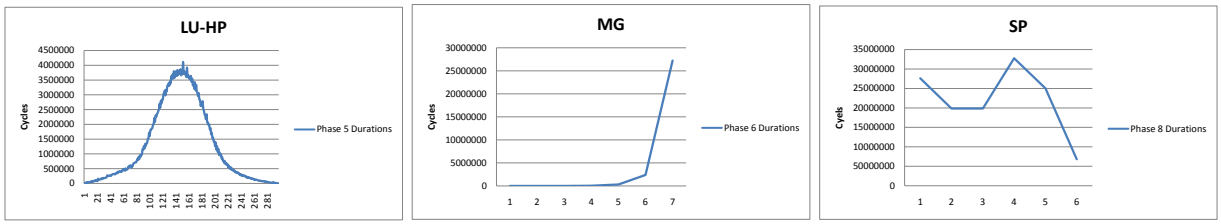

Figure 4.3: Variation in duration of phases within a single iteration.

executed many times for a really short duration before spiking up. Same is the case for LU-HP which follows the distinctive bell-curve for the duration for its phases. But, the sum of all the durations for the phases for that single iteration might still cross the million cycle threshold. Hence we might end up changing concurrency even for the times when the phase is being executed for really short duration, and thereby hurt the overall performance.

Effectively, we now have the order and duration of each parallel phase executed within an iteration. In the subsequent iterations, we query this call graph for the next expected phase and its corresponding duration and if it is below the threshold, we do not perform adaptation. The *phase mapper* reduces the overhead associated with adaptation in benchmarks, especially in LU-HP, where two phases were being called alternatively with increasing workloads. Since each of these two phases had a locally optimal configuration, adaptation would necessitate a switch to a new configuration after the end of each phase. *Phase mapper* skips over all the instances when the parallel phase was being called for a duration less than the threshold and thereby it improves the performance of by over 5%.

4.1.4 DVFS/Concurrency Throttling

Once the initial few iterations, necessary for prediction analysis and creation of call graph are completed, we have a in-memory record of the best configurations for each parallel phase within the application. For the remaining iterations, library reads from this record and switches to appropriate number of threads and frequency depending on the decision made by the *phase mapper*. Additionally, when entering a new phase, we ensure that each setting is only changed if it is not already

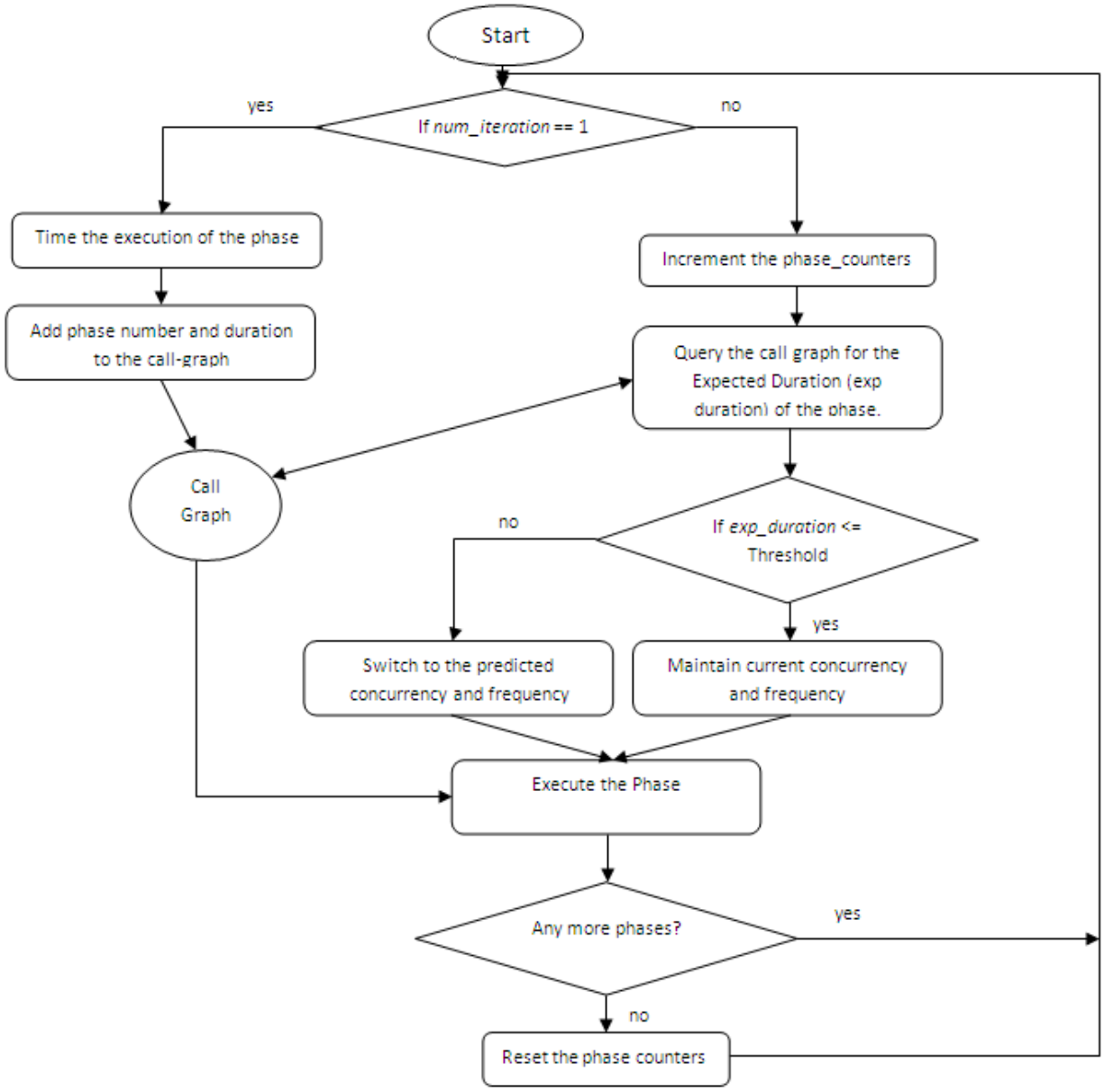

Figure 4.4: Phase Mapper.

set correctly for the phase. The library also controls the placement of the threads on cores so that they are placed as required by the predictor. We use the `omp_set_num_threads()` call to set concurrency within OpenMP and thread bindings with the Linux processor affinity system call, `sched_setaffinity()`. We do not permit overloading of a core so that each of the cores runs exactly 1 thread at any particular time and the maximum number of threads running at any particular time is equivalent to the maximum cores available within the machine. We set DFVS levels using the *cpufrequtils* library [2]. We use the `sysfs_modify_policy_governor()` call to initially set the governor to *userspace* which then allows us to dynamically change the frequency for each of the cores using the `sysfs_set_frequency()` call. The decisions for frequency made by the predictor are global i.e. we do not let different cores run on different frequencies since we assume a uniform workload across all the threads bound to each of the cores. After making predictions, the library uses the predicted optimal configurations for all subsequent traversals of each phase. Several forms of adaptation are possible through our runtime library. The simplest ones optimize either DCT or $DVFS$ but not both during a given run by using the corresponding model to predict the effects of that power-performance knob. The training for DCT model would consist of collecting the events while executing all the benchmarks in the training set across all possible concurrency levels supported on an architecture at the maximum frequency. Similarly we train the $DVFS$ model by collecting the events across different frequencies at maximum concurrency. We consider two mechanisms to adapt DCT and DVFS in a single program run. First, we apply the two individual models sequentially to adapt first concurrency and then apply DVFS accordingly on the cores that are kept active. We refer to this model as the $Sequential$ prediction model. Second, we create a new model that simultaneously predicts changes in both concurrency and frequency, which we refer to as the $Hybrid$ prediction model.

4.2 Heuristic Searches

We have implemented two empirical search approaches to identify optimal frequency and concurrency configurations. A straightforward approaches to finding the optimal configuration is by searching through the possible combinations until the best one is found. Search approaches have a few tradeoffs when compared to the adaptation techniques described earlier. The downside to searching methods is that they have the potential to perform worse since by the nature of a search several sub-optimal configurations will likely need to be examined before finding one that is truly optimal. This will cause delays that will ultimately affect the overall runtime and possibly power as well. It is important to note that power usage and runtime are closely linked and the longer the execution time stretches the more power will be consumed by the system even if it is operating at a throttled configuration. The benefit to searching methods however is that they can be broadly applied, they are hardware independent in that they don't require specific hardware counters to be supported on the system doing the prediction, and there is no training time required.

The metric we use for the performance measurements in these search schemes is execution time. By measuring the number of clock cycles from just after the parallel phase has begun to just before it ends (similar to the event collection process we described in Section 4.1.1), we are able to get the number of clock cycles needed for the slowest of the execution threads to complete. This is possible due to the barrier synchronization mechanism inherent at the end of OpenMP parallel regions. The reason this is necessary is that it provides us with the overall phase execution time and is not dependent on the number of threads that are executing. By dividing the number of clock cycles by the CPU frequency that the phase was executed at, we are able to derive a fine-grained execution time. By using the per-phase execution time as our metric, we are able to directly compare each adaptation configuration by its execution time.

4.2.1 Exhaustive Search

Exhaustive searching in our research was used mostly as a control test for the prediction accuracy of the other models we used. Just as its name implies, exhaustive search does a sequential traversal of all the configurations available on the system. This approach does not require any offline training, so the programmer can use it with minimal effort. For each number of cores, each phase was run at every DVFS level possible and the execution time taken was recorded. For each phase, once the linear search across both dimensions of frequency and concurrency was complete, we switch to the configuration with the best execution time. Obviously this has some limitations based on the number of iterations each phase goes through. In the case of our four-core machine that has four different DVFS levels, we required a minimum of 16 iterations per phase to converge on an optimal configuration. Obviously this is a taxing overhead and not all applications would even support that number of iterations, but again, we were mainly utilizing this scheme to analyze the accuracy of the other methods.

4.2.2 Binary Search

We have also implemented a heuristic based search scheme as proposed by Li and Martinez [29]. This approach conducts a binary search across the two dimensions to canvass the configuration space. While, a fair direct comparison between our prediction models and the approach discussed previously [29] is not possible, since our evaluation was conducted on a real system with a different workload (NAS benchmarks) contrary to the simulation-based study in previous work. Also, the adaptation through binary search was implemented by timing the execution time of phases during a single run instead of over multiple runs of each benchmark. Nevertheless, we believe that our comparison can still provide some useful insight on the trade-off involved in use of heuristic searches and prediction-based approaches to dynamic program adaptation.

Our implementation of binary search, as its name suggests, traverses the combinations of threads and frequency levels in a binary fashion. To do this, it traverses the number of cores first, and then once it has converged on an optimal concurrency level, it performs another binary search

through the different frequency levels. This method is used to address the primary limitation of the exhaustive method which is the inability to scale well on larger systems and the relatively slow convergence. For instance, in our eight-core machine with two frequency levels, it would take our exhaustive search more than 20 iterations of each phase to select the optimal configuration, whereas in the binary search method, it would only take five. This number also scales particularly well as the number of cores or frequency levels increases. The actual formula for the number of iterations required is

$$Log_2(N * F) + (M - 1) \tag{4.2}$$

Here N is the number of cores and F is the number of frequency levels. The value M in this equation indicates the number of different thread-mappings available on the system. For instance, a system with 1024 cores, four DVFS levels and four-thread mapping choices would end up taking only $Log_2(1024 * 4) + (4 - 1)$ or 15 iterations, which is less than the exhaustive case on our much smaller machine. While this is a significant increase in efficiency over the exhaustive method and scales well for larger systems, it still fails in case of smaller benchmarks which do not have that many iterations for the binary search to sample.

Chapter 5

Experimental Analysis

In this section, we discuss the evaluation our multi-dimensional prediction model on two machines with different properties. We begin with a brief description of the experimental setup. For each machines, we analyze the scalability of the benchmarks on the target machines pointing out configurations which allow for optimal performance and power consumption. Then, we evaluate the performance prediction models used to apply frequency (DVFS) and concurrency throttling (DCT). Finally, we compare the benefits of applying DVFS and DCT independently and synergistically, in terms of both performance and energy benefits.

5.1 Experimental Setup

We have implemented our models and ported them for two different machines. The first, *Hamilton*, has two Intel Xeon E5320 quad-core processors, for a total of eight cores. As seen in the Figure 5.1, each of two pairs of cores within a chip shares a 4MB L2 cache, creating an asymmetry in scheduling decisions in that two threads can be scheduled on a single chip in two different ways, with cache sharing and without it. Each core operates at a maximum frequency of 1.86 GHz, with the possibility of reducing to 1.60 GHz. The system contains 4GB of memory, and runs Linux kernel version 2.6.22.

The second machine, *Kimi*, has two 2.4 GHz Intel(R) Dual-Core processors, for a total of four

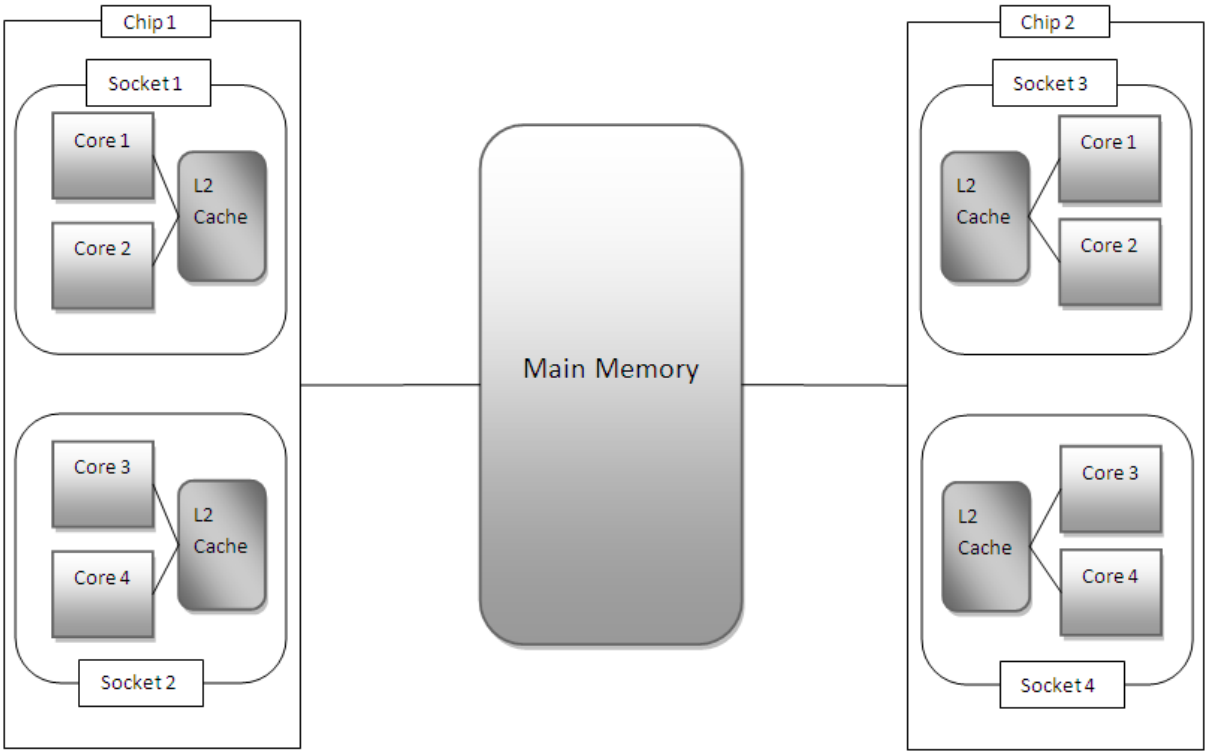

Figure 5.1: Layout of *Hamilton*.

processing cores, and 4 MB L2 cache shared by the two cores on a chip. This machine supports four DVFS different levels: 2.4 GHz, 2.1 GHz, 1.8 GHz, and 1.6 GHz. It has 2 GB of memory and runs on linux kernel version 2.6.18. The diverse nature of these two architectures allows us to explore the dimensions of concurrency and frequency and their effect on power and performance in great detail. The linux kernels on both these machines had to be recompiled with the *perfctr* patch in order to support the Performance Application Programming Interface (PAPI) used in our runtime system.We use Watts Up Pro power meter to collect the energy consumed by an entire system (although we focus on optimizing core utilization) during the execution of the benchmarks. We have used benchmarks which resemble the core of the applications deployed in the domain of high performance computing in all our experiments. Specifically, we use nine benchmarks from the OpenMP version of the NAS Parallel Benchmarks suite, version 3.1, compiled at class size B. These benchmarks are derived from computational fluid dynamics (CFD) applications and are widely used to evaluate the performance of parallel supercomputers. These benchmarks have

Table 5.1: Characteristics of the NAS Parallel Benchmarks.

Benchmarks	BT	CG	FT	IS	LU	LU-HP	MG	SP	UA
Iterations	200	15	6	10	250	250	4	400	200
Regions	10	14	8	5	10	20	13	15	59
Re-execution Regions.	5	5	5	1	3	11	6	9	49

Source: [8]

several distinctive and interesting properties which led us to use them for our experiments. They are listed below:

- The number of parallel regions or phases being executed as well as the number of times they were executed within the application varies widely as seen in Table 5.1.
- The applications have a wide range of global and phase-local optimal concurrency.
- They differ widely in terms of the computation performed and memory accesses required for their execution.

These properties contrive to make the decision to select the optimal configuration, in terms of energy consumption and performance, really challenging. Hence, it serves as an ideal evaluation tool for our library. Next, we look at the results from the experiments on *Hamilton* followed by *Kimi*.

5.2 Results on *Hamilton* (8 processing cores, 2 DVFS levels)

5.2.1 Application Scalability Analysis

We begin by profiling scalability of all applications in the benchmark suite and determine its effect on the performance and energy consumption. In order to do that, the applications are executed with a variable concurrency ranging from one to eight threads and bound to processor/core combinations as shown in Figure 5.2. The notation (X, Y) denotes non-adaptive (static) execution with $X * Y$ threads bound to X processors and Y cores per processor. The notation $2s$ indicates a shared cache and $2p$ indicates private caches. The cores marked in green are being used by the

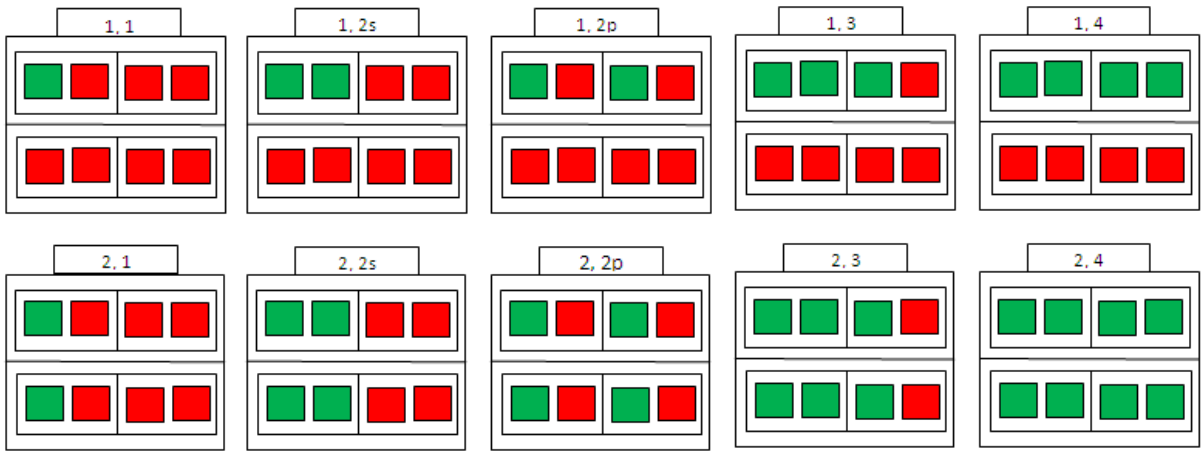

Figure 5.2: Thread to Processor/Core Bindings.

threads. The bindings shown, are not an exhaustive collection of bindings possible on this machine. For example, two threads can be bound in many different ways across the eight cores, but we explore only three of those. This is because, for our experiments, we have selected only those bindings which are unique in terms of performance and energy characteristics they exhibit. All the benchmarks in the NAS suite print out the time of execution at the end of a successful run. We use this time as the measure of performance in the analysis. The corresponding energy consumption readings are also recorded.

Figure 5.3 presents the results of the scalability analysis on *Hamilton*. As stated earlier, two threads on a single chip can execute with shared or private caches on this architecture. The notation (X, Y, Z) denotes non-adaptive execution with X processors and Y cores per processor with each core running at Zth DVFS level ($1 \Rightarrow$ 1.6 Ghz and $2 \Rightarrow$ 1.8 Ghz).

Figure 5.3 shows that, in general, applications are far from scaling perfectly on the target machine. In particular, only two applications achieves its best performance using all 8 cores. We observe essentially three categories of scalability in our experiments. First are those applications that manage reasonable speedup through the utilization of additional cores (BT, LU, LU-HP and UA). BT, LU and LU-HP show that the machine allows linear speedup with appropriately written code with speedups of 2.67x, 3.5x and 3.07x respectively. They also reduce their energy consumption proportionally indicating an optimal use of the processors. Second are applications that incur

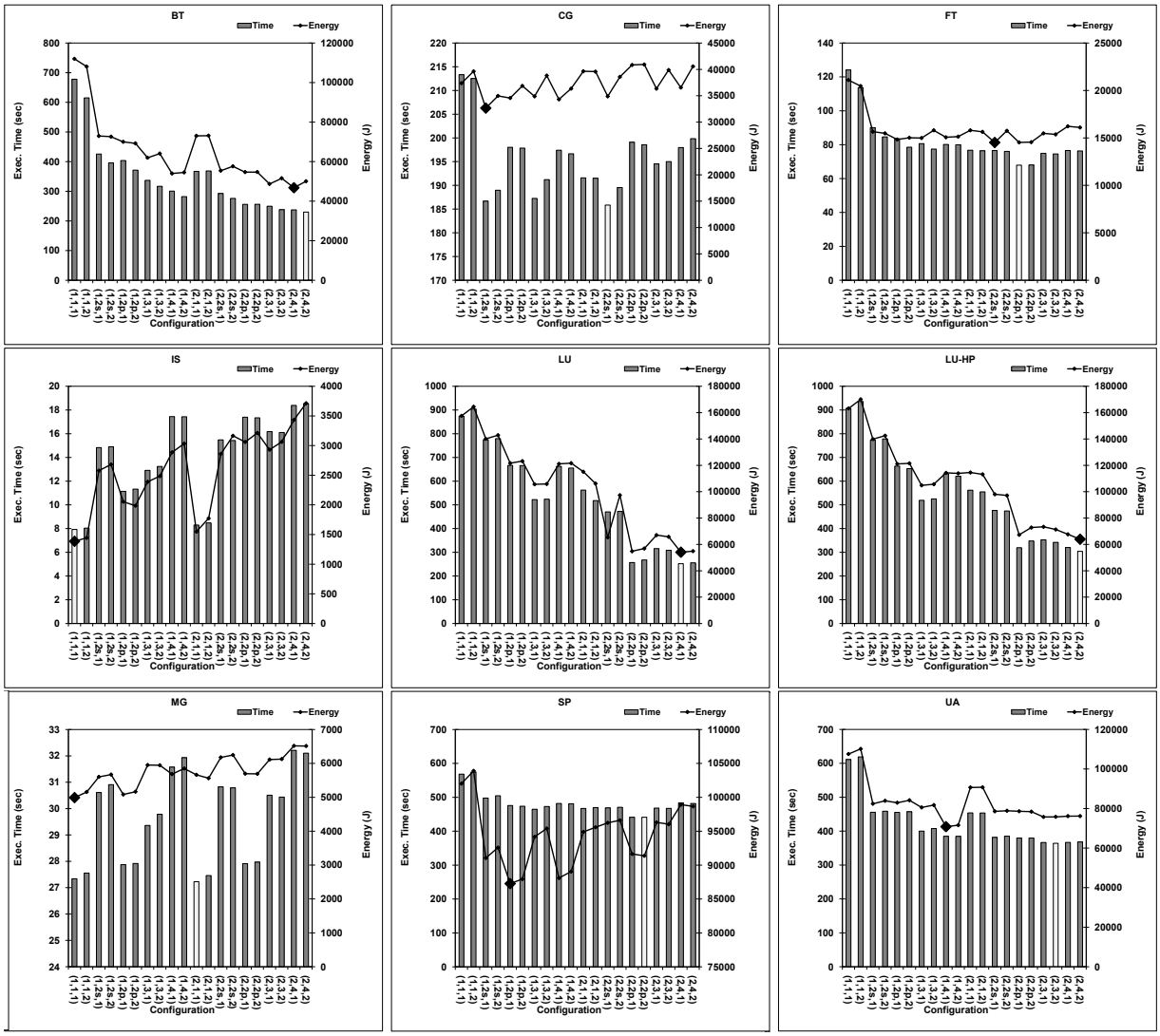

Figure 5.3: Execution time (bars) and energy consumption (lines) of the benchmarks across all configurations. The configurations with the best performance and energy for each benchmark are marked with a light gray gradient and a large diamond respectively.

a non-negligible performance *loss* when using *more* cores. Both IS and MG fall under this category losing the performance by as much as 2.3x and 1.16x times respectively over single threaded execution. This shows that both these benchmarks are essentially memory bound and lose performance due to a high degree of contention for this resource with increased concurrency. The optimal configuration for both these applications, $(1, 1, 1)$ for IS and $(2, 1, 1)$ for MG, is when threads are bound to cores such that they do not have to share the cache. Lastly, we have the applications that neither substantially gain nor lose performance from higher concurrency (CG, FT, and SP). Despite the poor utilization of additional cores, energy consumption generally increases with more cores.

The most energy-efficient configuration coincides with the most performance-efficient configuration for 4 out of the 9 benchmarks (BT, IS, LU and LU-HP). For 5 benchmarks (CG, FT, MG, SP, and UA), the user can use fewer than the performance-optimal number of cores, to achieve substantial energy savings, at a marginal performance loss.

Thread-mapping also plays a role in affecting the performance of certain applications. This is because the thread to core bindings directly affects the behavior of both, memory accesses and spatial locality. CG, FT and SP reach their optimal configuration at $cache - sharing$ sensitive configurations ($(2, 2s, 1)$ for CG, $(2, 2p, 1)$ for FT and $(2, 2p, 2)$ for SP) with speedups of 1.1x, 1.3x and 1.1x respectively and increased number of threads beyond that just manages to increase the CPU utilization and hence energy consumption without significantly affecting performance. It is interesting to note that substantial performance benefits can seen through execution with private caches in the two and four thread cases. Benchmarks like BT, FT, LU, MG, and SP perform significantly better when the threads are bound to cores such that they do not share the L2 cache. Both, LU and LU-HP improve their performance by as much as 45% and 32% respectively when running with $(2, 2p)$ configuration over $(2, 2s)$ configuration.

The poor scalability of the benchmarks is mainly due to contention for memory and cache resources. It is generally beneficial to bind threads to cores so that they can have uncontended access to the L2 cache. Memory bound applications and applications with large working sets rarely

benefit from increased concurrency since memory bandwidth is limited and increased number of threads only aggravates the demand for it. It is predominantly this category of applications with negative scalability that motivates the use of DCT to throttle concurrency to more efficient levels. Especially in the cases where the performance stabilizes at low concurrencies, it is very beneficial to lower the CPU utilization and hence the energy consumption through DCT and DVFS.

5.2.2 Performance Prediction Evaluation

In this section, we evaluate the accuracy and correctness of the prediction models. It is important to discuss the various factors that affected and contributed towards this accuracy.

- The Training Set

  If the runtime system is to function and make correct decisions regarding the configurations selected, it is very important to train the prediction models rigorously. Hence, it is essential to select the benchmarks which would have as diverse a set of characteristics as possible. We have experimented with a wide variety of training sets and found that training with NAS-UA allows us to have the most accurate models. While increasing the training set to include other benchmarks improved the accuracy marginally, the difference was not significant and it ends up taking a lot more time. We decided to stick with a smaller training set containing just UA, preferring less training time over potentially more accurate prediction model.

- The Sampled Configurations

  The ability of the prediction model to make correct decisions also depends largely on the configurations at which the events are sampled. Again, it is important here to expose the model to as diverse a set of runtime configurations as possible. The sampled configurations needed to exercise the two dimensions of concurrency and frequency to the maximum, so that the models would have the most amount of information while making predictions. Again, we experimented with various sample configurations and selected the ones which resulted in a clear improvement in the accuracy of the predictions. The selected configurations are as shown in Table 5.2. The configuration (X, Y, Z) indicates X, Yth binding as shown in

Figure 5.2 while the Z term indicates the frequency level. The models then predict the performance across all the configurations which were not part of the sampling set.

Table 5.2: Sampled Configurations for *Hamilton*

Model	Number of Configurations Sampled	Sample Configurations
DVFS	1	(2,4,2)
DCT	2	(1,3,2), (2,4,2)
Hybrid(DVFS-DCT)	3	(1,2p,2), (2,2s,1), (2,4,2)

- The Number of Events Monitored

  Increasing the number of events being monitored has a direct advantage towards increasing the accuracy of the prediction models. But current architectures have a very restricted number of event registers which are essential to monitor these events. This, in turn, imposes hard limits on the number of events we can record at the same time. PAPI allows the users to multiplex many events across the available event registers but this method has a non-trivial overhead and limited accuracy. As a result, we limit the events collected to the number of event registers available on our experimental machine. Hamilton supports only two event registers and we use one of them to monitor the $uIPC$ which is essential for our models. In order to select the other model, we analysed a set of thirty different events picked intuitively. The event with the highest degree of statistical correlation with the target IPC in the training data was then selected as the second event for our models. It turned out that for all three models, that selected event was L1 data cache accesses. We then used simple linear regression techniques on the sampled event rates of L1 data cache accesses and $uIPC$ to derive the co-efficents for the three models.

All the factors described above contribute towards the overall accuracy of our models. The accuracy is measured based on the ability of the model to correctly determine the $uIPC$ of a configuration using the sampled value of the event rates for the benchmark. The models are then used to predict the $uIPC$ of all the configurations (upto a maximum of 20 on *hamilton*) which are not being sampled. Figure 5.4 shows the percent of predicted samples for each model with error less

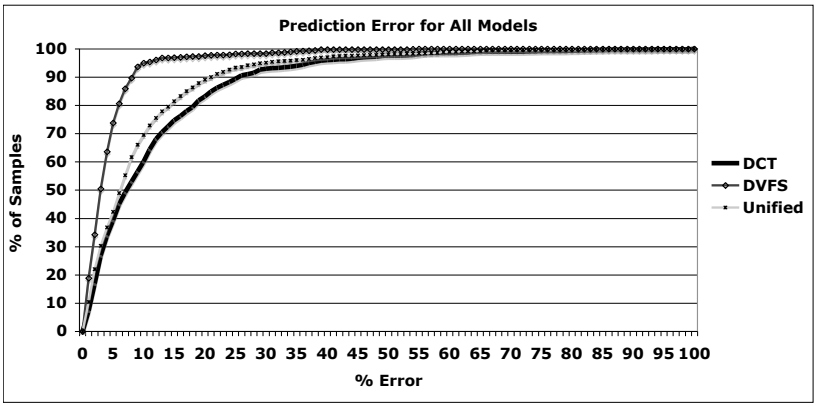

Figure 5.4: Cumulative distribution functions of prediction accuracy of the three prediction models.

than a particular threshold indicated on the x-axis. The results demonstrate high accuracy of the model in all three cases. In particular, the $DVFS$ model yields a median error of only 3.0% (4.2% mean), the DCT model a median of 7.3% (11.2% mean), and the unified model a median of 6.1% (9.5% mean). The higher accuracy of the predictions in case of $DVFS$ compared to DCT approach results from a simpler set of effects in changing the frequency level that our model captures easily, while changing concurrency has complicated performance effects, due to the irregular, non-monotonic scalability patterns of many phases. Of the twenty possible configurations, the $Hybrid$ model correctly identifies the single best configuration in 35% of cases, one of the top three in 51.3% of phases, and in only 7% of phases are any of the ten worst configurations incorrectly selected. The predicted optimal configuration is on average 6.1% slower than the true optimal configuration. These results indicate that the models are an accurate means of attaining performance estimates to tune power-performance parameters without requiring potentially expensive empirical searches. The results compare favorably with similar empirical models [9, 27].

5.2.3 Evaluation of DCT and DVFS Adaptation

In this section, we evaluate the performance of the NAS suite of benchmarks when integrated with the runtime library using our prediction models. We begin by comparing the use of only $DVFS$ or DCT. We then analyze two schemes for adapting both $DVFS$ and DCT, more specifically for applying them sequentially or in a unified manner. Finally, we compare prediction-based adaptation against the use of empirical search techniques in identifying optimal configurations. Figure 5.5

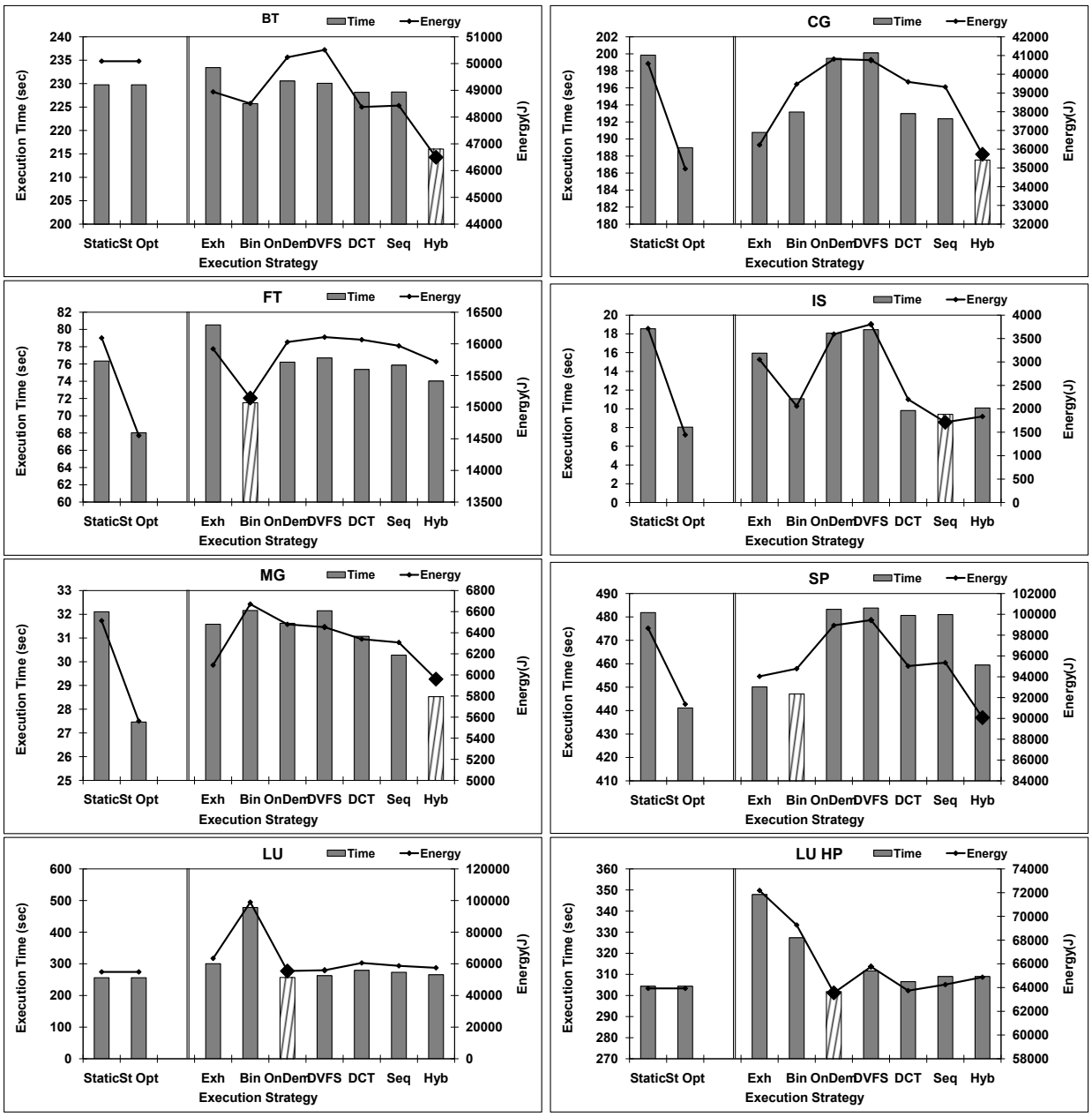

Figure 5.5: Results of adaptation through various techniques. The group of bars left of the divider represent static configurations and those right of the divider are the adaptive strategies. The adaptive configurations with the best performance and lowest energy for each benchmark are marked in stripes and a large diamond respectively.

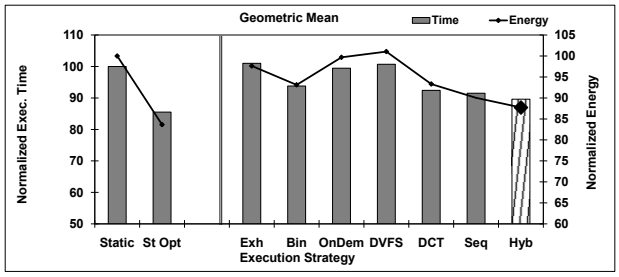

Figure 5.6: Geometric means of the benefits of adaptation through various strategies. The adaptive configurations with the best performance and energy overall are marked in strips and a large diamond respectively.

presents the results of adaptations through the various mechanisms for eight different NAS parallel benchmarks. Figure 5.6 shows the geometric mean of the normalized energy and execution time results for the different approaches. Figure 5.7 gives the geometric mean of ED and ED^2 across different techniques. ED represents the energy delay product while ED^2 represents the product of energy consumption and the square of execution time. These are both popular metrics for energy-efficiency in high performance computing.

We compare the various adaptation strategies against the results of static executions, which uses a single configuration for the entire execution. We measure the performance and energy consumption our approaches against the configuration using full concurrency and frequency ($Static$) and to the best performing of all static executions ($Static\ Optimal$). The $Static\ Optimal$ configuration for the particular benchmark is the configuration which was statically found to be the best performing one amongst all other possible configurations through exhaustive offline experimentation. We use static optimal as a potentially unrealistic, baseline of comparison for the other strategies. The static optimal, however, is not necessarily the overall optimal execution point, as each phase may have its own dynamic optimal configuration. We do not consider this possibility as its identification requires exponential time, making it unrealistic even for offline use.

Adaptation along one dimension ($DVFS$, DCT):

In this analysis, we make adaptation decisions by selecting the configuration with the highest predicted performance, because HPC applications in general must maintain maximum performance. Poor scalability of the benchmarks (only BT, LU and LU-HP perform optimally at highest concurrency) and fewer frequency gears leaves very few options for $DVFS$ model alone to improve

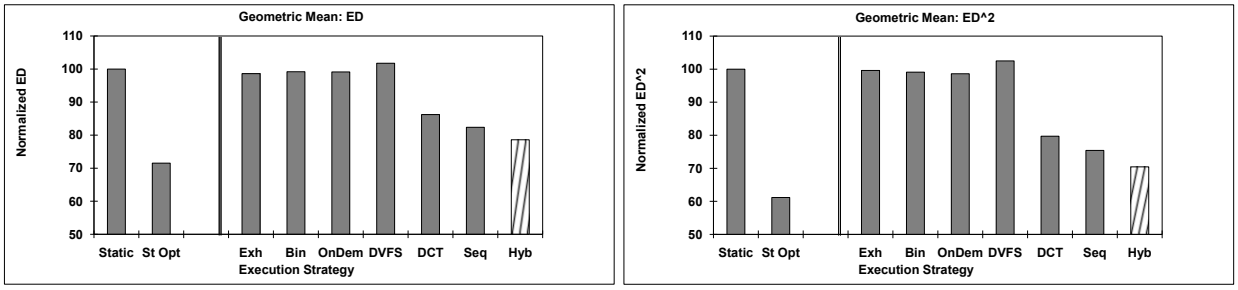

Figure 5.7: Geometric means of the benefits of adaptation through various strategies. The adaptive strategy with the best mean ED and ED^2 is marked with stripes.

performance significantly. Even amongst these three benchmarks, only LU has its peak performance at the lower frequency gear. Due to the rarity of such cases, the $DVFS$ model ends up running at the highest frequency gear most of the time. Specifically, our experiments reveal no benefit in terms of performance or energy from adapting to the frequency level with the highest predicted performance. Without allowing for some loss in performance, $DVFS$ alone is generally not able to significantly benefit energy consumption. Hence we see the model with DVFS performing within 1% (geometric mean) of the $Static$ executions in terms of energy, performance as well as power.

We also experimented with *Ondemand* [33], an existing mechanism for automatically changing frequency at runtime. *Ondemand* is tool for automatically applying DVFS that also uses hardware performance monitors for guidance [33]. It is an in-kernel tool that works by monitoring dynamic application CPU utilization to reduce processor frequency at times of low load and uses the same *frequtils* interface as we exploit. It, and most other similar tools, uses Intel ACPI, which is an interface to allow changes in DVFS levels and processor power states when idle, with transition criteria determined by the adaptation system. We can apply *Ondemand* in modes similar to our DVFS adaptation only and our Sequential strategies, through which we achieve similar results (within 1% for energy consumption and nearly identical run time). These results demonstrate that our DVFS scheme is competitive with state of the art DVFS tools. While lack of sufficient frequency gears on this processor restricts us on *Hamtilton*, we are able to exercise the its advantages more extensively in Section 5.3 when we evaluate our models on *Kimi*.

On the other hand, using DCT prediction model provides substantial benefits in execution time (7.5% mean avings) and energy consumption(6.67% mean savings), and ED^2 (20.3% mean savings) compared to the $static$ execution. Despite the overall success of DCT, mispredictions for LU result in an observed increase in execution time by 9.5% and energy consumption by 10.3%. In contrast, the largest benefit occurs with IS which sees a 41% reduction in energy consumption while improving performance by a 48.1%. When compared to the static optimal execution, DCT is within 7% mean performance. These results indicate that at least at the scale of a few multi-core processors integrated on a single node, DCT is able to leverage the poor scalability of benchmarks to improve performance and energy-efficiency by substantial margins.

Adaptation across two dimensions ($Hybrid$**,** $Sequential$**):**

It is possible to integrate the two power-performance control knobs of DCT and $DVFS$ in multiple ways. First, we consider applying them sequentially in the $Sequential$ approach. We decided to first apply DCT and then $DVFS$ on the active cores in each phase rather than the other way around, since DCT has a clear advantage over $DVFS$ in reducing power while improving performance, as exhibited in our earlier experiments. We note that this may not be necessarily true in other experimental platforms that allow more fine-grain control of voltage/frequency. $Sequential$ is able to improve performance by 2% in IS and by 4% in LU over the DCT approach, by exploiting the few opportunitites to switch to more performance optimal configuration at a lowered frequency. In case of the remaining benchmarks, the optimal configuration identified by DCT does not perform very well at lower frequencies. Hence there is no room for DVFS to scale down the frequency. As compared to the $Static$ values, it is able to reduce execution time by 8.44% and energy consumption by 9.96% and performs within 5.95% of the $Static\,Optimal$ configuration.

The $Hybrid$ model facilitates a simultaneous prediction of the effects of switching DVFS and DCT. We have utilized this prediction for concurrent adaptation of both approaches. The major advantage of the this unified prediction approach is that it eliminates blind-spots in the configuration space during the prediction process. Whereas sequential application of DVFS and DCT will

only evaluate DVFS options on the decided DCT level, the unified approach considers all possible values of each parameter at a single stage. Further, the unified scheme uses the same number of execution samples as the sequential approach, however it uses all samples for both DVFS and DCT models, instead of dividing them. It is able to exploit its higher accuracy to identify more effective DCT levels and finds opportunities to apply DVFS that do not hurt performance. Because of its advantages over the $Sequential$ approach, the $Hybrid$ scheme improves performance by 1.9% and reduces energy by 2.3% (geometric mean improvements over the sequential approach). When compared to the default execution using maximum concurrency and frequency, the advantages of $Hybrid$ adaptation become even clearer. Specifically, we see an 10.3% speedup simultaneous with a 3.9% reduction in power consumption, resulting in an overall reduction in energy consumption of 12.27% and ED^2 of 29.6% (geometric mean improvements over static execution on all cores). In fact, all benchmarks experience improved performance and reduced energy consumption, and the $Hybrid$ scheme achieves the lowest execution time and energy consumption amongst all adaptation approaches in six of eight benchmarks. Even when compared to the oracle-derived executions on the $Static\ Optimal$ configuration for each benchmark, unified adaptation achieves energy consumption within 4.1% and performance within 4.07% (geometric means), and better performance by 6.4% in the case of BT due to identification of improved per-phase configurations. This indicates that prediction models are both viable and effective in addressing the multi-dimensional program adaptation problem.

Prediction vs. heuristic search approaches

As described in Section 4.2, we have implemented exhaustive and heuristic search approaches to identify optimal frequency and concurrency configurations. The first of these performs an exhaustive search of the configuration space before making a decision, while measuring the execution time of phases with each configuration. However, the online overhead of testing many possible configurations stands to reduce any potential benefit of adaptation considerably, which is what occurs in practice. When compared to the $Static$ values, the exhaustive search method increases execution

time by 1.01%, while reducing power by 3.4%, energy by 2.4%, and ED^2 by 0.04%, well below the savings of our prediction-based techniques. The only place, Exhaustive search proves superior to prediction schemes is in SP, wherein it improves performance by 6% over the $Sequential$ and DCT models. Due to the considerably reduced overhead compared to the exhaustive search, the binary search results in 7.22% better performance and 4.5% lower energy consumption (geometric mean improvements over exhaustive search). Compared to the static execution, performance is improved by 6.22%, energy by 6.8%, and ED^2 by 19.2%, however power consumption is only reduced by 0.8%. This suggests that a heuristic search can be effective in the context of adapting DCT and DVFS at runtime. However, it still falls short of the $Static\ Optimal$ configuration by 8.2% for performance and 9.4% for energy.

The most interesting comparison is between the $Hybrid$ prediction model and binary search. Binary search achieves performance 4.17% worse than the $Hybrid$ prediction approach while consuming 5.3% more energy and seeing a 11.45% increase in ED^2 (geometric mean differences). Binary search essentially suffers from same deficiencies as the $Sequential$ model did in that it cannot select a configuration with a non-optimal concurrency which might have an performance optimal lower frequency level. For example, the optimal configurations of CG is $(2, 2s, 1)$, which will be completely missed by even the most accurate DCT and $Sequential$ approaches which, in turn, will incorrectly converge on $1, 2s, 2)$, the optimal configuration. These blind-spots prevent identification of effective configurations at low concurrency or DVFS levels, which tend to consume less power. Binary search does have better performance than the $Hybrid$ model in two of eight cases (FT and SP), however energy consumption is higher in all but one case (FT). In particular, the results of binary search suffers for MG and IS because they contain too few iterations to amortize the search overhead, in contrast to BT and SP, where binary search excels since the applications execute 200 and 400 iterations respectively.

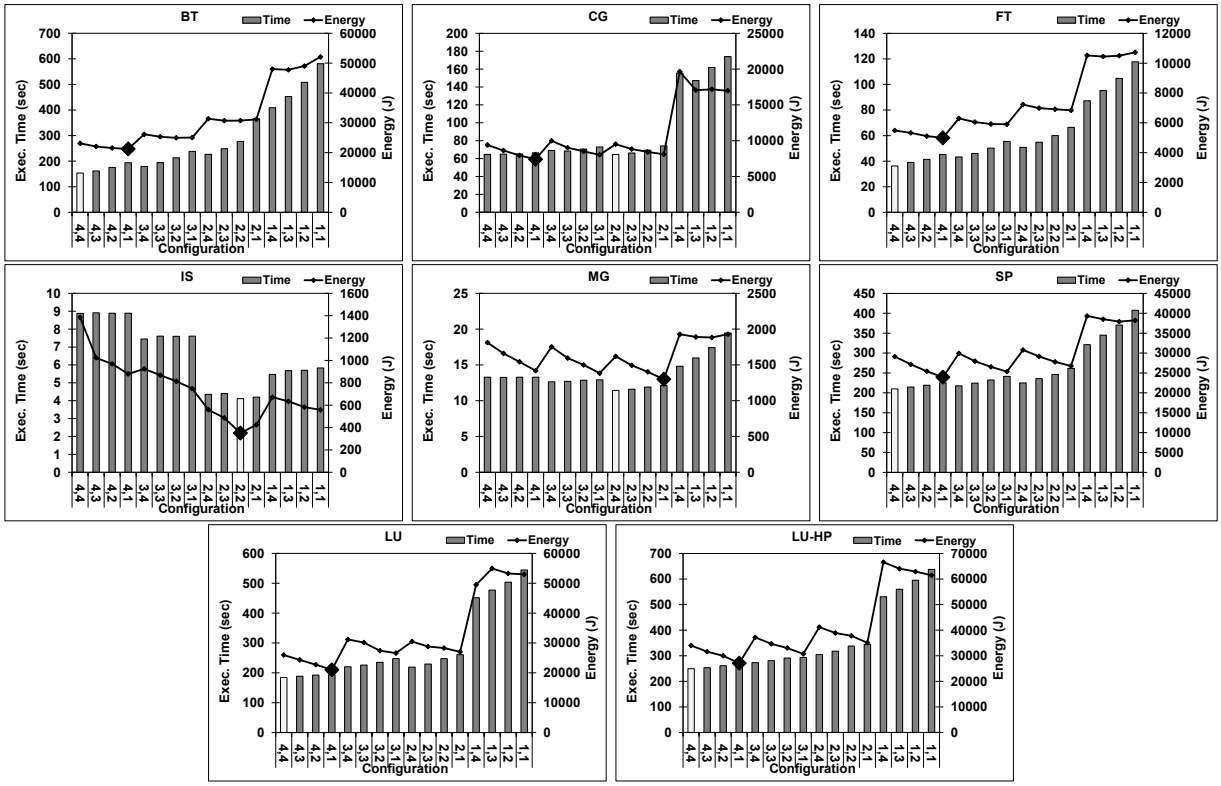

Figure 5.8: Execution time (bars) and energy consumption (lines) of the benchmarks across all configurations. The configurations with the best performance and energy for each benchmark are marked with a light gray gradient and a large diamond respectively.

5.3 Results on *Kimi* (4 processing cores, 4 DVFS levels)

5.3.1 Application Scalability Analysis

While *Hamilton* allowed us to explore the performance characteristics of our models with a focus on concurrency control, *Kimi* highlights the opportunities presented by a larger set of frequency gears.

A clear trend emerges from the static scalability analysis of the benchmarks as shown in Figure 5.8. The notation (X, Y) implies X number of threads and Yth frequency and ranges from $4 \Rightarrow 2.4$, $3 \Rightarrow 2.0$, $2 \Rightarrow 1.8$, $1 \Rightarrow 1.6$ Ghz. The $Static\ Optimal$ configuration for five out of the eight benchmarks (BT, FT, SP, LU, and LU-HP) turns out to be the $(4, 4)$ configuration. This indicates that most of these benchmarks scale very well upto four threads (IS, CG and MG being the exceptions). Hence, we can expect that if a model based its decisions by selecting the best perform-

ing configuration (configuration with highest $uIPC$ in our case), we would always select the static $(4, 4)$ configuration and would never have an opportunity to throttle frequency or concurrency. Another key point to note here is that the least energy consumption for a benchmark always occurs at the lowest frequency level corresponding to the optimal concurrency level. In other words, given that a benchmarks has least execution time at configuration (X, Y), the lowest energy is consumed at the configuration (X, Y_{min}) where Y_{min} is the minimum frequency supported on this machine. For example, when we move from the $(4, 4)$ configuration, to $(4, 1)$ configuration we see an average drop of 17% in energy consumption across the benchmarks. At the same time there is only a 10% increase in the execution time. This characteristic can be used very effectively to reduce the energy consumption by reducing the frequency through DVFS while tolerating some loss in performance, as we describe later on in Section 5.3.4.

5.3.2 Performance Prediction Evaluation

In this section we evaluate the accuracy of our prediction models trained for *Kimi*. As in the case of *Hamilton*, we used UA for training all the $DVFS$, DCT, $Sequential$ and $Hybrid$ models for *Kimi*. Again, in addition to $uIPC$, the most statistically significant event for all our prediction models was found to be L1 data cache accesses. Since *Kimi* also supports the monitoring of only two event registers simultaneously, we used the event rates for these two events as an input for our models. We sampled the events for the various models at configurations presented in Table 5.3.

Table 5.3: Sampled Configurations for *Kimi*

Model	Number of Configurations Sampled	Sample Configurations
DVFS	1	(4,4)
DCT	2	(2,4), (4,4)
Hybrid(DVFS-DCT)	2	(2,2), (4,4)

The predictions models are fairly accurate with the median error of 4% (7.5% mean) for the $DVFS$ model, median of 7.3% (9.2% mean) for the DCT approach and median of 9.6% (11.3% mean) for the $Hybrid$ approach.

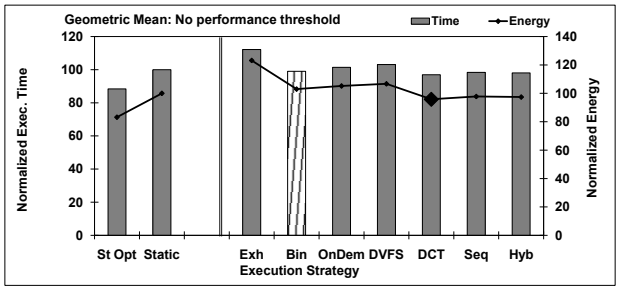

Figure 5.9: Geometric means of the benefits of adaptation through various strategies. The adaptive configurations with the best performance and energy overall are marked in strips and a large diamond respectively.

5.3.3 Evaluation of DCT and DVFS Adaptation *without* performance thresholds

Figure 5.9 shows the excution time and energy consumption characteristics of the different adaptation and search techniques used. The single dimensional models DCT and $DVFS$ perform close to the $Static$ values as do the two dimensional models $Sequential$ and $Hybrid$. There are no opportunities for the $DVFS$ model to throttle frequency since the $uIPC$ of a configuration with a higher frequency is higher in most of the observed data. Hence it runs at the $static$, $(4, 4)$ configuration for most of the benchmarks degrading performance marginally. At the same time, it performs comparably with automatic DVFS switching performed using the $ondemand$ governor. The models with the concurrency dimension, DCT, $Sequential$ and $Hybrid$, are able to leverage the performance benefits offered by switching to a lower concurrency in case MG, CG and IS and hence improve performance by 3.1%, 1.7% and 2% (geometric means) respectively over $static$ executions. At the same time they reduce the energy consumption by 4.3%, 2.4% and 2.6% mean respectively. The search strategies behave similarly with $Binary$ search managing to improve performance by 1.1% at the same time increasing energy consumption by 2.9%. As expected, the $Exhaustive$ search is the worst performer. It consumes upto 23.1% more energy and takes upto 12.1% longer to execute the benchmarks using the exhaustive searches when compared to $static$ executions. The adaptation approaches compare closely to the $Binary$ search approach since benchmarks scale pretty uniformly and due to absence of any blind-spots, rarely do the configurations predicted by adaptation approaches differ from the decisions made by the $Binary$ search.

5.3.4 Evaluation of DCT and DVFS Adaptation *with* performance thresholds

The results get much more interesting if we tolerate a 15% loss in performance so that the models have an option to select a configuration with a $uIPC$ value which is 15% less than the best predicted configuration. The models can now explore the frequency dimension freely, directly affecting the overall energy consumption. The evaluation of adaptation approaches using such a performance threshold mechanism are shown in Figure 5.10. We also present geometric means in Figure 5.11 and the $E^2 * D$ metric in Figure 5.12 to emphasize the benefits of the reduction in the energy consumption using the thresholds.

$DVFS_{15\%}$ and $DCT_{15\%}$ behave very differently to the threshold mechanism. Since $DCT_{15\%}$ can only switch to a concurrency level with maximum frequency, it ends up retaining the optimal configuration in most cases. For example, in case of BT, a change from $(4, 4)$ to $(3, 4)$ would result in a drop of 17% in performance and hence it would require a bigger threshold to actually tolerate the switch to a lower concurrency. The $DCT_{15\%}$ model manages to reduce the energy consumption by a further 5.4% exploiting a couple of phases where the $uIPC$ values at lower concurrency fell within the threshold, resulting in an overall reduction of 9.4% over $static$ executions. At the same time the performance suffers an increase of 8.1% over DCT executions with no threshold but it is still within 2.1% of the $Static$ values.

When we use the $DVFS_{15\%}$ approach, there are many more opportunities for the model to actually select configurations with lower frequency levels that fall within the 15% performance threshold. This results in a further 16% reduction of energy consumption over the $DVFS$ executions with no thresholds so that there is an effective reduction of 9% over $Static$ executions. However, we now see a 9% increase in the execution time when compared to the $Static$ executions. The benchmarks which benefit most from the threshold mechanism are MG and IS who see a drop in energy consumption by 19% and 22% respectively. Similarly, the $Hybrid_{15\%}$ as well as $Sequential_{15\%}$ approaches reduce their energy consumption by a further 13.1% and 11.4% respectively to have an total reduction of 15.3% and 14.1% while only suffering a 4.9% and 8.2%

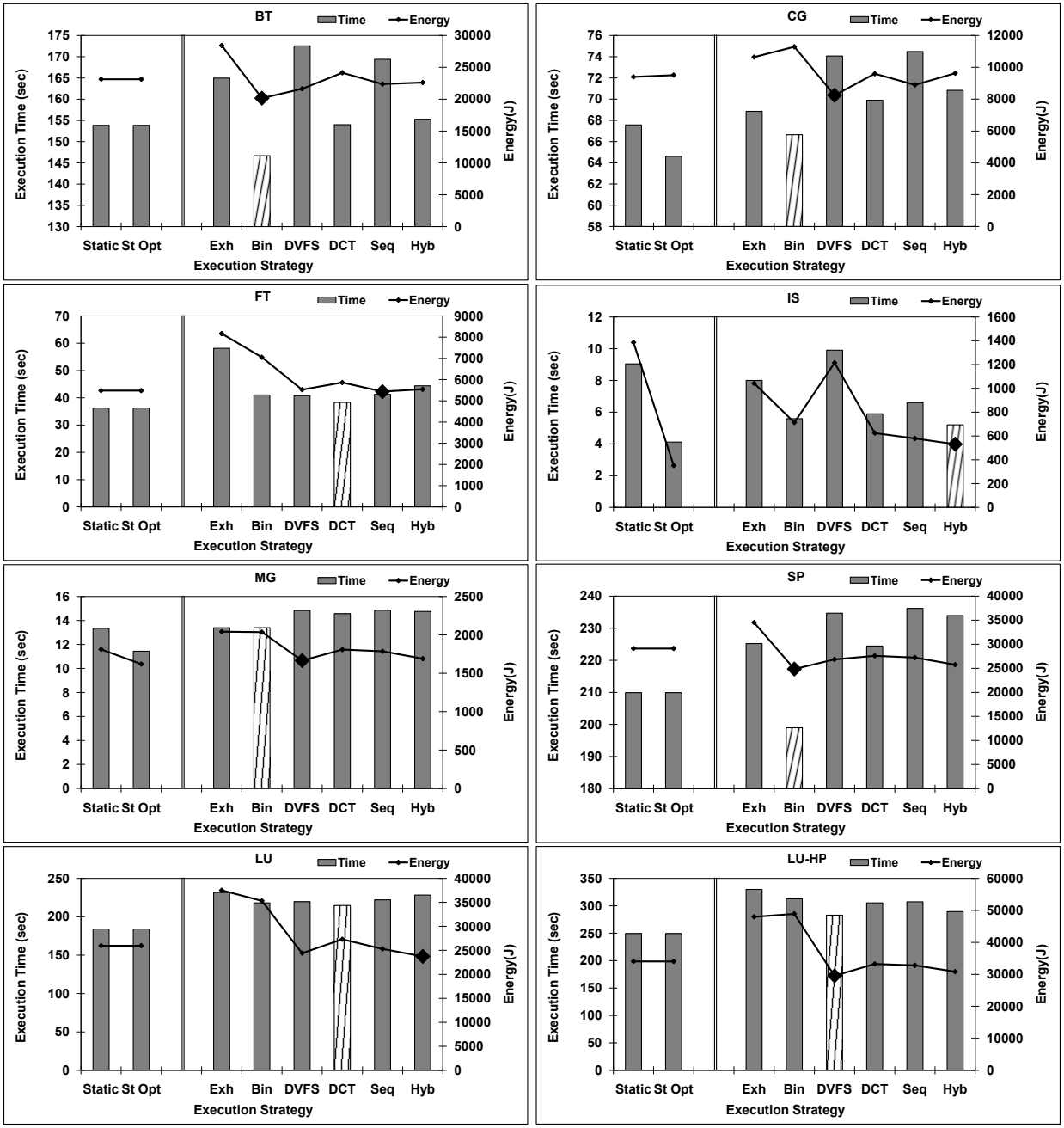

Figure 5.10: Results of adaptation through various techniques with a 15% $Performance\ threshold$. The group of bars left of the divider represent static configurations and those right of the divider are the adaptive strategies. The adaptive configurations with the best performance and lowest energy for each benchmark are marked in stripes and a large diamond respectively.

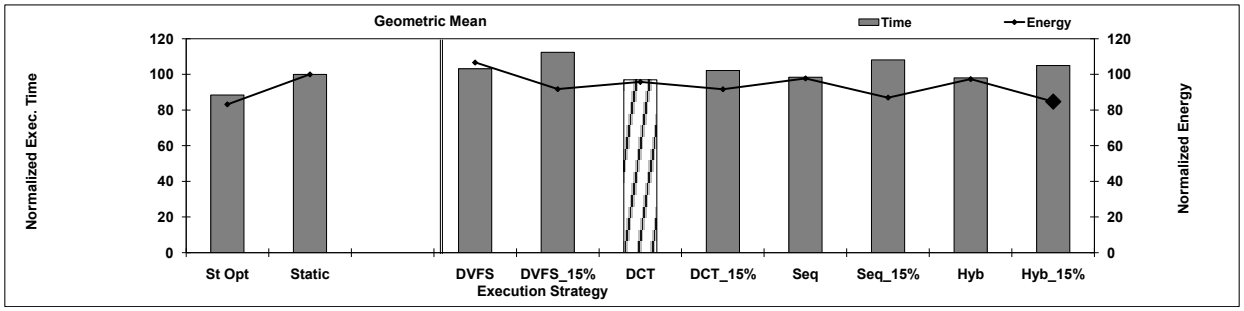

Figure 5.11: Geometric means of energy and performance values for adaptation strategies before and after applying a 15% threshold.

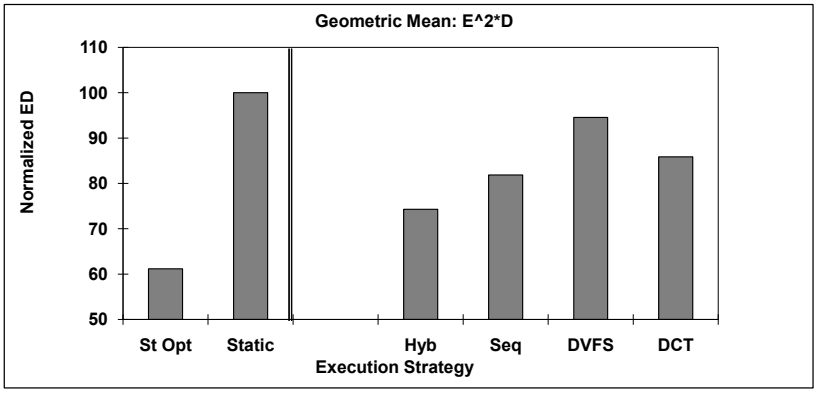

Figure 5.12: Geometric means of $E^2 * D$ values after applying a 15% threshold.

reduction in performance when compared to the $Static$ executions. Most importantly, the $Hybrid$ and $Sequential$ models come within 3.52% and 5.8% of the $Static\ Optimal$ in terms of energy consumption.

While, no approach is the best for every benchmark, for applications with few outermost iterations, prediction is certainly the best while still being competitive for large applications. As future systems continue to increase in parallelism as well as the number of DVFS levels available, the overhead of searching is expected to increase and the relative benefit of prediction is expected to grow. We conclude that not only does the $Hybrid$ prediction model outperform isolated application of DVFS and DCT, it also surpasses the prediction models and heuristics which apply both there control mechanisms.

Chapter 6

Related Work

Research on software-controlled dynamic power management has focused extensively on controlling voltage supply and frequency in sequential microprocessors. This research has derived analytical models for DVFS [39], compiler-driven techniques [40], and control-theoretic approaches [37]. Similar techniques have been employed to reduce dynamic power management in system components other than processors, such as RAM [10] and disk [6]. Researchers have recently modeled and analyzed the impact of a single control knob, either DVFS or concurrency throttling, on dynamic power management on shared-memory [9, 19, 30, 34], and on distributed-memory parallel systems [13, 16, 36].

Our work differs from earlier research on power-aware adaptation using a single knob in several key aspects. First, it achieves two-dimensional adaptation. Second, it leverages a scalable performance prediction model, instead of direct measurements or static analysis of idle execution intervals. Third, it analyzes the busy intervals of parallel computation to exploit opportunities for power savings and performance improvement simultaneously, as opposed to exploiting only slack time to reduce power. Fourth, it uses a model that is general and versatile: it can accommodate different optimization targets – both performance-centric and energy-centric – with ease and it is developed with an automated and portable methodology. In terms of actual implementation, the proposed model leverages phase-aware adaptation at the granularity of parallel loops, which has

been explored before in compiler-based DVFS algorithms for multiprocessors [30, 40].

Prediction models for adaptation via concurrency throttling were introduced by Curtis-Maury, et al [8, 9]. The current work makes several new contributions in the context of performance prediction for power-performance adaptation. We consider prediction models for DVFS and DCT simultaneously, effectively exploring a larger and significantly more challenging runtime optimization space. We draw comparisons between alternative power-performance adaptation methods and present effective strategies to synthesize multiple power-performance adaptation methods in software. Furthermore, we propose methods to generalize multi-dimensional prediction models using sampling of the target configuration space and we significantly improve prediction accuracy compared to previous work [8, 9], thus achieving better optimization with zero tolerance for performance loss. We improve earlier regression models for cross-configuration prediction [8, 9] through such techniques as variance stabilization, explicit consideration of event and configuration interactions, and architecture-aware sampling.

Our contribution shares similar objectives with research presented by Li and Martinez [29], whose work evaluated search algorithms for DVFS and DCT, so as to meet specific performance targets under a given power budget. Our work differs in that it uses statistical prediction models instead of direct search methods and that it considers only power-performance adaptation schemes that do not penalize performance, effectively targeting the more performance-sensitive HPC environments. The use of prediction makes our contribution an attractive alternative for runtime adaptation, since the number of hardware event counter samples needed to predict across all concurrency and voltage/frequency configurations of a system can be very small (2–3) compared to the samples needed by any search strategy. Thus, the performance of prediction-based adaptation scales more gracefully with the number of cores and voltage-frequency levels than search methods, while being highly competitive at small system scales.

Concurrency throttling is also proposed by Zhang and Voss [42]. In this work, the number of threads used per processor on an SMP on Intel Hyperthreaded processors is dynamically adapted at runtime based on live sampling. Jung et al. [23] also attempt to identify the number of threads to

use per SMT processor, however they consider the use of performance prediction using regression on performance counters collected online. However, these works do not consider energy effects of throttling and they greatly limit the degrees to which concurrency can be throttled.

A lot of research has used Dynamic voltage and Frequency Scaling (DVFS) to identify regions within a application which are not CPU intensive and scaling down frequency during those regions. The objective in most cases is to reduce power consumption while tolerating small loss in performace [4, 11]. Ge et al. perform runtime DVFS adaptation based on an analytical model [14] (as against the empirical model used in our research) which predicts the expected performance drop when switching between various frequency gears. They use an adaptation sytem called CPUMISER to decide the optimal target frequency based on the performance thresholds specified by the user. Research has also tried to model the impact of reduced frequency on the execution time using MIPS rate [16]. These β-algorithm [15] powers down the cpu by switching to a lowered frequency during phases where the program is heavily accessing memory. Sasaki et al. [35, 25] implement a method for performance prediction at various degrees of DVFS. Their model is based on multiple linear regression on performance counters collected at runtime: they identify the level with the lowest power consumption that meets a specified performance requirement and use that for each phase. Prediction-based DVFS adaptation is integral our research and is applied synergistically with prediction-based concurrency throttling to maintain low overhead in identifying an optimal execution point in this two-dimensional space.

Several researchers have previously explored the use of hardware event counters for characterizing performance and power properties [20, 31, 38]. Our models differ in that they provide cross-configuration predictions with multi-dimensional inputs, using hardware event counters. As such, they achieve accurate statistical correlation between event samples and performance, across a potentially large and hard to search system configuration space. In this respect, our work is more closely related to regression and machine learning methods for performance prediction and design space exploration on parallel architectures [27, 28]. Both our contribution and earlier regression-based performance prediction methods use statistical analysis of the correlation between multiple

parameters and performance. The key difference is that our framework is used for online workload adaptation, rather than for off-line exhaustive exploration. Our prediction model is more simplistic than models used in design space exploration, however it is fast enough to use in runtime optimization.

Chapter 7

Summary and Future Work

7.1 Summary

The number of cores integrated in a single microprocessor is increasing at an almost exponential rate; however most applications, even from the highly specialized HPC domain, fail to exploit many cores. We have shown that HPC applications observe performance losses even beyond modest concurrency levels on an 8-core system. In this paper, we presented a model to predict the performance effects of applying multiple energy saving techniques simultaneously. The model applies statistical analysis of hardware event rates to estimate how voltage/frequency scaling and dynamic concurrency throttling influence performance in application phases and across system configurations. Over a range of benchmarks, our model achieves a median error of only 6.1% in prediction, in response to simultaneous tuning of DVFS and DCT. The high prediction accuracy allows for the successful identification of efficient operating points and phase-aware adaptation in HPC applications.

We have applied our model to adapt program execution by regulating concurrency as well as DVFS levels. Our results indicate that while DVFS on its own is not ideal for the HPC domain where performance is critical, DCT is an attractive solution. Further, we find that combining the two approaches in a synergetic fashion, can simultaneously improve performance and energy-

efficiency relative to either approach in isolation. Specifically, a $Hybrid$ adaptation model achieves performance improvements of 10.3%, power savings of 3.9%, and energy savings of 12.27% compared to using all cores at full frequency, outperforming an approach which sequentially applies DCT and DVFS. We also compare our prediction model to methods using exhaustive or binary search that time system configurations. We find that while binary search outperforms exhaustive search, it is not necessarily superior to the prediction-based approach due to overhead and blind-spots. In fact, since it only searches through one dimension (either frequency or concurrency) at a time it can only come close to identifying optimal configurations as suggested by the $Sequential$ search (assuming that the $Sequential$ search has zero prediction error). It can easily overlook the best static configuration predicted by the $Hybrid$ approach. As we scale to more cores and DVFS levels, the overhead of search-based approaches is likely to increase, widening the advantage of prediction. Given that the performance of prediction-based methods can effectively approximate the performance of an oracle, we conclude that they are a viable alternative for future-generation systems with many cores and fine-grain power control capabilities.

7.2 Future Work

Our work is not without limitations, which we will attempt to address in future research. A linear regression model buys speed for runtime adaptation, at the cost of accuracy. More elaborate models, such as piecewise polynomial approximation or neural networks, may improve prediction accuracy, at the cost of increased runtime overhead. A detailed analysis of this trade-off is needed to draw more accurate conclusions. Adaptation schemes have both direct and indirect costs while switching system configurations. Direct costs stem from the actual switching overhead, while indirect costs stem from gradual redistribution of the working set of the application between cores and caches. Our multi-dimensional prediction model currently does not account for any indirect costs of adaptation. Both the selection of samples during training/actuation and the configuration interaction terms in the model need to be revisited to account for interference between adjacent

phases. Preliminary investigation of this problem shows that although cross-phase interference is not necessarily acute on small-scale platforms such as the one evaluated in this paper, it is far more noticeable on large-scale multiprocessors, such as NUMA systems.

While we continue to work on more accurate models, some of the on-going work also deals adding more dimensions to the framework. More specifically, we have also been working developing frameworks which would allow OpenMP applications to perform with optimal *thermal* properties along with power and performance optimalities. We are working towards integrating the DCT models with a runtime temperature control component [5] which would result in a tool which allows us to execute a thermally stable application without loosing any of the performance benefits.

Adaptation capabilities are not readily available in all applications, as they are often prohibited by the semantics of the programming environment. For example, MPI applications are typically much harder to implement adaptively than OpenMP applications. Addressing this issue will require efforts to make parallel programming runtime environments more amenable to dynamic concurrency throttling, and extension of our prediction model to take into account any implications of the programming model and the runtime environment on adaptation.

Bibliography

[1] M. Azimi, N. Cherukuri, D. Jayashima, A. Kumar, P. Kundu, S. Park, I. Schoinas, and A. Vaidya. Integration Challenges and Tradeoffs for Tera-scale Architectures. *Intel Technology Journal*, August 2007.

[2] D. Brodowski. cpufrequtils. http://www.kernel.org/pub/linux/utils/kernel/cpufreq/cpufrequtils.html.

[3] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters. In *Proc. of Supercomputing'2000*, November 2000.

[4] K. Cameron, R. Ge, and X. Feng. High-Performance, Power-Aware Distributed Computing for Scientific Applications. *IEEE Computer*, 38(11), November 2005.

[5] Kirk W. Cameron, Hari K. Pyla, and Srinidhi Varadarajan. Tempest: A portable tool to identify hot spots in parallel code. In *ICPP '07: Proceedings of the 2007 International Conference on Parallel Processing*, page 37, Washington, DC, USA, 2007. IEEE Computer Society.

[6] E. V. Carrera, E. Pinheiro, and R. Bianchini. Conserving Disk Energy in Network Servers. In *Proc. of the 17th International Conference on Supercomputing*, June 2003.

[7] K. Chakraborty, P. Wells, and G. Sohi. A Case for an Over-provisioned Multicore System: Energy Efficient Processing of Multithreaded Programs. Technical Report TR-1607, Department of Computer Sciences, University of Wisconsin-Madison, 2007.

[8] M. Curtis-Maury, F. Blagojevic, C. D. Antonopoulos, and D. S. Nikolopoulos. Prediction-Based Power-Performance Adaptation of Multithreaded Scientific Codes. *IEEE Transactions on Parallel and Distributed Systems*. Accepted, to appear, 2008.

[9] 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 International Conference on Supercomputing*, June 2006.

[10] Bruno Diniz, D. O. G. Neto, W. Meira Jr., and R. Bianchini. Limiting the Power Consumption of Main Memory. In *Proc. of the International Symposium on Computer Architectures*, June 2007.

[11] 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.

[12] Vincent W. Freeh, David K. Lowenthal, Feng Pan, Nandini Kappiah, Rob Springer, Barry L. Rountree, and Mark E. Femal. Analyzing the energy-time trade-off in high-performance computing applications. *IEEE Transactions on Parallel and Distributed Systems*, 18(6):835–848, 2007.

[13] R. Ge, X. Feng, and K. W. Cameron. Performance constrained Distributed DVS Scheduling for Scientific Applications on Power-aware Clusters. In *Proc. of Supercomputing*, November 2005.

[14] Rong Ge, Xizhou Feng, Wu chun Feng, , and Kirk W. Cameron. CPU MISER: A Performance-Directed, Run-Time System for Power-Aware Clusters. In *Proc. of the International Conference on Parallel Processing*, September 2007.

[15] C. Hsu and W. Feng. Effective Voltage Scaling through CPU-Boundedness Detection. In *Proc̀of the Fourth Workshop on Power-Aware Computer Systems*, December 2004.

[16] C.-H. Hsu and W. Feng. A Power-Aware Run-Time System for High-Performance Computing. In *Proc. of Supercomputing'05*, November 2005.

[17] C. Huang, O. Lawlor, and L. Kale. Adaptive MPI. In *Proc. of the 16th International Workshop on Languages and Compilers for Parallel Computing, LNCS 2948*, 2003.

[18] Intel Inc. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2B: Instruction Set Reference, N-Z. pages 251–252. http://developer.intel.com/design/processor/manuals/253667.pdf, 2003.

[19] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi. An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget. In *Proc. of the International Symposium on Microarchitecture*, December 2006.

[20] C. Isci and M. Martonosi. Runtime Power Monitoring in High-End Processors: Methodology and Empirical Data. In *Proc. of the Annual International Symposium on Microarchitecture*, December 2003.

[21] Canturk Isci, Gilberto Contreras, and Margaret Martonosi. Live, Runtime Phase Monitoring and Prediction on Real Systems with Application to Dynamic Power Management. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, 2006.

[22] E. Joseph, A. Snell, C. G. Willard, S. Tichenor, D. Shaffer, and S.s Conway. Council on Competitiveness Study of ISVs Serving the High Performance Computing Market. July 2005.

[23] C. Jung, D. Lim, J. Lee, and S. Han. Adaptive Execution Techniques for SMT Multiprocessor Architectures. In *Proc. of the Symposium on Principles and Practice of Parallel Programming*, June 2005.

[24] T.S. Karkhanis and J.E. Smith. A First-Order Superscalar Processor Model. In *Proc. of the 31st International Symposium on Computer Architecture*, June 2004.

[25] M. Kondo, H. Sasaki, and Hiroshi Nakamura. Improving Fairness, Throughput and Energy-Efficiency on a Chip Multiprocessor through DVFS. In *Proc. of the Workshop on Design, Analysis, and Simulation of Chip Multiprocessors*, Orlando, FL, November 2006.

[26] S. Kumar, H. Raj, K. Schwan, and I. Ganev. Re-architecting VMMs for Multicore Systems: The Sidecore Approach. In *Proc. of the 2007 Workshop on the Interaction between Operating Systems and Computer Architecture*, June 2007.

[27] B. C. Lee and D. M. Brooks. Accurate and Efficient Regression Modeling for Microarchitectural Performance and Power Prediction. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2006.

[28] B. C. Lee, D. M. Brooks, B. R. de Supinski, M. Schulz, K. Singh, and S. A. McKee. Methods of Inference and Learning for Performance Modeling of Parallel Applications. In *Proc. of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, November 2007.

[29] J. Li and J. Martinez. Dynamic Power-Performance Adaptation of Parallel Computation on Chip Multiprocessors. In *Proc. of the International Symposium on High Performance Computer Architecture*, February 2006.

[30] C. Liu, A. Sivasubramaniam, M. T. Kandemir, and M. J. Irwin. Exploiting Barriers to Optimize Power Consumption of CMPs. In *Proc. of the 19th International Parallel and Distributed Processing Symposium*, April 2005.

[31] A. Merkel and F. Bellosa. Balancing Power Consumption in Multiprocessor Systems. In *Proc. of EuroSys Conference*, April 2006.

[32] T. Moseley, J. Kim, D. Connors, and D. Grunwald. Methods for Modeling Resource Contention on Simultaneous Multithreaded Processors. In *Proc. of the 2005 International Conference on Computer Design*, October 2005.

[33] V. Pallipadi and A. Starikovskiy. The Ondemand Governor. In *Proc. of the Ottawa Linux Symposium*, July 2006.

[34] S. Park, W. Jiang, Y. Zhou, and S. V. Adve. Managing Energy-Performance Tradeoffs for Multithreaded Applications on Multiprocessor Architectures. In *Proceedings of the 2007 ACM SIGMETRICS*, June 2007.

[35] H. Sasaki, Y. Ikeda, M. Kondo, and H. Nakamura. An Intra-Task DVFS Technique based on Statistical Analysis of Hardware Events. In *Proc. of the International Conference on Computing Frontiers*, May 2007.

[36] R. Springer, D. K. Lowenthal, B. Rountree, and V. W. Freeh. Minimizing Execution Time in MPI Programs on an Energy-Constrained, Power-Scalable Cluster. In *Proc. of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, March 2006.

[37] A. Varma, B. Ganesh, M. Sen, S. R. Choudhury, L. Srinivasan, and B. L. Jacob. A Control-Theoretic Approach to Dynamic Voltage Scheduling. In *Proc. of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, October 2003.

[38] A. Weissel and F. Bellosa. Process Cruise Control: Event-Driven Clock Scaling for Dynamic Power Management. In *Proc. of the International Conference on Compilers, Architectures and Synthesis of Embedded Systems*, October 2002.

[39] Q. Wu, P. Juang, M. Martonosi, and D. W. Clark. Formal Online Methods for Voltage/Frequency Control in Multiple Clock Domain Microprocessors. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.

[40] Q. Wu, M. Martonosi, D. Clark, V. Reddi, D. Connors, Y. Wu, J. Lee, and D. Brooks. Dynamic Compiler-Driven Control for Microprocessor Energy and Performance. *IEEE Micro*, 26(1), 2006.

[41] Y. Li and B. C. Lee and D. Brooks and Z. Hu and K. Skadron. CMP Design Space Exploration Subject to Physical Constraints. In *Proc. of the IEEE International Symposium on High Performance Computer Architecture*, February 2006.

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