Improving the Efficiency of Parallel Applications on Multithreaded and Multicore Systems

Matthew F. Curtis-Maury

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

Doctor of Philosophy
in
Computer Science and Applications

March 19, 2008
Blacksburg, Virginia

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

Improving the Efficiency of Parallel Applications on Multithreaded and Multicore Systems

Matthew F. Curtis-Maury

ABSTRACT

The scalability of parallel applications executing on multithreaded and multicore multiprocessors is often quite limited due to large degrees of contention over shared resources on these systems. In fact, negative scalability frequently occurs such that a non-negligable performance *loss* is observed through the use of *more* processors and cores. In this dissertation, we present a prediction model for identifying efficient operating points of concurrency in multithreaded scientific applications in terms of both performance as a primary objective and power secondarily. We also present a runtime system that uses live analysis of hardware event rates through the prediction model to optimize applications dynamically. We discuss a dynamic, phase-aware performance prediction model (DPAPP), which combines statistical learning techniques, including multivariate linear regression and artificial neural networks, with runtime analysis of data collected from hardware event counters to locate optimal operating points of concurrency. We find that the scalability model achieves accuracy approaching 95%, sufficiently accurate to identify improved concurrency levels and thread placements from within real parallel scientific applications.

Using DPAPP, we develop a prediction-driven runtime optimization scheme, called ACTOR, which throttles concurrency so that power consumption can be reduced and performance can be set at the knee of the scalability curve of each parallel execution phase in an application. ACTOR successfully identifies and exploits program phases where limited scalability results in a performance loss through the use of more processing elements, providing simultaneous reductions in execution time by 5%–18% and power consumption by 0%–11% across a variety of parallel applications and architectures. Further, we extend DPAPP and ACTOR to include support for runtime adaptation of DVFS, allowing for the synergistic exploitation of concurrency throttling and DVFS from within a single, autonomically-acting library, providing improved energy-efficiency compared to either approach in isolation.

Dedicated to my amazing wife Jen, who is always there for me with love and support.

ACKNOWLEDGEMENTS

There are many people who have contributed to this dissertation in one way or another, directly or indirectly. First and foremost, I would like to thank my wonderful wife, Jen, for her continued support. Without her I probably would still have finished this dissertation, but I would not have had such a great life along the way. I can't thank her enough for putting up with me during graduate school. I also owe a great deal to my parents, Michael and Debbie, for their encouragement and love. I would also like to acknowledge other family members who have been there for me in their own ways: Lucien and Jean Maury; David, Pattie, Vicki, and Bobby Mintz; and Jackie and George Wilson.

I am especially thankful to my advisor Professor Dimitrios S. Nikolopoulos for his guidance during my graduate study. Significant portions of this work were based on his ideas and many of the hurdles I have overcome would have been insurmountable without his feedback.

I would also like to thank my Ph.D. committee for their discussions and suggestions for this work: Professor Kirk W. Cameron, Dr. Bronis R. de Supinski, Professor Wu-chun Feng, and Professor Calvin J. Ribbens.

This work has benefitted greatly from input from other members of the PEARL research group, and I would particularly like to thank Filip Blagojevic and Ankur Shah for their considerable efforts related to this project.

I have been very fortunate to collaborate with individuals from Lawrence Livermore National Laboratory and Cornell University, and their work has led to far greater success in this dissertation than would have otherwise been possible. Specifically, I would like to thank Dr. Bronis R. de Supinski (again), Dr. Martin Schulz, Professor Sally A. McKee, and Karan Singh. It has been a real pleasure working with these groups.

Finally, I would like to thank the institutions that have funded this research: the National Science Foundation and the U.S Department of Energy.

Contents

This page intentionally left blank.

List of Figures

This page intentionally left blank.

List of Tables

This page intentionally left blank.

Chapter 1

Introduction

In this chapter, we provide the necessary background for understanding the research performed in this dissertation. Specifically, Section 1.1 presents background on the multithreaded and multicore architectures that we consider in this work. Section 1.2 provides insight into the problems that we are attempting to address on both power-consumption and parallel application scalability. In Section 1.3, we outline the contributions that we make in this dissertation and provide some motivation for their use. Section 1.4 gives an outline of the remainder of this dissertation.

1.1 Overview of Multithreaded and Multicore Architectures

Traditional microprocessor design has emphasized increasing clock rates and greater complexity aimed at exploiting instruction level parallelism (ILP) for improved single-threaded performance. However, due to diminishing returns and reliability concerns, designers have recently redirected their focus in an entirely new direction. Given the limitations of ILP, state-of-the-art microprocessors are beginning to exploit thread level parallelism (TLP) within each chip, in the form of chip multiprocessing (CMP) and simultaneous multithreading (SMT). The ever-expanding transistor budget available to chip designers is now being spent on maximizing the

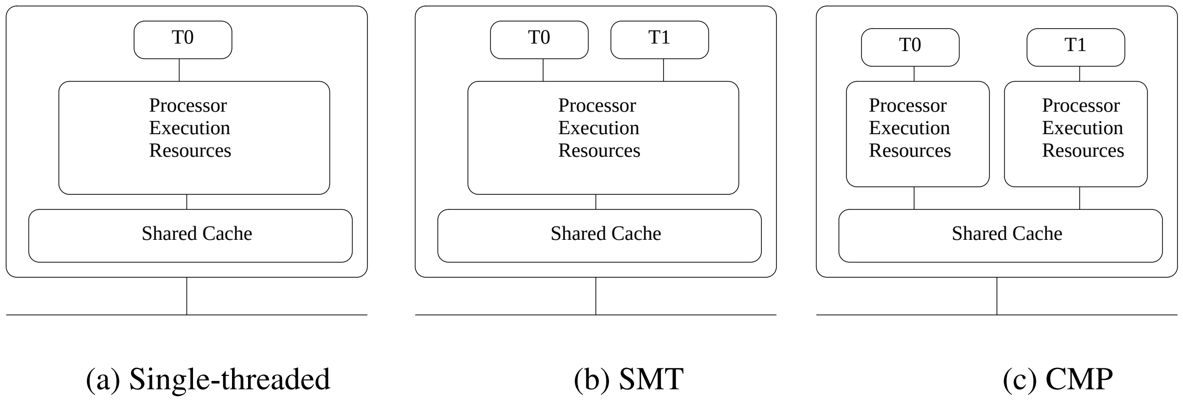
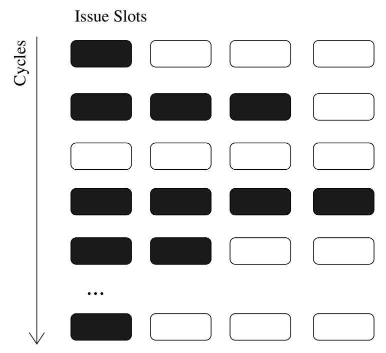

T0
Processor Execution Resources
Shared Cache
(a) Single-threaded
T0
T1
Processor Execution Resources
Shared Cache
(b) SMT
T0
T1
Processor Execution Resources
Processor Execution Resources
Shared Cache
(c) CMP

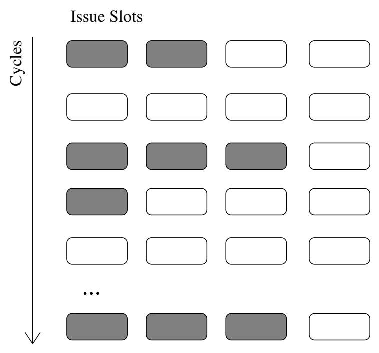
Figure 1.1: Illustration of the differences between single-threaded, SMT, and CMP architectures. T0 and T1 indicate the thread state for two threads.

number of cores per chip, and this new metric is taking the place of CPU frequency for characterizing performance and driving marketability of a computer system. Moore's law is now interpreted as "the number of cores on a microprocessor is expected to double every one to two years", and hardware vendors are racing for the most cores that can be packaged on a single chip [64, 71].

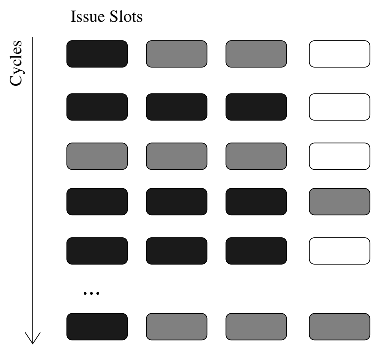
While both multithreaded and multicore systems provide facilities to exploit TLP within an individual processor die, they exist at opposite ends of the resource sharing spectrum. On the one hand, SMT architectures tend towards almost complete resource sharing between co-executing threads as shown in Figure 1.1(b) compared to a traditional single-threaded processor shown in Figure 1.1(a). Specifically, thread state (in the form of registers) is the only resource that must be duplicated [113]. The motivation for SMT is that both "vertical waste," cycles left empty due to insufficient TLP, and "horizontal waste," empty issue slots, can be addressed by finding independent instructions from different instruction streams, or threads [113]. Figure 1.2 illustrates the problems of horizontal and vertical waste using single-threaded processors, and how SMT can improve resource utilization. In SMT architectures, TLP is "converted" into ILP [88] to fully exploit available resources each cycle. SMT targets increased processor utilization and throughput, at the possible expense of individual application performance, in that threads now compete over resources. Many real world implementations of SMT are in exis-



(a) Thread A



(b) Thread B



(c) A and B using SMT

Figure 1.2: Illustration of the advantages of SMT over single-threaded processors at reducing horizontal and vertical waste.

tence. Specifically, Intel announced its Hyperthreading Technology in 2002 containing two execution contexts [72], in 2004 IBM debuted the two-way SMT Power5 [64], and Sun has released its Niagara processor with an 4-way SMT engine in 2005 [71].

On the other hand, CMPs tend to have near complete resource duplication [73], with the exception of outer levels of caches, which tend to be shared between cores as shown in Figure 1.1(c). On these architectures, multiple complete computing cores are placed on a single die. CMPs are the dominant approach to maintaining the long-running trend towards increasing performance, as higher performance potential is present through exploiting TLP than with traditional ILP mechanisms. In essence, transistors are being redistributed from the traditional focus on processor frequency, as multiple simpler cores are used. Examples of CMP implementations abound, with Intel's Core 2 Quad quad-core and AMD's dual-core Opteron. Furthermore, many-core systems are beginning to appear, starting with Intel's 80-core prototype architecture [114] and Rapport's Kilocore [81].

This paradigm shift raises the major challenges of parallelization and scalability to hundreds or thousands of cores on a single processor and even more across multiple chips, two issues that remain largely unresolved, even among the research community that has been investigating them for several decades. In the new landscape of highly parallel microprocessors and system architectures, system software appears to be largely unprepared for the transition that is already underway. Further, the programming process required to parallelize and to optimize code for these architectures remains arduous.

1.2 Problem Summary

There are many difficulties related to high performance execution on emerging parallel architectures such as SMTs and CMPs. In this section, we discuss two of the major challenges: scalability and energy-efficiency. These problems motivate the work performed in this dissertation.

1.2.1 Scalability Limitations

The performance of many applications fails to scale with the number of processing elements used. That is, the performance does not continue to increase proportionally with the number of processing elements used to execute the application. Diminishing returns from increased concurrency and parallelism are a well known property in a wide variety of parallel applications. Further, this situation is a problem on both high-end servers as well as standard desktop computers given the availability of multicore and multithreaded architectures that are bringing parallelism to the mainstream.

Even worse, there are applications, as well as phases of applications, where inherent program characteristics – such as limited algorithmic concurrency, fine computational granularity, and frequent synchronization – and architectural properties – such as capacity limitations of shared resources – limit the scalability and the maximum degree of exploitable concurrency in an application, resulting in an observed performance *loss* through the use of *more* parallelism. This surprising situation where increasing the number of processing elements used to execute an application causes the execution time to increase, is actually quite common.

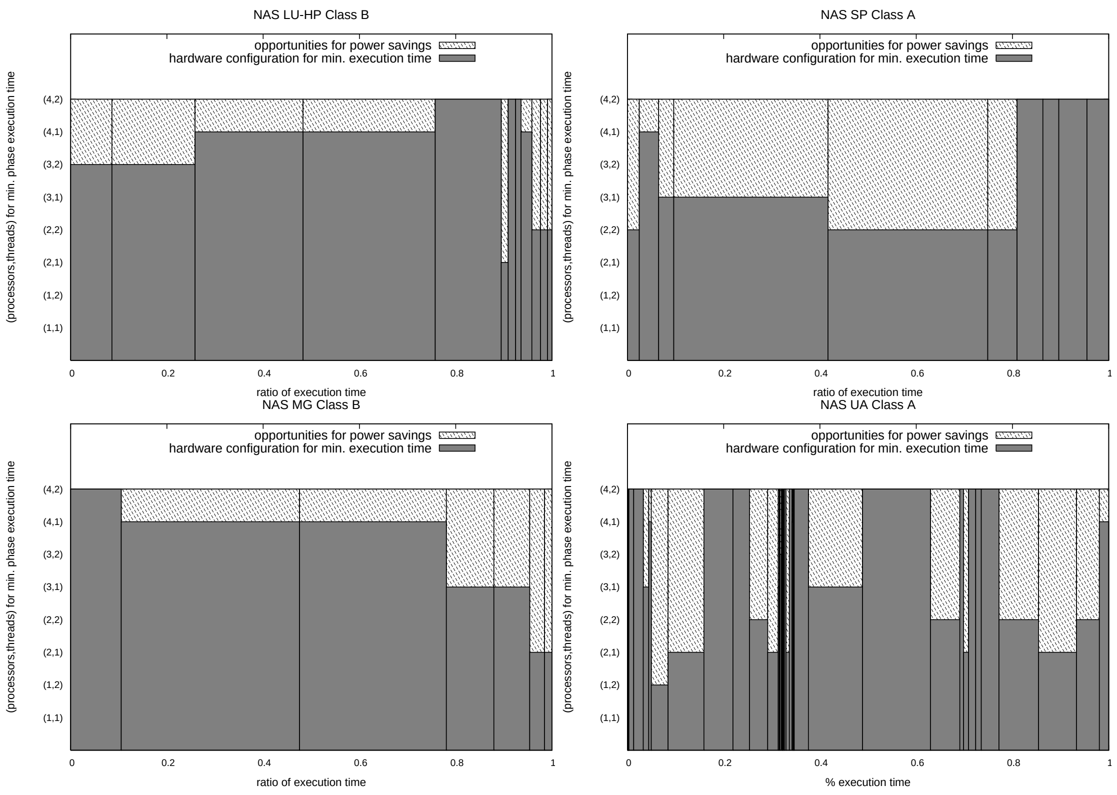


Figure 1.3: Execution time breakdown of four NPB applications on a four-processor server with Intel Hyperthreading processors. Each rectangle represents the configuration that results in the lowest execution time for each application phase.

As examples of the limited exploitable parallelism present in many applications, Figure 1.3 shows a breakdown of the parallel execution time of four applications from the NAS Benchmarks Suite [58] into phases. The breakdowns were obtained during execution of the benchmarks on a quad-processor server with Intel Xeon processors using Hyperthreading technology. Each chart depicts the $(processors, Hyperthreads/processor)$ configuration that minimizes the execution time of each phase through the height of each rectangle. The fastest configuration is identified experimentally, by executing each target phase in all possible hardware configurations of the system. LU-HP-B, SP-A, MG-B, and UA-A execute optimally with at least one Hyperthread per processor deactivated, thus saving power while simultaneously improving performance, during 60%, 73%, 79%, and 33% of their parallel execution times respectively. The same applications execute with at least one entire processor deactivated during 40%, 81%, 18%, and 59% of their execution time. Applications are actually losing performance using the default execution behavior of using all available execution contexts.

Despite the fact that performance often fails to scale with the number of processors, power consumption by the processors does. That is, each additional processor used by an application consumes a considerable additional quantity of power. As already stated, this results in more electrical power being drawn and more heat being dissipated by the processors. These problems together often create a situation where the use of all available processing elements in a multiprocessor is both *hurting* performance in many cases and *wasting* power, because the power is being consumed with no gains in performance.

This situation creates the possibility for simultaneously addressing two of the most important aspects of high-performance computing – performance and power consumption – with a single tool: *concurrency*. In the next chapter we present our proposed solutions to the issues brought forth so far, that exploit limited scalability during application phases to reduce concurrency, thereby improving performance and decreasing power consumption through the deactivation of processing elements. Additionally, we present a hybrid approach that throttles both concurrency and processor voltage/frequency to improve energy-efficiency.

1.2.2 Energy-Efficiency Problems

As part of the continued push for higher performance, modern high-end computing systems are consuming an ever-increasing quantity of electrical power. Further, the last few percent increase in performance tends to come with a disproportionately high amount of the overall power consumption [6, 39]. The level of power consumption has reached the point where it is a serious problem that must be addressed in both research and industrial communities. The problems include the cost to supply electricity to run these systems, as well as many problems relating to the subsequent heat dissipation resulting from current and future levels of power consumption.

It has been noted that at the present trend in power increase, the cost of supplying electrical power will quickly become as large as the initial purchase price of the computing system [6]. A large fraction of this cost, often cited as around 50%, comes from the cost to keep the temperature at acceptable levels in the computer room. All told, the electrical costs of large-scale server farms can reach into the hundreds of millions of dollars each year. Estimates have been made that the yearly power consumption of United States data centers is around 25 gigawatts, enough to require tens of power plants dedicated to converting fossil fuels to electric power just to supply the electricity. From a financial perspective alone, these figures mandate that attempts be made to reduce power consumption. Furthermore, the environmental impacts of such high quantities of fossil fuel being converted to energy are likely to be non-negligible.

Beyond the cost of supplying electrical power to high-end computing systems, other problems surface from the current trends in power consumption. As the amount of power consumed increases, so too does the heat that is dissipated by these systems. This results in a situation where temperatures are steadily climbing in the machine room, routinely reaching thresholds above acceptable operating levels. As captured in Arrhenius's equation for microelectronics, the failure rate of a system doubles for every increase of 10°C (18°F) in operating temperature [51]. Increased failure rates have two substantial impacts on an organization. First, failed parts must be replaced at some cost to the organization. Second, system failures cause outages that can result in idle compute systems and a loss of productivity. In these ways, heat production again becomes a financial issue that must be addressed.

Hsu, et al. [51] note that they have observed silent errors when operating temperatures increase. Specifically, they report Linpack benchmark results outside the residual for a cluster running in an 85°F room, while achieving correct results at 65°F. Errors occurring transparently to users or system operators are a frightening state of affairs. The utility of data that cannot be trusted to a high degree is quite low for many high-end computing users. This issue shows one non-financial repercussion of increasing power consumption, that could affect all computer users.

Power dissipation is now a major consideration for system software optimization on parallel architectures [1, 36, 38, 41]. The introduction of many simple cores on a microprocessor has been largely motivated by the poor energy-efficiency of microarchitectural components that attempt to improve performance at the cost of hardware complexity and reliability [8]. It has been shown that multicore processors have significantly improved energy-efficiency compared to single-threaded processors, achieving significant speedups within the same power envelope [82]. These results are made possible through the elimination of power-hungry, yet limitedly beneficial, ILP-targeted optimizations, and a redirected focus on additional cores with lower frequency and voltage [83].

Given the numerous problems that arose due to the ever-increasing power consumption of modern high-end computing systems, many researchers have begun exploring potential avenues for its reduction at all layers of the computer. Computer architects are investigating alternative designs to reduce the power consumption of processor components, including the register file [70], cache [2, 44], and branch prediction [97]. Disk power consumption is being targeted, largely at the server class of machines, where the cumulative disk power amounts to a large fraction of the total system power [47]. Memory is also being considered for power optimization in the way that pages are allocated to facilitate the deactivation of memory banks [75, 110].

One technique for power reduction that has seen an especially large amount of research is dynamic frequency and voltage scaling (DVFS). Many different heuristics have been developed for determining when to apply DVFS and by how much. In the bulk of this research, system software identifies opportunities where the processor frequency can be scaled down with only limited increases in the execution time of an application [10, 38, 86].

While there are many situations where it is desirable to trade performance for reduced power consumption, in the domain of high-performance computing, performance remains the primary target and energy may be overlooked by end-users. Applications written for high-end computing systems create a challenge for energy-aware system software, which needs to identify opportunities to reduce power consumption with a non-negative impact on performance. In well-tuned, heavily optimized scientific applications, idle periods or long memory latencies, two known opportunities for performance-aware power reduction via DVFS [66, 87], may not arise as frequently, or not be as long as needed to enable substantial power reduction. Programmers will usually do their best to eliminate idling and minimize memory access latency via load balancing and extensive data caching optimizations respectively.

1.3 Summary of Contributions

In this dissertation, we have performed research on many techniques that can be used to understand and improve the performance and energy-efficiency of multithreaded and multicore systems. In each of the following subsections, we discuss particular research contributions made in the completion of this work. Each contirubtion will be presented with more detail in subsequent chapters of this dissertation.

1.3.1 Performance Analysis of Multithreaded Systems

As discussed earlier in this chapter, modern systems are moving towards increased parallelism in the form of multithreaded and multicore architectures. It is necessary to understand the performance of parallel scientific applications executing on these architectures because they are being adopted so quickly. We utilize hardware event counters collected during parallel application execution to perform a detailed study of the performance of multithreaded (SMT) architectures. These counters provide considerable input into the architectural bottlenecks, as they grant insight into the hardware/software interaction.

Utilizing multiple threads per processor can have complicated effects on both performance as well as the utilization of internal processor resources. Given the high degree of resource sharing on SMT architectures, multithreading can cause considerable negative interference between co-executing threads, resulting in severely limited performance gains and even performance losses. Our analysis can steer the design of future multithreaded architectures as well as the optimization of parallel applications for execution on these systems.

We additionally investigate alternative forms of multithreaded execution, both previously proposed and new ones, in an effort to overcome the poor performance of SMT processors at executing scientific applications. We consider adaptive concurrency throttling on each multithreaded processor and speculative precomputation (SPR). Although these techniques have been used in earlier work as tools to enhance SMT performance, they have been explored in different contexts. Most importantly, effective integration of these techniques with regular thread-level parallel execution in the same code has not been explored before. We propose methods to integrate adaptive throttling and speculative precomputation with traditional thread-level parallelization using the OpenMP programming paradigm and we show that combining these techniques results in significant performance improvements.

1.3.2 Dynamic Concurrency Throttling

Conventional wisdom holds that when concurrency is increased, performance will be improved, but with an associated increase in power consumption. Conversely, when concurrency is decreased, power consumption will be reduced, at a cost for performance. In this way, power and performance can be balanced, trading one for the other depending upon the goals of a given user. However, in situations where additional concurrency results in a performance *loss*, decreasing concurrency will actually improve performance and power consumption simultaneously. This reduction and increase of concurrency is known as *concurrency throttling*.

Concurrency throttling is primarily a technique for improving performance, as performance-optimal configurations are sought for each phase. However, concurrency throttling also stands to considerably reduce power consumption by transitioning processors and/or cores to lower power states. As a power reduction technique, concurrency throttling has significant advantages over existing software-level optimizations, including adaptive DVFS scheduling. Concurrency throttling can be applied to well-tuned, compute-intensive phases of an application, that otherwise provide limited opportunities for power optimization through known approaches. Further, the vast majority of power-aware execution strategies sacrifice performance to achieve power savings. Many existing techniques are, in fact, able to locate opportunities to save power while only increasing execution time by a few percent, however in high-performance computing (HPC), any performance loss is generally unacceptable. Concurrency throttling, on the other hand, throttles concurrency only where an associated performance gain is expected. Therefore, this technique is highly appropriate in the HPC domain, as performance is not sacrificed, and will generally be improved. As a result of the simultaneous reductions of both execution time and power, it achieves a multiplicative reduction in the total expended energy.

Despite its appeal, concurrency throttling is an opportunity which may present itself to varying degrees across different programs, across different phases of the same program, or even across different inputs fed to the same program. Identifying concurrency throttling opportunities statically is hard, because it requires fine-grain analysis of the dynamic behavior of parallel code across and within parallel execution phases. Besides the problem of identification and quantification of the opportunities, applying concurrency throttling directly in applications requires exposure of the programmer to architectural details, such as the number and physical layout of processors. This tactic is widely considered as one of the factors that make parallel programming exceptionally difficult [4]. Given the complexity along with the inherent drawbacks of delegating concurrency throttling decisions to the user or to a static analysis tool, runtime systems appear to be ideal candidates for the identification and exploitation of concurrency throttling opportunities.

To address the problems of limited scalability and high power consumption in parallel applications, we have developed a runtime system aimed at improving performance and reducing power consumption in applications in high-performance computing environments. The system, which we call *ACTOR*, or Adaptive Concurrency Throttling Optimization Runtime system, exploits the scalability variance of applications on current architectures to steer the degree of concurrency, thereby simultaneously reducing execution time and power consumption. ACTOR seeks the optimal operating point of concurrency in multithreaded programs, at the granularity of program phases.

1.3.3 Dynamic Phase-Aware Performance Prediction

Within ACTOR, identification of the optimal operating points of concurrency for different phases of an application at runtime can be performed in a variety of ways. One set of approaches is based on directly searching the possible optimization space at runtime, testing various possibilities for small periods of time and selecting the one that results in the best performance to be used for the remainder of application execution. In our preliminary research, we have evaluated two such approaches based on an exhaustive search and a heuristic search.

In addition to concurrency throttling schemes based on live empirical search of operating points, we have extended ACTOR to rely on a dynamic, phase-aware performance prediction (DPAPP) model. The model predicts the optimal operating point of concurrency on different configurations of processors, cores, and threads, here on referred to as *hardware configurations*, or more simply *configurations*. The key contribution of the DPAPP model is that it enables a drastic reduction of the overhead associated with searching the optimization space for concurrency throttling. The small number of test iterations needed with performance prediction can greatly cut down on the overhead of adaptation and make runtime adaptation via dynamic prediction of optimal concurrency feasible. However, prediction-based approaches do require accurate predictions of the performance on varying hardware configurations to be successful.

The prediction methodology that we have developed uses live input from hardware event counters, collected while executing program phases on operating points of maximal concurrency. We use a multivariate linear regression process for selecting the critical hardware events that best predict performance, and for training the DPAPP model in assessing the scaling effects that changing hardware configurations have on overall program throughput. The DPAPP training process derives distinct predictors for thread-level, core-level, and processor-level parallelism, to account for the presence of multidimensional parallelism and variance in the impact of sharing of resources between parallel processing units within and across chip boundaries. We use the DPAPP model to steer a runtime concurrency throttling system, which succeeds in identifying phases where power consumption can be reduced while sustaining or improving performance.

The DPAPP model provides functionality that has not previously been available. Specifically, based on a small number of sample points in the execution of an application, its performance on other configurations can be predicted. While we have found the prediction model to be highly effective in the context of concurrency throttling, as we will show in this dissertation, other uses of this new form of performance prediction may arise as well. We believe that the model will prove useful to a wide variety of other strategies for power and/or performance optimization. We anticipate that the scalability predictor will be an important tool, useful to a large group of researchers and system developers.

1.3.4 Integrating DVFS with DCT

The central theme of this dissertation is the development of a complete runtime system for the autonomic adaptation of parallel applications to improve performance and energy-efficiency. The technique that we have discussed thus far, concurrency throttling, is very successful at improving performance and energy-efficiency, however we have also extended the runtime system in another direction as well. Despite the performance losses that are generally associated with applying DVFS, it remains a useful tool for power-aware computing. Further, its use is orthogonal to concurrency throttling. As an additional step in the development of a energy-aware runtime system, applicable in the context of high-performance computing, we have integrated concurrency throttling with dynamic frequency and voltage scaling.

In order to synergistically apply both DCT and DVFS at once, we have extended the DPAPP model to support prediction of the effects of applying varying levels of DVFS. Further, a hybrid model allows for the simultaneous prediction of applying multiple power saving *knobs* in combination, to determine the combined effects. This hybrid model facilitates adapting along both dimensions, rather than making decisions for each dimension independently. By considering the interaction of the two techniques, we are able to improve performance and energy-efficiency beyond that achieved by applying each approach sequentially or in isolation.

1.3.5 Considering ANNs for Model Training

While the use of multiple linear regression provided high accuracy in scalability prediction, there are other forms of model formulation that have certain advantages and disadvantages. A second well known form of model training is artificial neural networks (ANNs). ANNs can be used with the same model we have already evaluated with linear regression, they simply provide an alternative mechanism for learning the relationship between observed event rates and scalability.

Specifically, ANNs have the advantage that they require less work on the part of the modeler in the model specification process. That is, while regression learns the relationship only between user-specified predictor and response variables in the model, ANNs additionally consider interactions between all terms, providing higher potential accuracy where many interactions exist, without requiring manual work on the part of the modeler in finding and specifying extant interactions. Additionally, ANNs are able to capture non-linear relationships with ease, which provides an advantage in circumstances where the relationship is in fact non-linear.

For these reasons, we evaluate both the absolute success of ANNs for scalability prediction and for use with prediction-based concurrency throttling. Further, we compare the resulting models in terms of accuracy, suitability for use with concurrency throttling, and ease of use with those derived using linear regression.

1.4 Organization of this Dissertation

The rest of this dissertation is organized as follows:

In Chapter 2, we present an overview of related work and present the context for the research performed in this dissertation. Specifically, we break the related work into five areas from which the current work draws and contrast it with this dissertation.

In Chapter 3, we perform a performance evaluation of parallel scientific applications executing on multithreaded architectures. Specifically, we present an analysis based on the collection of hardware event rates on each architecture. This work provides an important background for later work on analyzing the scalability that results from parallel execution on these architectures.

In Chapter 4, we present a novel scalability model (DPAPP) of parallel applications executing on systems composed of multiple multithreaded and multicore processors. The model is based on the statistical analysis (multiple linear regression) of hardware event counters collected during brief sample executions at maximal concurrency and makes predictions at phase granularity in parallel applications.

In Chapter 5, we discuss an approach to dynamic concurrency throttling (DCT) whereby optimal concurrency levels are determined through the use of the DPAPP scalability predictor. We also show the advantages of using performance prediction over simpler empirical search based approaches, particularly as emerging systems continue to increase in on-chip thread-level parallelism.

In Chapter 6, we present our approach to extending both the DPAPP model as well as the ACTOR runtime system to support simultaneous adaptation of both DVFS and DCT from within a single unified library. Such adaptation provides considerable potential for performance and energy savings relative to either approach alone, and we present a detailed study of the benefits.

In Chapter 7, we present an overview of artificial neural networks (ANNs) as well as the results of DCT adaptation using an ANN-based model. We then discuss the advantages of using ANNs versus multiple linear regression and present the relative performance and energy-efficiency results of each approach.

In Chapter 8, we present a brief summary of the research done in this dissertation.

Chapter 2

Background and Literature Survey

As the research performed in this dissertation is not confined to a single narrow topic, there is a wide variety of related work in the literature. In some areas, such as power-aware computing and performance prediction, the background is so deep that it is simply not feasible to present a complete picture of the current state of the art. Here we attempt to present the work that we feel is most closely related to our own, with each of the five categories of work given its own section.

2.1 Hardware Event Counters for Adaptation

Much previous research has been performed on optimizing the execution of programs using feedback from hardware event counters (HECs), however it has predominantly been offline, profile-guided in nature. For example, Marathe, et al. [90] have performed NUMA multiprocessor page placement using hardware monitoring assistance. The authors monitor the occurence of two specific events to track accesses to memory pages – long-latency loads and DTLB misses – during an offline test execution of the application. The total accesses to specific pages, both statically and dynamically allocated, are recorded and this information is used to map pages to the processor that had the largest number of accesses, thereby reducing average latency.

Cascaval, et al. [14] present CPO (Continuous Program Optimization), a runtime system that includes management of variable page-size systems where page-size decisions are made by analyzing hardware event counters. Specifically, data access addresses and TLB misses are monitored, along with page faults as seen by the operating system and memory allocation events, during an offline execution to predict the benefit that would be received by utilizing a larger page size. During live executions, the database of stored page information is examined and the currently available number of pages is considered to determine which data structures should be mapped to large pages. Decisions are then monitored by CPO for validation of predicted benefits.

In contrast to these offline optimization techniques, little work has been done on runtime optimization utilizing hardware event counters as the program executes. One existing example is HEC-based SMT job schedulers by Settle, et al. [100]. In this work, the authors extend the hardware to provide architectural support for monitoring of each thread's cache access patterns. When scheduling decisions are to be made, the operating system can compare the access patterns to predict the degree of inter-thread interference in the cache. Competition between two co-executing threads on an SMT processor results in potential performance loss due to contention in the shared cache. Cache miss rates are reduced because contention between threads can be predicted and often avoided through intelligent scheduling.

In the ADORE runtime optimization system [89], Lu et al. dynamically detect and react to performance problems that are difficult to optimize statically at compile time, such as cache misses and branch misprediction patterns. Optimizations including register allocation, cache prefetching, and code layout are applied dynamically at the granularity of program phases based on recorded HEC values. When an optimization is to be applied, the alternative, optimized code is created and the execution is redirected to the new code when a given block is re-executed. Optimizations can be executed speculatively to determine their effectiveness. Rather than simply applying their approach whenever possible, they limit overhead by selectively applying to hot regions that are executed with high frequency.

2.2 Performance Prediction

The field of performance prediction is quite mature, and we will focus our attention on that work most relevant to our model of prediction. Performance prediction of parallel programs has been studied in great depth, however the majority of research is targeted at offline prediction. The work most closely related to ours is that of performance prediction used for microarchitectural exploration, where the performance of different configurations is predicted through the analysis of quantitative architectural properties, such as reorder buffer size and cache size, as well as application characteristics. The similarities arise from the fact that we are also predicting the effects of architectural changes – specifically the number of processors and cores – however, we perform our prediction online during program execution.

Minimizing design space evaluation time for processor development has spurred much research on predicting the performance effects of altering various microarchitectural parameters, because of the considerable design-time reduction that it can provide. Lee, et al. [77, 78] employ a regression-based prediction strategy with inputs of architectural and application-specific properties. Their goal is to reduce required simulation time by limiting the number of required simulations, predicting for un-simulated configurations based on statistical inference of the effects of varying architectural parameters. While they also utilize regression, they have extended the techinque with many statistical optimizations to improve the resulting prediction accuracy. Beyond their regression model for performance prediction, Lee, et al. [79] also evaluate possible configuration sampling methodologies to maximize the possible statistical inference from a limited set of samples (i.e. executed architectural simulations).

Similar work has also been done to predict the performance of a given configuration through the analysis of event rates that it produces in executing an application [76, 95]. In effect, the degrees to which each event determines performance are modeled to produce a linear regression formula of performance. This is useful in architectural analysis, however exploratory value is limited since performance could simply be observed while recording events, on either a simulator or a real system. Further, despite the fact that the events are directly collected on the desired configuration, prediction accuracy is fairly low. In fact, our prediction model has higher accuracy than either approach, even though we are predicting across configurations. This may in part be attributed to the fact that we collect performance on the observed configuration, which has been shown to be the primary determinant of performance across other configurations [77].

Architectural exploration research has been performed using statistical methods, without the use of performance predictions as well. Joseph, et al. [61] present a linear regression based method of identifying key architectural parameters. Their regression model is based on cycle-accurate simulation and is applied iteratively to meet some specified error bound. Specifically, they predict which of the 26 micro-architectural parameters that can be varied in SimpleScalar have the most impact on CPI. However, their model does not make predictions of performance, but rather just provides a significance ordering. Similar work is performed by Yi, et al. [120] that uses the Plackett-Burman method to rank the significance of parameters in simulation.

In addition to work on performance prediction based on regression analysis, much research has been done to exploit machine learning through artificial neural networks (ANNs) in this context. Singh, et al. [105] predict the performance of two different HPC applications whose performance variations are poorly understood on a given architecture with varying data size inputs and number of processors using multilayer neural networks. Training is performed by specifying the input size and the observed execution time to create a function of input size to predict future queries on input size. Their approach is able to predict the performance of the applications with an average error as low as 2%, however all training is per application and cannot therefore be applied at runtime to new applications. Additionally, neural networks have been used to predict the effects of compiler optimizations to restrict the necessary simulation space when compilers are being developed for new architectures [15].

Neural network based performance prediction has been done for architectural exploration as well. Both Joseph, et al. [60] and Ipek, et al. [54] have evaluated similar nonlinear models for predicting the performance of an application subject to many variations in the architectural parameter space. Each approach suggests an improved method of configuration sampling for simulation over simply picking uniformly at random. The main difference appears to be that the work of Joseph, et al. requires considerably fewer training samples, which further limits the cost to use, while achieving slightly higher accuracy.

Lee, et al. [80] compare the effectiveness of piecewise polynomial regression and ANNs for predicting performance in the context of varying input parameters. Their findings suggest that prediction accuracies between the two approaches are comparable, but each approach is advantageous in different contexts. However, they report that the training process is significantly simplified through the use of ANNs. While there is overlap between Lee, et al. [80] and our contribution, we evaluate performance predictions in a very different context. Specifically, we consider predictors for online use in the prediction of parallel application scalability. Further, the linear regression model we evaluate reduces the end-user burden during model specification while still resulting in high accuracy.

Many application-specific performance models have been created as well. As just one example out of many, Kerbyson, et al. [68] develop a model to predict the performance of a multidimensional hydrodynamics code from ASCI called SAGE on different numbers of processing elements. One interesting property of their model is that architectural parameters can be specified to derive a prediction for any given architecture, rather than creating a new model. Such models aid in the analysis of existing architectures for the improvement of next generation systems. Further, they can be used to locate optimization opportunities within an application.

Yang et al. [119] develop cross-platform performance translation based on relative performance between the target platforms, and they do so without program modeling, code analysis, or architectural simulation. Like ours, their method targets performance prediction for resource usage estimation. They observe relative performance through partial execution of two applications; the approach works well for iterative parallel codes that behave predictably (achieving prediction errors of 2% or lower) and enjoys low overhead costs. Prediction error varies much more widely (from 5% to 37%) for applications with variable overhead per timestep. Likewise, reusing partial execution results for different problem sizes and degrees of parallelization renders their model less accurate.

Carrington et al. [11] demonstrate a framework for predicting performance of scientific applications on LINPACK and an ocean modeling application. Their automated approach relies on a convolution method representing a computational mapping of an application signature onto a machine profile. Simple benchmark probes create machine profiles, and a separate tool generates application signatures. Extending the convolution method allows them to model full-scale HPC applications [12, 106]. They require generating several traces, but deliver predictions with error rates between 4.6% and 8.4%, depending on the sampling rates of the underlying traces. Using full traces obviously performs best, but such trace generation can slow application execution by almost three orders of magnitude. Some applications demonstrate better predictability than others, and for these trace reduction techniques work well: prediction errors range from 0.1 to 8.7% on different platforms.

Marin and Mellor-Crummey [91] present a toolkit to measure and to model application characteristics semi-automatically in an architecture-neutral way. They predict application runtime using properties of the architecture, the binary, and the application inputs, and evaluate their predictions against measurements collected using hardware performance counters.

Our model for performance prediction is unique in several ways. The approach works at runtime during the execution of an application and this property limits the computational overhead associated with prediction. To our knowledge, no prior work has considered online predictors of parallel execution performance on shared-memory architectures, using runtime input on IPC and hardware event counts. Further our model is more sophisticated in that we account for the effects of events in a multiplicative way on the baseline IPC of the sample configuration, rather than an additive way that is often assumed. This means that events are modeled to effect the resulting IPC by some percentage rather than by a fixed amount.

2.3 Power-Aware, High Performance Computing

Power-aware, high-performance computing has recently become an important topic of research. The realization that current trends in power consumption will lead to limited performance gains resulting from unmanageable heat dissipation and increased power cost has led the high-performance computing community to undertake considerable work to reduce the problem. Efforts in this area range from power-scalable and power-efficient clusters [1, 36] to runtime systems providing support for dynamic frequency and voltage scaling for parallel applications [38, 66]. Our work is most closely related to the latter, which attempts to identify opportunities at runtime to achieve power savings without sacrificing performance. As such, we focus on runtime power savings techniques in this section, ignoring the extensive work on architecture-level approaches [2, 44, 70, 97]. Memory [75, 110] and disk [47] power optimizations are available as well, which are beyond the scope of the related work presented here.

The vast majority of software-level energy-aware optimizations involve the use of DVFS to reduce processor power consumption at the expense of compute speed. Each technique searches for opportunities to enable different levels of DVFS without sacrificing performance by too large of an amount [39, 50]. In general, there are three categories of approaches to exploit limited CPU-boundedness in high-performance computing applications studied in the literature. First, most work has been performed on quantifying memory boundedness, and applying DVFS during phases where the program is heavily accessing memory. This works under the experimentally verified assumption that DVFS will have less impact on performance during memory intensive regions [17,38,51,117,118]. Second, several researchers have explored opportunities to exploit *slack* between processes to enable DVFS, either slowing down the shorter job [66] or enabling DVFS once a subset of jobs have reached the barrier [87]. Third, periods of (MPI) communication between cluster nodes can be executed with DVFS [86], because the CPU is not the bottleneck for these phases. Additionally, some research has considered combinations of the above metrics in applying DVFS [39].

Hsu, et al. [51] model the effects of DVFS on performance as measured in MIPS, using regression to derive an application-specific coefficient for the observed performance loss using a lower DVFS level, which can then be used to predict the performance of other DVFS levels. The authors select the lowest DVFS level that will maintain some maximum performance loss, thereby maximizing the energy savings under the given constraints. Very similarly to their work, we wish to use regression to model the performance impacts of applying various degrees of DVFS, however we use multiple events in the prediction model rather than performance alone.

Ge, et al. [42] compare three strategies for identification of the optimal DVFS level. Specifically, evaluations are done between CPUSPEED, which is a system monitor available with Fedora Core that selects the DVFS level based on past CPU usage; command line control based on a user-estimated DVFS level; and calls to a DVFS library to set the level from within the application. Over a variety of benchmarks and microbenchmarks, the authors find that all approaches are able to save considerable energy, with program internal control performing the best after extensive optimization.

A similar analysis is performed by Isci, et al. [55] who further find that to meet a given power budget on a CMP, DVFS must be applied adaptively based on changes in execution properties to maximize energy savings and performance, rather than statically. Further, the authors present an approach to scale DVFS per core autonomically to meet the energy limits for multiprogramming workloads. To optimize throughput and fairness, they utilize performance predictions at different DVFS levels, however they apply the same static scaling of performance at different DVFS levels regardless of program properties (e.g. memory boundedness), leading to errors in the predicted performance that we improve upon through the use of a wide variety of application characteristics in our prediction.

DVFS has been combined with the deactivation of nodes within a cluster to reduce the energy consumption while still meeting a specified energy budget [107]. The authors use regression based prediction of the performance at different systems schedules of DVFS level and number of active nodes used to execute the parallel application. Once predictions are made for all schedules, the single schedule that minimizes execution time while meeting the energy limit is selected for use. While their approach is very effective, there are a few drawbacks that our work targets as improvements. First, Springer, et al. [107] require a relatively large number of sample executions to train the predictor for the given application. In contrast, our approach works by training offline with other applications and applying observed trends to a very small number of online samples. Specifically, the authors require a training point for each DVFS level, whereas we propose the use of prediction to determine the performance effects at each level. Second, in our proposed work we target the minimization of execution time, rather than meeting some user-specified energy limit which certainly requires application-specific knowledge. Thus we simply require the user to decide what metric is most important without requiring specific *requests* for energy consumption. There are additional differences in context that differentiate our work from theirs. Most importantly, the authors target clusters where it is known that applications scale well, and thus they do not attempt to *improve* performance. On the other hand, we exploit limited scalability on SMPs to improve both the power and performance of an application. Finally, there is no reason why the two approaches could not be applied sequentially, to determine the optimal number of processors to use per node in a cluster, after their approach has been applied.

Kappiah, et al. [66] have explored the potential for applying DVFS in the presence of load imbalance in parallel programs executed on power-scalable clusters. The primary insight behind their work is that when load imbalance occurs, one or more processes will sit waiting for the others to reach a synchronization point. The authors thus monitor synchronization points in MPI (i.e. MPI communication library calls) to determine which node is the bottleneck at each communication point across iterations of the application. Each non-bottleneck node is then set to the DVFS level that will cause it to reach the synchronization point "just-in-time" for the others, thereby reducing wasted energy compared to the default strategy of executing all nodes at full speed even when doing so will not improve performance. One interesting similarity between their work and ours is that they also exploit knowledge gained by observing inter-iteration patterns.

In addition to memory intensive phases, communication phases are not CPU intensive. As such, DVFS can be applied without significantly impacting the performance of the application. Li, et al. [86], first identify groups of MPI calls that are temporally close enough to be treated as a phase, and then identify the best DVFS level at this granularity. The entry point to each identified phase is kept internally by the call stack and the program counter upon entering an MPI library call. In iterative codes, such as the ones we also consider, the same call site will occur many times with similar execution properties between invocations. Therefore, the authors predict the best DVFS level to use based on retired micro-ops/microsecond, as an indicator of CPU intensity, and monitor execution time between executions. Further, the approach is embedded within the MPI library by hijacking MPI calls, so no user-intervention is required.

A detailed analysis of the energy properties of scientific codes has been performed as well [37]. This study is useful for better understanding the potential of various energy optimizations as breakdowns are provided for each system component at idle and loaded times, with the most obvious trend being that the CPU consumes much more power than the other components unless the application is disk-bound. Feng, et al. further analyze the energy effects of scaling the number of nodes in a cluster and find that while energy continues to increase, performance does not, and they present an approach that could be used to trade a small amount of performance for significant energy savings in these cases. Finally, the paper shows that instantaneous power profiles are both dynamic within a given iteration and repetitive across iterations.

Prediction of phases has been applied to exploit DVFS proactively as well [56]. Specifically, Isci, et al. use technology similar to branch prediction to maintain a phase history table that tracks recent phases to predict which phase is likely to be encountered next. Using these highly accurate predictions as input, the authors are able to apply DVFS proactively so that the right setting will be used when the phase is executed. This approach prevents delays in DVFS application that occur with reactive approaches. Similarly, our approach is proactive, in that we use knowledge of when a given phase will begin execution based on calls inserted with minimal programmer effort into the source code.

Sasaki et al. [99] 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. The model they present is very similar to our prediction model for both concurrency throttling and DVFS, however our model can be used to predict the collective effects of applying both techniques (see Section 6) and our scalability model was published well in advance of their work on DVFS. We believe that our comparison of multiple linear regression and ANNs presented in this work can be used in systems such as those discussed by Sasaki, et al. [99] to improve the model development process.

Other work has focused on deriving an analytical model of DVFS [40, 43]. First the authors developed a model of speedup achieved by increasing CPU frequency on processors with DVFS. Their model also predicts the effects of increasing the number of cluster nodes used in conjunction with DVFS changes, though it requires execution samples on all desired numbers of processors and on all processor frequencies during sequential execution. In contrast, the models we present in Section 4 and Section 6 draw few samples from within the two-dimensional configuration space and statistically analyze performance counters, thereby minimizing burden on the modeler. However, their analytical model does provide considerably increased insight into the architecture compared to our empirical model. Ge, et al. have also utilized their analytical model to develop a runtime DVFS adaptation system called CPU MISER [43]. This system takes input from the user requesting a specific performance threshold that must be maintained while applying DVFS, allowing the system to reduce processor frequency to conserve power as long as the performance requirement is met. On a 16-node cluster, their approach is able to conserve energy by 20%, which they show to compare quite favorably with CPUSPEED, a standard DVFS tool.

Our work further extends upon previous research in that concurrency throttling is a performance optimization, rather than just a power optimization, which is not true of the work presented in this section. This important difference makes our techniques more relevant in the context of high performance computing where performance is the primary target.

2.4 Concurrency Throttling

Concurrency throttling has been previously applied for optimization of multithreaded codes on shared memory multiprocessors. Specifically, concurrency throttling can enable adaptive execution in multiprogramming environments. For example, Anderson, et al. [3] present scheduler activations, where applications schedule their own threads onto processors, and the operating system is responsible for allocating which processors an application can use. In this system, applications notify the kernel when more or fewer processors are needed. Similarly, Tucker, et al. [112] propose a scheme to give the operating system a mechanism to control the number of processors given to each application in the presence of varying multiprogramming degrees, with the application adjusting its concurrency as a result. Yue, et al. [122] have more recently evaluated the potential for concurrency throttling at the boundaries of parallel regions (as we do), when a parallel region is entered during heavy system utilization. Hall, et al. [48] propose an approach to identify limited scalability within phases of an application, which they attribute to poor parallelism and false sharing between processors, with the goal of improving multiprogramming workload throughput. However, in none of these approaches are processors intentionally left idle for performance or power gains, but rather reallocated to other applications in the system to improve system throughput.

Further, standalone programs can benefit from concurrency throttling across different phases with potentially different execution and scalability characteristics. It has been observed many times that SMT as implemented on the Intel Hyperthreaded processors [92], tends to perform poorly for executing parallel scientific applications due to interference both in the cache as well as in the execution resources [22, 62, 123]. To address this issue, Zhang, et al. [123] developed a loop scheduler that decides the number of threads to use per processor based on sample executions of each possibility. The authors extend that work to incorporate decision tree based prediction of the optimal number of threads to use [124]. Rather than deriving performance predictions, they simply predict which number of threads will be faster for the given phase.

Adaptive serialization is explored by Voss, et al. [115], where the authors attempt to identify parallel regions that will scale poorly when parallelized and then execute those regions sequentially. Identification is performed using two different approaches. First, the length of loop is measured and if it is below some threshold, the loop will be serialized because the benefits of parallel execution will be unlikely to outweigh the parallelization overhead. Second, a prediction of sequential performance is made based on observed scalability properties of the machine and offline recorded loop execution times. If the target parallel loop is expected to see performance loss through parallelization, then it is serialized. Our concurrency throttling approach has the advantage that we do not limit possible concurrency levels to one or the number of processors, rather allowing any number of threads in between to be used as well.

Concurrency throttling that combines both of the previous approaches, that is, serialization and SMT-disabling, is presented by Jung, et al. [62]. In this work, the authors present approaches for static and dynamic serialization of parallel regions. Further, they compare two strategies for identification of the optimal number of threads to use per SMT processor. In the first strategy, both configurations are simply tested. In the second, execution properties observed on the one thread per processor run are used to predict the two thread performance. This prediction is made based on regression analysis with L2 misses, L3 misses, and retired instructions, however no accuracy results are presented.

Recent work has considered applying concurrency throttling and DVFS on single chip multiprocessors, with decisions utilizing search algorithms of the configuration space [83]. This research shares many motivations with our work, mainly maintaining performance while reducing power consumption, however the suggested solutions to the problem differ significantly.

First, our approach is implemented on a real system, rather than simulated, verifying that our technique works in practice with all overheads considered. Second, we utilize performance prediction rather than empirical searches of the configuration space to reduce the number of test executions necessary to perform adaptation. Further, we show that the overhead of search based techniques hinders the performance of short-lived codes, particularly when compared to prediction. Third, we attempt to maximize performance in the face of energy reduction rather than attempting to meet some performance goal. Additionally, our approach targets multiprocessor systems where the combined energy consumption of the processors plays a much larger role than in uniprocessor systems such as that evaluated in the related work [83]. Finally, the technique that we present is sufficiently low-overhead to be applied to application with very few iterations, in contrast to the work presented by Li, et al. [83] which requires artificial lengthening of the program to perform adaptation through empirical search.

More recently, other research has investigated the potential of concurrency throttling to improve both performance and power consumption of parallel applications in shared memory environments [109]. In their work, the authors introduce a technique called *Feedback Driven Threading* that seeks the optimal number of threads in the face of limited application scalability. However, they consider only bus contention and frequent synchronization as detrimental to parallel scalability, though they acknowledge that other factors – such as cache contention – are likely also important. In this dissertation, we consider any architectural factors that can be captured using hardware event rates in predicting application scalability. Further, we experiment with *real* systems as opposed to the simulated systems used by Suleman, et al. [109]. It is also important to note that our work predates that of Suleman, et al. by two years.

Programmers have long had the ability to specify concurrency levels manually, however few runtime systems provide the functionality to manage these decisions automatically from within. Our work provides such a system, offering fully automatic concurrency throttling based on performance predictions of each configuration with the goal of simultaneously improving both performance and power consumption.

2.5 Exploiting Phases

Programs have been generally acknowledged to experience considerable fluctuations in program execution, with periods of stasis occurring within regions known as program phases [102, 103]. Within phases, large groups of execution characteristics have been shown to be stable, including cache behavior, branch prediction, and IPC among other things. Not only do they stay largely constant during the time periods, but they also tend to change at the same boundaries. Phases can be exploited by targeting optimizations at observed behavior that is constant during a given phase, rather than optimizing for average behavior over an entire application. Two important aspects of work on phases have been identification of phases within programs and runtime systems that take advantage of known phases to apply optimizations.

Early significant work on program phase identification was performed by Sherwood, et al. [102]. In this work, the authors proposed a scheme called a basic block vector (BBV), which encapsulates multiple architectural metrics, such as IPC and cache miss rates, for different regions of code. Phases can then be determined by comparing the distance between blocks in terms of their respective metric values. That is, blocks experiencing similar execution properties are defined to be as part of a single phase, regardless of the temporal location of code execution. This work is extended to include the prediction of phase transition and the identity of the upcoming phases through the analysis of recently executed phases [104].

Another important work on identification of program phases works through the creation of a working set signature based on the current working set of an application [31]. Intuitively, changes in the working set determine phase boundaries and can thus be identified through the resulting changes in the signature. Further, the authors use the phase identifications to optimize dynamically configurable hardware, such as TLBs and branch predictors. By exploiting knowledge of recurring phases, the total number of reconfigurations required at runtime is reduced by 95%.

In our runtime system, we exploit fairly stable execution properties experienced within parallel regions of OpenMP programs. While this is only a coarse-grained approximation of phases, it is the finest granularity at which the number of threads can be changed at runtime, and is therefore the lowest level at which our system can operate. However, there are opportunities to consider more sophisticated strategies for program phase identification in programs parallelized using other programming models, such as MPI, to determine the potential gains of finer-grained phase-based adaptation.

Chapter 3

Evaluation of Multithreaded Systems

Simultaneous multithreaded processors [113] have shown considerable popularity in both mainstream and high-performance computing markets. These processors were introduced in academic studies [88,113] and later adopted as a core design technology for mainstream processors from Intel, IBM and other vendors [64,111]. SMTs provide an incremental path to improve the performance of conventional superscalar processors by converting thread-level parallelism to instruction-level parallelism, with only marginal increases in cost, area and power consumption.

With the dominance of multithreaded processors clearly visible on the horizon, it is necessary to assess their effectiveness at executing parallel codes. The first contribution of this chapter is an evaluation of parallel application execution on a real SMT-based multiprocessor, consisting of Intel Hyperthreaded processors. We attain detailed performance measurements from hardware event counters and use them to gain insight into parallel application execution on this architecture. One goal of our analysis is to identify architectural bottlenecks of real, commercially available SMT processors, that reduce the scalability of parallel applications. Secondly, we look for software techniques that can be used to provide more efficient execution on such processors.

Scientific applications typically exhibit a high degree of exploitable thread-level parallelism. SMTs seem to be a first-class choice for the execution of these applications. However, our results show that co-executing threads on SMT processors can often lead to limited performance gains, or even cause a slowdown, due to resource conflicts resulting from the high degree of resource sharing that characterizes the SMT architecture. We consider the effects of resource sharing on many resources within the processor by collecting data from hardware event counters. We specifically consider how resource sharing affects the number of L2 and L3 cache accesses and misses, bus accesses, data TLB misses, stall cycles and execution time.

The second contribution of this chapter is the investigation of alternative forms of multithreaded execution, both previously proposed and new ones, in an effort to overcome the poor performance of SMT processors at executing scientific applications. We consider adaptive thread throttling and speculative runahead execution. Although these techniques have been used in earlier work as tools to enhance SMT performance, they have been explored in different contexts (i.e. for speeding up sequential desktop workloads and selecting loop schedules). Most importantly, effective integration of these techniques with regular thread-level parallel execution in the same code has not been explored before. We propose methods to integrate adaptive throttling and speculative runahead execution with regular thread-level parallelization using the OpenMP [96] programming paradigm and we show that combining these techniques results in significant performance improvements.

The study presented in this chapter is a first step in deriving algorithms for selecting the best mode (or modes) of multithreaded execution for any given program. It is also the first study to investigate whether multiple multithreaded execution modes can be combined effectively in a synergistic manner to maximize performance on multi-SMT systems.

The rest of this chapter is organized as follows. In Section 3.1 we provide an overview of the experimental setting. In Section 3.2 we discuss and evaluate the execution of a broad set of parallel applications on a multi-SMT system. We also use various performance metrics, obtained directly from hardware performance monitoring counters on real silicon, to identify specific bottlenecks preventing more effective exploitation of these architectures. Section 3.3 describes the adaptive thread throttling mechanism we implemented for multi-SMT systems and provides an analysis of its performance. Section 3.4 goes over our speculative runahead execution approach, its integration with thread-level parallelization and its performance. Finally, in Section 3.5, we summarize our conclusions.

3.1 Hardware and Software Environment and Configuration

In order to ascertain the effects of the characteristics of modern processor architectures on the execution of OpenMP applications, we have considered multiprocessors based on SMTs [29]. SMTs incorporate minimal additional hardware in order to allow multiple co-executing threads to exploit potentially idle processor resources. The threads usually share a single set of resources such as execution units, caches and the TLB. CMPs on the other hand integrate multiple independent processor cores on a chip. The cores do, however, share one or more outer levels of the cache hierarchy, as well as the interface to external devices.

We used a real, 4-way server based on Hyperthreaded (HT) Intel processors as a representative SMT-based multiprocessor. Intel HT processors are a low-end / low-cost implementation of simultaneous multithreading. Each processor offers two execution contexts that share execution units, all levels of the cache, and a common TLB. Separate physical processors share only off-chip resources such as the memory bus. Table 3.1 describes the configuration of the two systems in further detail.

	Processors	**L1 Cache**	**L2 Cache**	**L3 Cache**	**TLB**	**Main Mem.**
SMT	4 x Intel P4 Xeon, 1.4 GHz Hyperthreaded x 2 Execution Contexts per Processor	8K Data, 12K Trace (Instr.), Shared	256K Unified, Shared	512K Unified, Shared	64 Entries Data, 128 Entries Instr., Shared	1GB

Table 3.1: Configuration of the SMT-based multiprocessor used throughout the experimental evaluation.

We evaluated the relative performance of OpenMP workloads on the target architecture using seven OpenMP applications from the NAS Parallel Benchmarks suite (version 3.1) [58].

We executed the class A problem size of the benchmarks, since it is a large enough size to yield realistic results. At the same time, it is the largest problem class that allows the working sets of all applications to fit entirely in the available main memory of 1GB.

We executed all the benchmarks on the SMT with 1, 2, 4, and 8 threads. The main goal of this experiment set was to evaluate the effects of executing 1 or 2 threads on the 2 execution contexts of each processor. We thus ran our experiments under six different thread placements: i) 1 thread, ii) 2 threads bound on 2 different physical processors, iii) 2 threads bound on the 2 contexts of 1 processor, iv) 4 threads bound on 4 processors, v) 4 threads paired on the execution contexts of 2 processors, and vi) 8 threads paired on 4 processors. Each thread is pinned on a specific execution context of a specific processor using the Linux `sched_setaffinity` system call. The applications were executed using Intel VTune [53] performance analyzer. We recorded both the execution time and a multitude of additional performance metrics obtained from the hardware performance counters available in the processor. Such metrics provide insight into the interaction of applications with the hardware, thus they are a valuable tool for understanding the observed application performance.

All experiments were performed on a dedicated machine in order to rule out data perturbations due to interactions with third-party applications and services. The operating system on both the real and the simulated system was Linux 2.4.25.

3.2 Experimental Results

In this section we present the results from the evaluation of the relative performance of the benchmarks with the configurations discussed in Section 3.1. The different binding schemes are labeled as `(num_processors, num_threads)`, where `num_processors` stands for the number of physical processors onto which the threads are bound and `num_threads` for the number of threads used for execution. We report timings and results obtained from monitoring several performance metrics, using hardware performance counters. We present a bottom-up evaluation starting from individual performance metrics and proceeding to the overall performance of the codes.

3.2.1 L2 Cache

The L2 cache of Intel's Hyperthreaded processor is shared between the two hyperthreads. The first metric we studied, the number of level-2 cache misses observed under each configuration, is depicted Figure 3.1. In many cases the number of cache misses goes up, sometimes considerably, when two threads are forced to execute on the same physical CPU. This is a result of cache interference between co-executing threads. If the working sets for both co-executing threads do not fit in the L2 cache, then cross-thread cache-line eviction is likely to increase the miss rate. Each thread is likely to evict the other thread's data from the cache and the effective amount of space in the cache for each thread is reduced. For the specific scientific benchmarks the cache is expected to be partitioned almost evenly, since threads perform nearly identical computation on different data. Nevertheless, the extra cache misses in the two thread case may be responsible for a reduced benefit of hyperthreading.

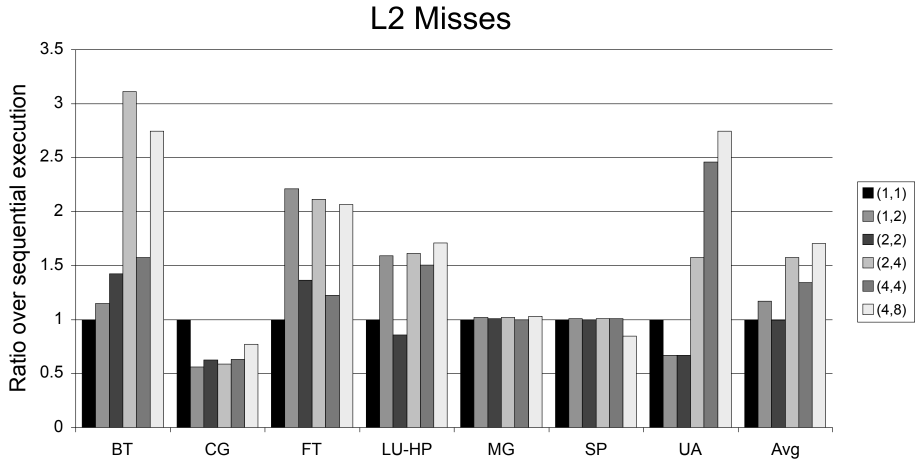


Figure 3.1: Normalized number of L2 cache misses of the benchmarks. The L2 misses of the sequential execution have been used as a reference for the normalization.

BT and FT are two benchmarks that exhibit high thread interference in the L2 cache. In BT, inter-thread conflicts account for an average increase of 69% and a maximum increase of

119% in L2 cache misses (happening on 2 CPUs). In FT, L2 cache misses increase by 82% on average and as much as 121% when the benchmark is executed on one CPU. UA also exhibits high interference on 2 and 4 CPUs, although in UA, this interference does not seem to hurt performance as much as in BT and FT, apparently due to the aggressive cache miss overlap mechanisms of the Pentium 4 (which can have up to 8 outstanding cache misses at a time). The average increase in the number of L2 cache misses when two threads are used on each CPU is 33%, taken across the entire suite of benchmarks. Cache misses can also increase with concurrency, even with more processors are used, due to false sharing in the cache.

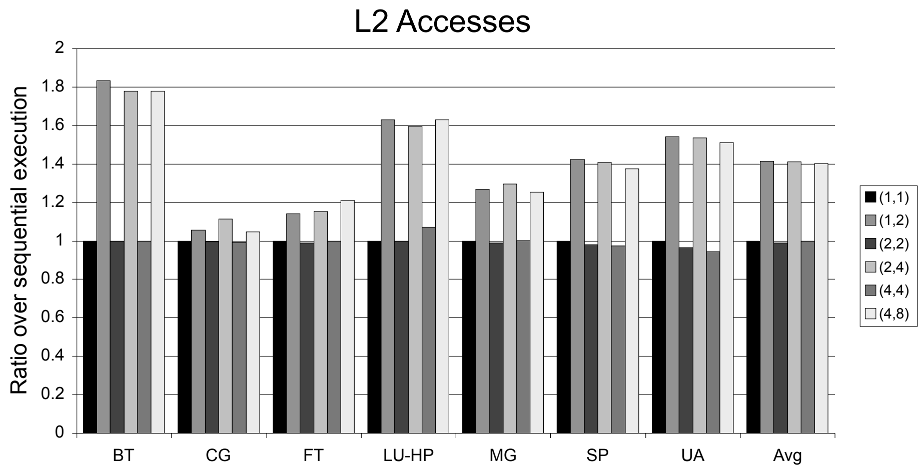


Figure 3.2: Normalized number of L2 cache accesses of the benchmarks. The L2 accesses of the sequential execution have been used as a reference for the normalization.

The number of L2 accesses (shown in Figure 3.2), an indicator of L1 cache misses, goes up from one thread to two threads per CPU by 42% on average. In fact, in all cases, running two threads on a single processor resulted in an increase in the number of L2 accesses. Larger numbers of L2 cache accesses are expected when two threads executing on each processor, because of inter-thread interference in the L1 cache. It is interesting to note that for a given number of threads per processor, the number of L2 accesses is quite stable. This means that, although with an increasing number of CPUs the total L1 cache space available to the program grows, the benchmarks do not benefit significantly from this increase. This phenomenon is a possible indicator of excessive conflict misses in the benchmarks.

3.2.2 L3 Cache

Inter-thread interference does not seem to affect the performance of the L3 cache significantly (shown in Figure 3.3). The notable exceptions are again BT and FT. Note that the number of L2 misses reported in Figure 3.1 corresponds to the number of L3 accesses. In general, L3 misses do not seem to closely correlated with either the number of threads used per processor or the number of processors used.

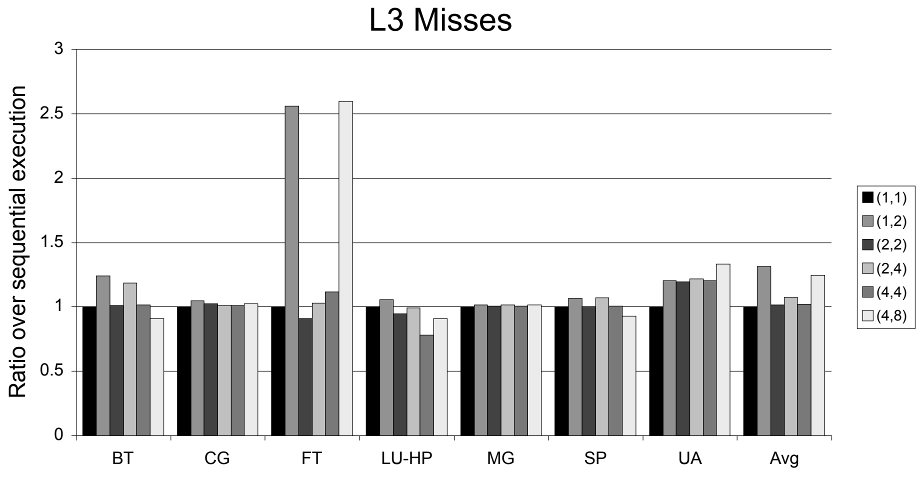


Figure 3.3: Normalized number of L3 cache misses of the benchmarks. The L3 misses of the sequential execution have been used as a reference for the normalization.

The L3 cache is twice the size of the L2 cache and this alleviates, to some extent, the problem of inter-thread conflicts. Furthermore, since most of the memory accesses are filtered in the L1 and L2 caches, sharing the L3 cache does not significantly affect the performance of these benchmarks. In the three benchmarks in which L2 suffers from inter-thread conflicts (BT, FT and UA), the L3 cache seems to exhibit a similar behavior (increased misses with two threads per processor) albeit at a much smaller scale for BT and UA. FT still suffers from a more than two-fold increase in the number of L3 cache misses, when two threads are used in the executions on 1 CPU and 4 CPUs.

3.2.3 Data TLB misses

The results observed for the number of data TLB misses per instruction show that intra-processor multithreading has a strongly adverse effect on the data TLB performance of SMTs (shown in Figure 3.4). The number of misses goes up by at least a factor of five when the configuration is changed from one thread to two threads on a single processor, and by factors of more than twenty in several cases. In one extreme case (occurring in SP) the rate of data TLB misses increases by a factor of 28. When more processors are used, the trend continues, but with lower differences between the one thread and the two thread per CPU cases. Overall, the rate of data TLB misses increases by an average of 925% from one thread to two threads per CPU. The poor data TLB performance is explained in part by the fact the data TLB of the Intel Hyperthreaded Pentium 4 is shared and relatively small (64 entries). Interestingly enough, although the processor uses a partitioned instruction TLB with 64 entries per thread, the data TLB has been chosen to be smaller and shared. It is often the case that co-executing threads work on different portions of the virtual address space, and therefore cannot share data TLB entries. This, in turn, results in an effective halving of the data TLB area per thread when both contexts of the SMT are activated. Both software and hardware approaches can be considered to alleviate data TLB thrashing in such cases, but investigating them is beyond the scope of this work.

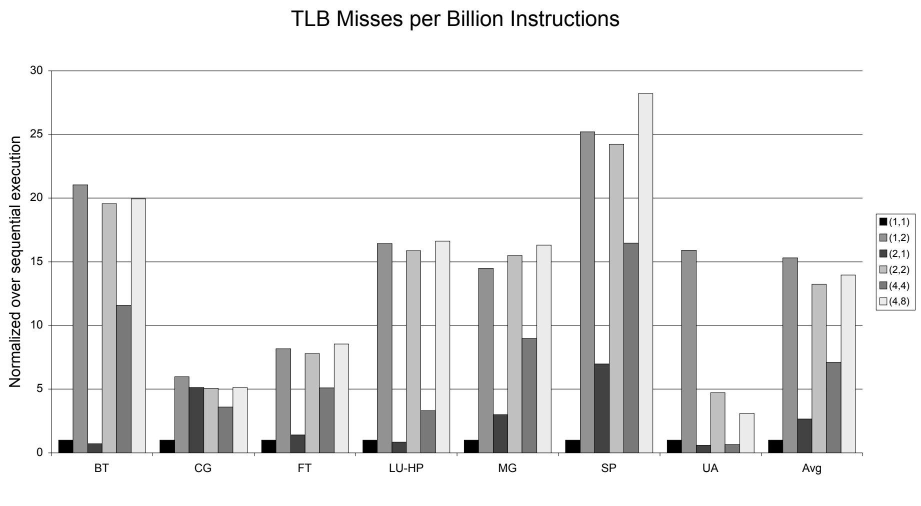


Figure 3.4: Normalized number of data TLB misses of the benchmarks per billion instructions executed.

3.2.4 Stall Cycles

SMT architectures share the vast majority of their resources, including functional units and all levels of the memory hierarchy. Stall cycles are a metric indicating how much time each application spends waiting, without making further progress, for either functional units or any level of the memory hierarchy to return a result. A relative change in stall cycles when using two instead of one threads per processor is an indirect indication of thread contention for all shared resources in the processor.

As shown in Figure 3.5, for nearly every benchmark and number of CPUs tested, the percentage of stall cycles goes up dramatically when the configuration is changed from one to two threads per processor. In only one case the number of stall cycles per total cycles executed goes down from one thread to two threads. There is an average increase in the number of stall cycles of 3.1 times when the second context of each SMT is used to run an additional application thread. Normalization of this data is done by the total number of stall cycles rather than the single-threaded stall cycles to provide insight into the total percent of cycles which are stalled under each configuration.

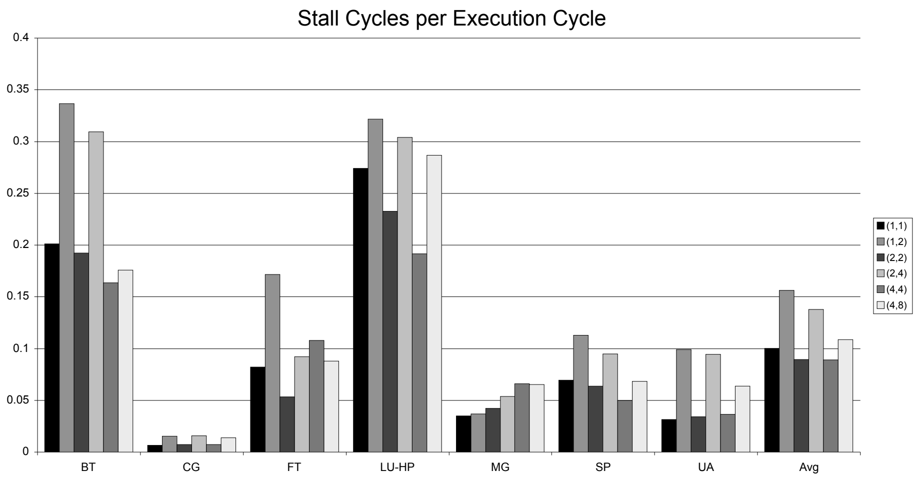


Figure 3.5: Number of stall cycles of the benchmarks normalized by the total number of cycles.

Although the results indicate indirectly that hyperthreading is responsible for a dramatic increase in stall cycles, the hardware performance counters of the Pentium do not provide sufficient information to characterize the stall cycles and attribute them specifically to inter-thread contention or other factors. We have used an indirect experimental method to confirm the intuition that the lack of resource replication should be blamed for the suboptimal performance when two threads run simultaneously on each processor. We executed the same experiments on a simulated chip multiprocessor (CMP) architecture with 4 dual-core processors, using the Simics simulation environment [35]. Due to the approximately 7000-fold slowdown in execution time under the simulator, we limited the execution of each benchmark to just the first three iterations of the outermost loop. In contrast to our real SMT processor, the CMP we modeled only shares the two outermost levels of the cache hierarchy and none of the functional units. This reduces the possible sources of stall cycles due to inter-thread interference, because on the CMP their number can be affected only by cross-thread interference in the L2 and L3 caches. On our simulated CMP machine, stall cycles were much more stable and relatively immune to changes in the number of threads per processor as well as to thread placement. The number of stall cycles only increased by an average of 3% on the CMP machine when the second hardware context was activated. These results provide evidence that the vast majority of stall cycles on the SMT machine were caused by the conflicting requirements of co-executing threads for internal processor resources.

3.2.5 Execution Time

Execution time is the ultimate performance metric. Although the results we obtained from hardware performance counters indicate several flaws of hyperthreading, it is important to observe whether these flaws translate into severe performance penalties. The execution times of the benchmarks are summarized in Figure 3.6, with data normalized with respect to the sequential execution times for each application.

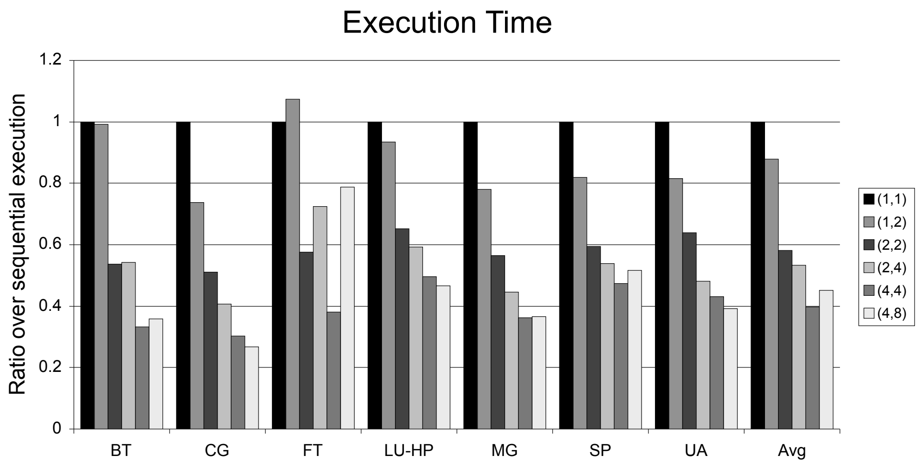


Figure 3.6: Normalized execution time of the benchmarks on the SMT multiprocessor. The sequential execution time is used as a reference for the normalization.

Not surprisingly, on the SMT architecture, execution time is always improved by running a fixed number of threads on as many different CPUs as there are threads, instead of co-executing two threads on each CPU. This is because running threads on separate physical CPUs eliminates the effects of inter-thread interference on the shared resources. Interestingly, it is not always clear whether it is better to use one thread or two threads per processor. For example, in CG, LU-HP and UA, using the second thread is beneficial. In CG, this is to be expected because the stall cycles results from hardware counters indicated little to no thread interference. The result is somewhat counter-intuitive for LU-HP, and even more so for UA, since both benchmarks suffer from increases in L2 misses and stall cycles when two threads are used per processor. In both cases, latency overlap through multithreading helps the benchmarks to mask and to counterbalance the negative effects of contention for shared resources.

One more interesting phenomenon is that for a given application, it may be the case that neither one thread nor two threads per processor is always the most effective choice. In SP, for example, using two threads per processor is more effective in the 1 and 2 CPUs case, but less effective for 4 CPUs. Furthermore, there are benchmarks that obtain no benefit or are actually harmed by hyperthreading. FT always suffers a slowdown (which is severe on 4 CPUs) when a second thread is used on each CPU. BT obtains very little benefit on 1 CPU and loses some of its performance from hyperthreading on 2 and 4 CPUs.

One may easily suspect that the decrease in performance observed with the introduction of the second thread comes from increased overhead from further parallelization rather than side effects of the SMT architecture. However, two of the seven benchmarks saw decreased performance from two threads to four threads on two processors, while rebounding substantially when the four threads were allowed to run on four seperate physical processors.

When two threads run together on a single processor, there is an average speedup of merely 7%, compared to running with one thread per processor. The CG benchmark enjoys the best gains from two threads per CPU with an average speedup of 25%. UA is a close second with 22%. There are two benchmarks that, on average, see a slowdown with the use of the second Hyperthread, namely BT (4%) and FT (35%). In the rest of the chapter we focus on these two benchmarks, and use them as motivating examples for introducing adaptive and more flexible multithreaded execution mechanisms to utilize SMTs in parallel programs.

3.3 Adaptive Selection of the Optimal Number of Execution Contexts on SMTs

Based on our observations about the execution of parallel applications on SMT architectures, we implemented a means by which each application can adaptively choose the optimal number of threads to use per physical processor. The mechanism exploits the fact that under some circumstances two threads per processor will execute more effectively and other times only one will, because of contention. A similar adaptive execution mechanism was used previously [123] to select the optimal number of threads and the best scheduler for each parallel loop, using code injected by an OpenMP compiler. Our implementation of this mechanism differs in that it is based on a compiler-independent runtime library that can be used via a preprocessor. In our approach, the application makes calls to our library at runtime that measure execution times of the application and then change the number of threads used and thread bindings if it is likely to improve performance.

To make use of our library we slightly changed the semantics of the `OMP_NUM_THREADS` environment variable. Whereas it is usually used to specify the number of threads to use for execution, we use it as a suggestion for the number of processors to be used. Additionally, we introduce a new environment variable, `OMP_SMT`, that specifies the number of threads to use per processor. If it is set to one or two, then the application will use one or two execution contexts respectively. Alternatively, if its value is set to 0 or is undefined, then the system will activate adaptive execution.

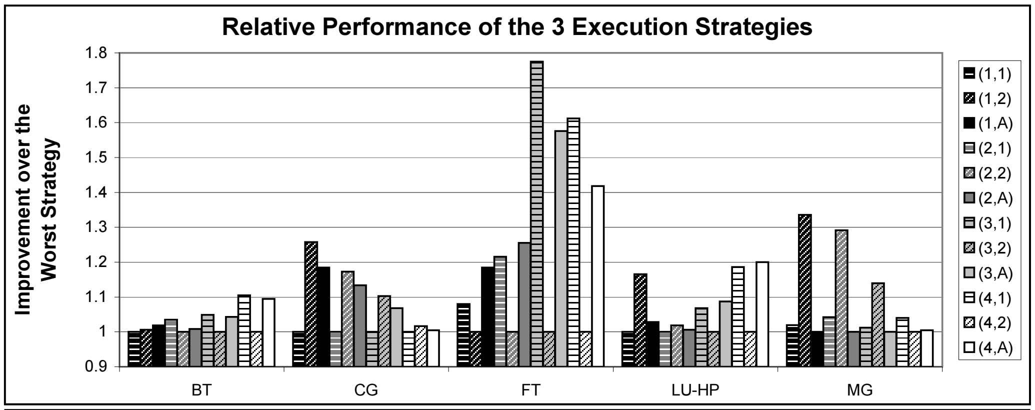
Our runtime library is called at the beginning and end of each parallel region so that a decision can be made for the optimal number of threads to use for this region. In scientific codes, parallel regions typically delimit phases of the code with different execution properties and performance characteristics. Throttling concurrency at the entry points of parallel regions is therefore an effective means also to control the performance of the code. Furthermore, since in most scientific applications parallel regions are executed many times across different iterations of outermost loops, the first few executions of each parallel section can be used to come to a conclusion on the optimal concurrency. In our experiments, we used three iterations for this purpose. We ignore the first iteration to account for warmup effects. Then the library tries both one thread per processor and two threads per processor, for one iteration each, and compares their execution times. Whichever number of threads results in the lower execution time is used in the future whenever this loop is encountered.

The major advantage of adaptive throttling is that it can achieve runtime performance optimization without modifications to the parallelization and execution strategies of applications. Adaptive thread control is an easy feature to implement in any thread management library. With proper engineering, it can be used transparently and effortlessly in any shared-memory programming model. The disadvantage of adaptive throttling is that it leaves idle hardware threads, which could be used for the purpose of accelerating the code running on the non-idle hardware threads. We explore this option via speculative runahead execution in Section 3.4. By having the OpenMP code call into our library at the beginning and end of each parallel section, the process of inserting the calls can be automated. This is important because the application programmer will not have to add any code manually to achieve the benefits of our library. Further, due to the location of the calls, the process can be performed by a simple preprocessor that just inserts the calls whenever it sees a parallel region. This allows the automation of the process to occur without modification to the compiler or the OpenMP runtime. This last point is one thing that distiguishes our work from that of Zhang, et al. [123].

3.3.1 Results from Adaptive Thread Throttling

We tested our adaptive thread throttling mechanism, using the NAS Parallel Benchmarks, and the OpenMP codes MM5 and COBRA. MM5 [46] is a mesoscale weather prediction model, and COBRA [7], a matrix pseudospectrum computation code. We ran each of the benchmarks with one and two threads per processor on 1, 2, 3, and 4 processors, with the number of threads per processor fixed throughout the execution of each benchmark, to obtain a baseline for comparison against adaptive throttling. In all cases the threads were bound to specific execution contexts to prevent any penalization from a suboptimal thread placement by the Linux scheduler. Following, we activated the adaptive thread throttling mechanism and repeated the experiments on 1, 2, 3, and 4 processors. The results are summarized in Figure 3.7.

The results show that adaptive thread throttling fares well compared to a static execution scheme selected from an oracle. When compared to the optimal static number of threads for each case, the adaptive mechanism is only 3.0% slower on average. In comparison, adaptation achieves a 10.7% average speedup over the worse static number of threads for each benchmark. The average speedup observed over all cases is 3.9%. A closer look at the results reveals that adaptively selecting the number of threads provides, in many cases, improvements over running with a static and fixed numbers of threads. In fact, in 17 out of the total 36 experiments, adaptive



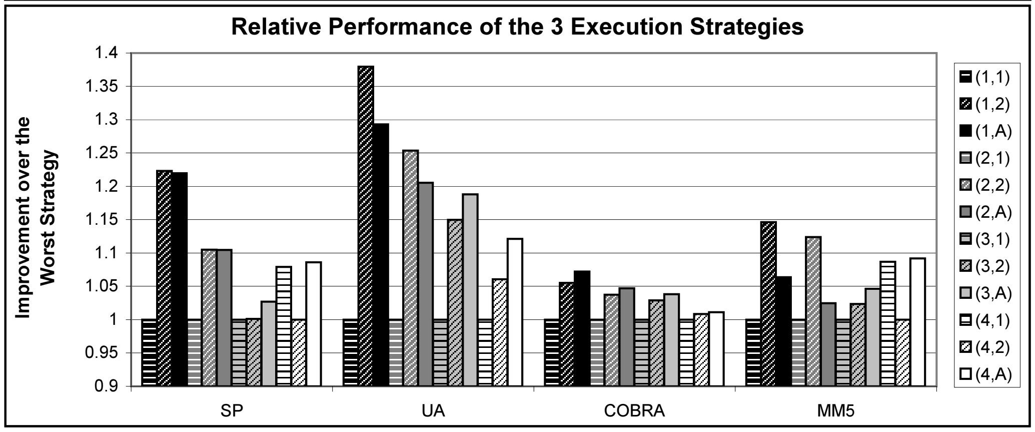


Figure 3.7: Relative performance of adaptive, 1 and 2 threads per physical processor execution strategies. The execution times have been normalized with respect to the execution time of the worst strategy for each experiment

throttling outperforms both static executions (with one thread and two threads per processor). Adaptive thread control consistently yields the best performance in COBRA on any number of processors and achieves a speedup improvement in several other cases as well. This result was made possible because some of the applications possess phases that have different optimal numbers of threads. Our approach is able to exploit this by optimizing the number of threads for each parallel region.

Two applications for which adaptive throttling does not perform well are MG (in all cases) and FT (on 3 and 4 processors). The main reason for the poor performance of adaptive thread control is that neither has a sufficient number of iterations to amortize the startup costs of searching for the best number of threads to use. MG performs only 4 iterations and FT only 6 iterations, so with the 3 iteration initialization phase, the applications only use the adaptively selected number of threads for 1 and 3 iterations respectively. This result exposes the main shortcoming of any adaptive control strategy based on runtime information.

One question that arises from these experiments is how the adaptive throttling mechanism performs against a static execution with an oracle that knows how many threads to use on a loop by loop basis, rather than in the entire program. To answer this question, we conducted additional experiments with BT, which suffers an average 4% slowdown when the second hardware context is used. BT and FT are the two benchmarks in which there seems to be no benefit from using the second hyperthread on each processor. We recorded the number of threads used for each section by the adaptive approach and hardcoded it in the application. When the best number of threads for each parallel section is given from an oracle, BT enjoys a speedup of 1.8% from the occasional use of the second hyperthread per processor in a selected set of parallel loops. The adaptive throttling mechanism converges to the optimal number of threads for each parallel section, but yields an average slowdown of 1.1% compared to static execution with one hyperthread used per processor. This indicates that the adaptive mechanism performs comparably, but still pays some additional and apparently significant overhead to converge to the best degree of multithreading across the code. Therefore, faster analysis and convergence mechanisms may be needed to increase the effectiveness of adaptive throttling. This is important especially for future SMTs with more threads, since a solution that simply polls all possible degrees of multithreading before converging is not scalable. Finally, it must be noted that the Hyperthreaded processors have a fully shared cache hierarchy. This means that changing the number of threads used on each processor is not expected to affect performance adversely along the memory hierarchy, since for any given loop, any number of threads will load approximately the same data (with perhaps a different access and eviction pattern) in a shared cache. This may not be the case on SMTs with private L1/L2 caches per hardware thread, in which adaptive thread control may cause implicit data migration between caches. Drawbacks such as potential compromises in memory performance and slow convergence necessitate the consideration of other options for utilizing simultaneous multithreading to improve performance.

3.4 Integrating Simultaneous Multithreading with Speculative Precomputation

The earlier discussion revealed that some programs are highly sensitive to contention for shared resources on SMT processors [29]. The most characteristic example is FT, in which using the second thread on an SMT yields a slowdown of 35% on average. The main computational kernel of FT is composed of three FFTs (along the x, y, and z dimensions), each of which walks two arrays (named x and $xout$ in the code) with very long strides (equal to 256×128 elements, or 32K), causing an excessive amount of cache misses and TLB faults, as well as high contention for memory bandwidth. In memory-intensive codes such as FT, simultaneous multithreading intensifies the effect of the major performance bottlenecks, namely cache space and memory bandwidth. An obvious solution in this case would be to throttle simultaneous multithreading, providing all the resources of the processor to a single thread. Adaptive throttling was successfully used for example in BT, which also suffered performance degradation on the executions with 2 hyperthreads per processor. However, BT did not exhibit as acute cache contention and memory bandwidth problems as FT and several parts of the code could be executed with a reasonable speedup using both hyperthreads on each processor. On the contrary FT, would benefit from direct memory latency reduction and / or latency tolerance techniques.

Speculative precomputation (SPR for short) is a memory latency reduction technique proposed specifically for SMT and chip multiprocessors [18]. SPR uses either microarchitectural or software support to employ an otherwise idle thread for precomputing addresses of *critical* memory accesses and prefetching the data touched from these accesses into the cache, anticipating that the data will arrive early enough to be used from a sibling computation thread without suffering cache misses. Critical accesses are those responsible for a large number of cache misses in the outermost levels (L2, L3) of the cache hierarchy. SPR is based on run-ahead execution. The precomputation thread runs a reduced copy of the sibling computation thread, in which only the slices of instructions that lead to critical memory accesses and the accesses themselves are executed. In most cases, this allows the precomputation thread to execute faster and ahead of the computation thread, which in turn enables the precomputation thread to prefetch data in a timely manner into the caches.

SPR is a resource-conserving approach to leverage multithreading on a single chip, since the speculative thread works in synergy and to the benefit of the non-speculative thread. It has been used successfully to speed up sequential codes dominated by pointer chasing [18, 116] but less attention has been paid to using SPR for scientific codes, partially because memory access patterns in these codes are often highly predictable. We argue that SPR, using a spare hardware context, has value in scientific codes with long streams of long-strided memory accesses as well, not necessarily because of better prediction abilities, but because of being less resource-consuming than in-place sequential prefetching, without sacrificing timeliness. To evaluate this hypothesis, we applied SPR in conjunction with thread-level parallelization in the FT code, which proved to be highly problematic when parallelized using multiple threads within an SMT. We investigate whether SPR can improve its scalability on multi-SMT systems.

To use SPR, we had to devise a method that would allow assisted run-ahead execution within an SMT, while still enabling loop-level parallel execution across SMTs. We used the nested parallel execution features of OpenMP. We organized the code of each one-dimensional FFT in two parallel sections, one executing the main FFT loops with multiple threads bound to different SMTs, and the other executing speculative precomputation threads also bound to different SMTs, in the contexts left unused by the main computation threads. Effectively, the program uses two levels of heterogeneous parallelism to merge precomputation with regular multithreaded code. One limitation of this technique is that the scheduling of both the parallel loop and the precomputation loop should be aligned, meaning that both loops should be scheduled with the same strategy and this strategy should deterministically assign the same set of iterations to each thread.

We have identified critical memory accesses using an execution-driven cache simulator derived from Valgrind. We ran a stripped-down binary of FT with only two iterations of the PDE solver, in which we have identified that more than 95% of the L2 and L3 cache misses in FT occur in three routines of the code (`cffts1`, `cffts2` and `cffts3`, corresponding to FFT's in the x, y and z direction respectively), and on just two elements, x and $xout$, corresponding to the input and output vectors of each FFT. Note that x and $xout$ point to the same vector in `cffts2` and `cffts3` but to different vectors in `cffts1`. Interestingly enough, we found that the dominating misses prove in most applications to be the same, regardless of the problem size, the data input and the number of processors/threads used to execute them. This indicates that a profile-driven approach for identifying critical loads is quite effective.

We have constructed the precomputation threads by stripping out the accesses to x and $xout$ from the sibling computation loops. An example of the precomputation code in subroutine `cffts3` is shown in Figure 3.8. The precomputation code is composed of loops with the same bounds and structure as the sibling computation loops, except for the innermost loop, in which the precomputation code performs strided prefetching. We have used the native Intel prefetch instructions (`prefetchnta`, `prefetcht0|1|2|3`), and experimented with all prefetching

```
!$omp parallel sections num_threads(2)
!$omp section
!$omp parallel  private(i,j,k,jj,y1,y2)
!$omp&  shared(is) num_threads(4)
!  bind thread i on hardware thread 2*i
!$omp do onto (2*i)
      do k = 1, d(3)
         do jj = 0, d(2) - fftblock, fftblock
            do j = 1, fftblock
               do i = 1, d(1)
                  y1(j,i) = x(i,j+jj,k)
               enddo
            enddo
            call cfftz (is, logd(1),
     >                  d(1), y1, y2)
            do j = 1, fftblock
               do i = 1, d(1)
                  xout(i,j+jj,k) = y1(j,i)
               enddo
            enddo
         enddo
      enddo
!$omp end parallel
!$omp section
!$omp parallel private(i,j,k,jj,y1,y2)
!$omp&  num_threads(4)
!  bind thread i on hardware thread 2*i+1
!$omp do onto (2*i+1)
   do k = 1, d(3)
      do jj = 0, d(2) - fftblock, fftblock
         do j = 1, fftblock
            do i = 1, d(1), stride
              prefetch((x(i,j+jj,k),mode)
            enddo
          enddo
       enddo
    enddo
!$omp end parallel
!$omp end parallel sections
```

Figure 3.8: Merging thread-level parallelism and precomputation, using nested parallel execution in OpenMP. The `onto` clause is used to instruct the compiler to enforce binding of specific OpenMP threads on specific hyperthreads. Although not available in vendor OpenMP compilers, the `onto` clause is an experimental OpenMP extension proposed by the NANOS compiler group [45]. We use it in this example for the sake of brevity. In our implementation we actually emulate the `onto` functionality by inserting calls to our runtime library.

modes available on the processor. Among these modes, prefetching solely in the L2 cache to avoid L1 cache pollution, and prefetching in selected ways of the L2 cache to avoid both L1 cache pollution and L2 cache conflicts were the most effective. We opted for the latter method. Strided prefetching was necessary because each prefetching instruction fetches at least 32 bytes. The simple transformation performed is highly automatable by a compiler.

Note that our precomputation method is implemented entirely at user-level with no special hardware or compiler support, and is in general much simpler to use than schemes proposed earlier in the literature. Furthermore, although a strided, in-place software prefetching scheme could be used in the case of FT, since the critical loads are predictable, such a scheme would occupy many instruction slots and other precious resources while executing on the critical path of the main computation thread. Offloading strided prefetching to a speculative thread alleviates this problem via partial decoupling of the speculative and the non-speculative instruction stream. Moreover, the use of SPR in some loops does not prevent the use of TLP in other loops, via regular parallelization. In the experiments we used TLP in the initialization loops of FT as well as the left-hand side (*lhs*) update loop of the PDE solver, in every iteration of the code. We used one Hyperthread per processor during initialization loops, and two hyperthreads per processor in the lhs update loop, which benefits from SMT. The FFTs within each iteration were accelerated with SPR.

3.4.1 Results from SPR on Deactivated Execution Contexts

Figure 3.9 summarizes the performance results of the three execution strategies. `Adapt+prec` corresponds to experiments in which parallel execution across CPUs is combined with SPR within CPUs for the three main FFT routines. This hybrid execution method is compared against thread-level parallel execution with 1 or 2 hyperthreads per processor. In the thread-level parallel execution we have activated software prefetching via the Intel compiler, but the same option was deactivated in the executions with our SPR mechanism, to avoid conflicts be-

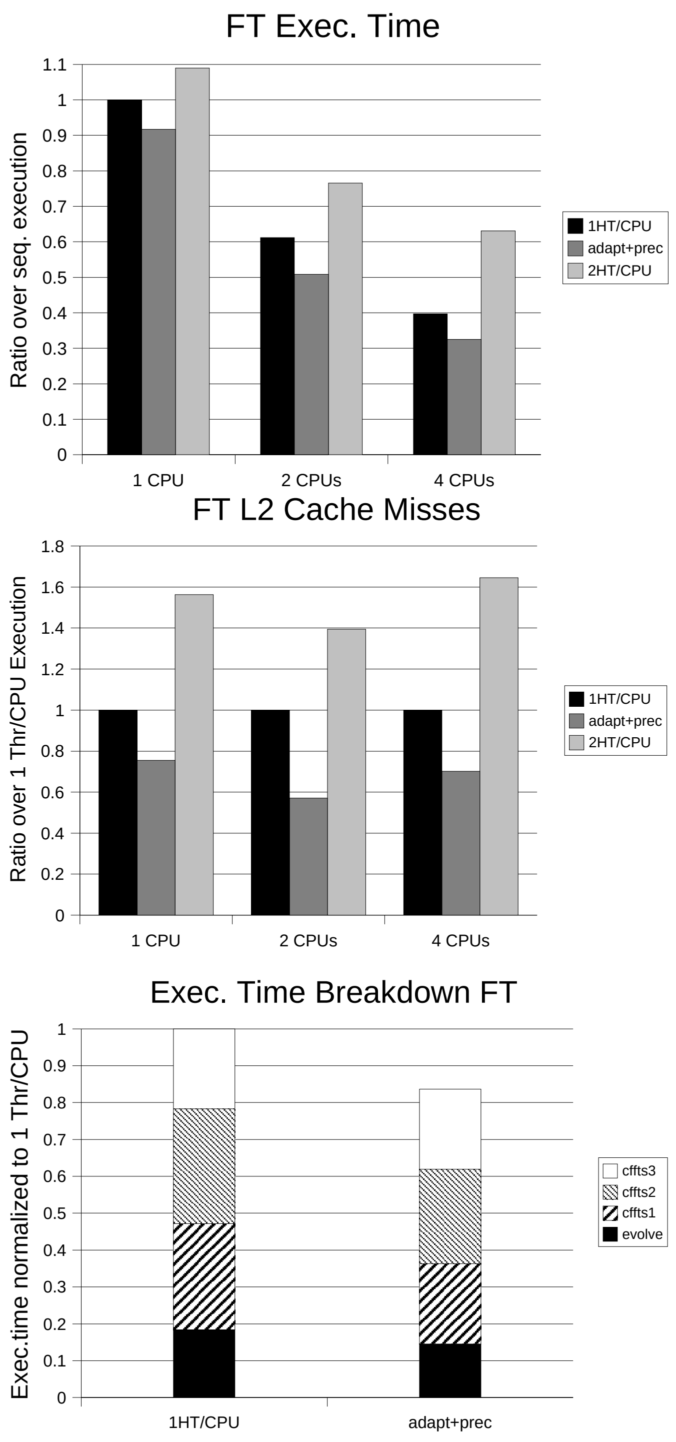


Figure 3.9: Impact of using selective speculative precomputation in conjunction with thread-level parallelism in NAS FT.

tween our prefetching code and code inserted for the same purpose by the Intel compiler. Note that there might still be some interference between SPR code and the hardware prefetching engine of the Hyperthreaded processor. However, since this engine is a black box to the software we could not exert any control over this type of interference.

SPR in the three memory-bound loops successfully eliminates 25–43% of the L2 cache misses in FT (Figure 3.9(b)). This, in turn, translates to execution that is 9%–22% faster than the parallel execution with one hyperthread per SMT CPU (Figure 3.9(a)), as opposed to a slowdown of up to almost 40% incurred in executions in which two hyperthreads split the parallel computation on each CPU. The speedup over the sequential execution time of FT with one hyperthread on one CPU is improved from 2.5 to 3.0 on 4 CPUs. It is important to note that this speedup arises mostly but not solely from the use of SPR. Figure 3.9(c) gives a breakdown of execution time between the four primary subroutines of FT, obtained during execution on 4 CPUs. The subroutine `evolve` executes the lhs update code of the PDE solver. Two of the three FFT routines (`cffts1` and `cffts2` along the x, and y dimensions) benefit significantly from prefetching and enjoy speedups of 18% and 25% respectively, while the third FFT routine (`cffts3` along the z direction) obtains little benefit. On the other hand, the routine `evolve` benefits from regular parallelization with two hyperthreads per processor, with a speedup of almost 20%. In conclusion, mixed-mode multithreaded execution yields superior performance in a code in which thread-level parallelization within an SMT suffers from conflicts due to simultaneous multithreading. Several scientific applications exhibit such behavior, therefore the proposed adaptive multithreaded execution mechanisms are expected to play an important role in scaling these applications on high-end, SMT-based systems.

3.5 Summary

Simultaneous multithreading is a processor design methodology that merges thread-level parallelism and instruction-level parallelism to surpass the performance of conventional superscalar processors with minor architectural modifications. SMT processors are currently designed with memory hierarchies and execution resources that are shared between threads, because sharing offers a number of advantages with respect to area, power and complexity. Unfortunately, sharing introduces a number of performance implications that limit the ability of programs parallelized with conventional methodologies to utilize multiple threads on the same processor. This chapter has illustrated these implications, using physical experimentation on a real multi-SMT system and several performance metrics obtained from hardware event counters.

We proposed the use of two software mechanisms for improving the performance of parallel codes running on multiple SMT processors. First, with adaptive thread throttling, the number of threads used on each SMT is regulated per parallel section, so that code phases that do not benefit from SMT execution are adaptively sequentialized. Using an iterative process for adaptation, we were able to achieve performance close to that obtained when the optimal number of threads per parallel section is known a priori and used from the beginning of execution. Although quite effective, the adaptive throttling strategy has several limitations, the most important of which are underutilization of hardware threads and delayed convergence to the optimal number of threads in codes with only a few iterations.

The second technique we considered was speculative precomputation. In this approach, the second context is used to perform precomputation, to load data needed by the thread executing on the first context into the shared cache. We proposed a mechanism to merge speculative precomputation with conventional thread-level parallel execution in OpenMP and we have shown that this mechanism can significantly improve performance in parallel codes with long streams of long-strided memory accesses. Combined with selective thread throttling, our hybrid execution mechanism achieved a 9%–22% performance improvement from the use of the second hyperthread on each processor in FT, a code which proved to be particularly difficult to scale otherwise within an SMT.

Our study has shown that significant performance gains can be achieved through the use of multiple alternative forms of multithreading. Further, we have shown that these forms are not only effective independently, but they can work together to allow an even more efficient use of SMT processors. We believe this work will motivate compiler writers and application developers to integrate multiple forms of multithreading into a single binary in order to achieve maximum performance.

This page intentionally left blank.

Chapter 4

DPAPP Predictor

The goal of dynamic phase-aware performance prediction (DPAPP for short) is to predict the performance of a multithreaded, compute-intensive region of code in a program – which we hereafter refer to as a *phase* – across varying configurations of the processing units on a shared-memory parallel architecture [23, 24]. We use the term *processing units* as an umbrella term covering hardware threads, scalar or superscalar processor cores, and single- or multiple-chip uni- or multi-processors. As a base hardware substrate, we consider shared-memory multiprocessors with three distinct types of processing units, namely processors, cores within processors, and threads within cores. We refer to each of these types of processing units as a *dimension of parallelism* in the system. More formally, we define a *dimension of parallelism* as a set of homogeneous processing units that share a given level of the memory hierarchy, which is also shared by processing units nested in lower dimensions of parallelism, but not shared by processing units in higher dimensions of parallelism. In principle, each dimension of parallelism shares a distinct set of execution and memory resources, and therefore exhibits distinct scalability properties. The dimensions of parallelism that we consider are representative of current commercial multiprocessors [64, 71]. Our DPAPP technique considers phases that are identified as parallel loops, as these structures encapsulate the bulk of parallel code in real scientific applications.

Our DPAPP model works by predicting the cumulative *useful* Instructions Per Cycle ($uIPC$) of multithreaded phases. While $uIPC$ has been previously defined to exclude instructions expended for synchronization [84], we extend this definition to exclude cycles executed for performing parallelization. Ignoring parallelization and synchronization overheads makes $uIPC$ inversely proportional to the execution time of a fixed number of instructions on a given hardware configuration. If synchronization instructions were to be included, the IPC value could become arbitrarily inflated, as a result of spinning, leading to IPC values far removed from actual system performance [84]. Similarly for parallelization instructions, inclusion would bias the recorded IPC value away from the performance for the actual computation of a code region. Note that although $uIPC$ ignores instructions for triggering and synchronizing threads, it still considers the effects of interference between threads on shared hardware resources during concurrent execution. The objective of DPAPP is to identify phases where concurrency can be reduced during the execution of useful application computation, with a non-negative impact on performance. The use of $uIPC$ as a prediction target focuses the optimization process on lengthy, compute-intensive parts of applications, where performance and power optimization opportunities may be limited with means other than concurrency throttling.

In the next section, we present an overview of the scalability prediction model and then provide the underlying details in Section 4.2. In Section 4.3, we describe the offline training process used to develop the model based on a set of training samples. Section 4.4 presents a technique to identify effective hardware events that heavily impact scalability, and should therefore be included in the model. Section 4.5 discusses the online use of the offline-derived model. We present an approach whereby our predictor can be used on architectures with multiple dimensions of parallelism in Section 4.6. Section 4.7 describes a few optimizations that were applied to the original model to provide increased accuracy based on architectural insight. We thoroughly evaluate the model in terms of prediction accuracy in Section 4.8 and provide a brief summary of this chapter in Section 4.9.

4.1 DPAPP Outline

DPAPP makes distinct predictions on the optimal number of processing units to use at each dimension of parallelism in the system. For ease of presentation, we first describe the operation of DPAPP for a given dimension of parallelism d. We defer the discussion of how DPAPP predicts across dimensions of parallelism until Section 4.6.

DPAPP takes input from live samples of hardware event counters (*HECs*). HECs are sampled at the beginning and end of each phase, while the phase is executed on the configuration that activates all processing units at dimension d. The set of hardware events sampled are specific to d and are selected using a formal statistical process, according to their contribution to $uIPC$. We refer to these events as *critical events*. Samples of critical event rates are fed to a model that estimates $uIPC$ per phase, per configuration, for all feasible configurations of processing units at dimension d. Intuitively, DPAPP attempts to predict how the rate of retirement of useful instructions, $uIPC$, will change in a phase when the number of processing units used to execute the phase changes. To make this prediction, DPAPP uses a multivariate regression model, which correlates observed event rates on a sample configuration and observed $uIPC$ values on all feasible hardware configurations during training runs. The model outputs a set of scaling factors for $uIPC$ and the critical hardware events, for each feasible hardware configuration. These outputs are used as constant coefficients during production runs, to predict optimal operating points of concurrency for each phase in the code. We describe the model in more detail in Section 4.2 and the process for training the model in Section 4.3. The process for selecting critical events is discussed in Section 4.4.

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

4.2 $uIPC$ Prediction Model

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

Although in theory, the DPAPP predictor can use any feasible configuration as a sample configuration, we heuristically chose to use the configuration where all processing units at the given dimension of parallelism are active. Intuitively, $uIPC$ and the critical event rates sampled in this configuration encapsulate the cumulative impact of hardware components on scaling, at maximum system capacity at the given dimension of parallelism. In practice, these values are often direct predictors of scalability. For example, on our experimental platform, a quad Intel Xeon HT server, phases with cumulative $uIPC > 1.0$ across the four processors are always scaling linearly to four processors, therefore they offer no opportunity for concurrency throttling via processor deactivation. Similarly, phases with cumulative $uIPC$ less than 0.25, are never scalable to four processors and offer opportunities for concurrency throttling and power conservation, via processor deactivation. Cumulative $uIPC$ values between 0.25 and

1.0 are harder to predict, however they still exhibit a bias, i.e., phases with $uIPC$ close to 1.0 tend to be scalable and phases with $uIPC$ close to 0.25 tend to be non-scalable.

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

$$\overline{uIPC(t)} = uIPC(s) \cdot \alpha(t, e_1(s), ..., e_n(s)) + \beta(t) \tag{4.1}$$

for a set of n critical hardware events, which may function either as enhancers, or as impediments of scalability. The selection of the events in this set is discussed further in Section 4.4. Notice that both the scaling factor α and the residual β of the linear function are specific to and dependent on the target hardware configuration t. In other words, each target configuration t exerts its own scaling impact on $uIPC(s)$, which can be positive or negative. To gauge how individual critical events affect scalability, the linear scaling factor is in turn modeled as a linear combination of hardware event rates observed during the sampled configuration s:

$$\alpha(t, e_1(s), ..., e_n(s)) = \sum_{i=1}^{n}(x_i(t) \cdot e_i(s) + y_i(t)) + z(t) \tag{4.2}$$

The linear model of event rates stems from the empirical observation that a change in the configuration used to execute a program phase will result in changes – either upwards or downwards – of critical hardware event rates, reflecting the contention or effective hardware utilization at each level of parallelism. These event rates are linearly related – positively or negatively – with the $uIPC$. This relation is captured in Equation 4.2 with positive or negative event coefficients respectively. Our model attempts to estimate these coefficients using multivariate regression, discussed further in Section 4.3.

Combining equations 4.1 and 4.2, the estimated $\overline{uIPC}$ for a target configuration t can be calculated as:

$$\overline{uIPC(t)} = uIPC(s) \cdot \sum_{i=1}^{n}(x_i(t) \cdot e_i(s)) + uIPC(s) \cdot \gamma(t) + \beta(t) \tag{4.3}$$

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

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

4.3 Offline Training and Estimation of Coefficients

We use multivariate linear regression on the multithreaded phases of a set of training benchmarks to determine the values of the coefficients in Equation 4.1. Although more advanced machine learning techniques could be deployed for prediction, the number of cycles invested in making predictions at runtime is a primary concern for DPAPP, therefore we opt for the simplest linear prediction model. Specifically, training benchmarks are executed under all feasible hardware configurations, at all dimensions of parallelism, while recording per-phase $uIPC$ and the critical hardware events used for prediction (see Section 4.4). The training benchmarks are selected empirically so as to include phases with variance in three characteristics: scalability ranging from poor to perfect; granularity of parallel computation, ranging from fine to coarse;

and ratio of computation to memory accesses, ranging from low to high. In our experimental evaluation, we use two parallel benchmarks (MM5, a mesoscale weather modeling code and the Unstructured Adaptive application from the NAS Benchmarks) with 119 phases in total. In this work, we define phases to be OpenMP parallel regions enclosing parallelized loops. The training benchmarks achieve excellent coverage of diverse phase characteristics.

Our multivariate regression analysis uses the events collected under the selected sample configuration s multiplied by the $uIPC$ of the sampled configuration, i.e. $e_i(s) \cdot uIPC(s)$, and the actual $uIPC$ alone ($uIPC(s)$) as *independent variables*, to predict the $uIPC(t)$ of each target configuration t as the *dependent variable*. We use the product of e_i and $uIPC$ of the sampled configuration for coefficient derivation because our model uses multiplicative effects of events on the observed $uIPC$ rather than additive ones, in accordance with Equation 4.3. This process estimates the necessary coefficients for each event in function $\alpha(t)$. Regression analysis is performed separately to predict $uIPC$ for each target configuration t, therefore we derive independent sets of coefficients and independent scaling factors for each target configuration. For a system with p_d units in dimension d of parallelism, $1, \ldots, D$, multivariate regression analysis derives a total of $\sum_{i=1}^{D} p_d$ sets of coefficients.

4.4 Rigorous Event Set Selection for $uIPC$ Prediction

The accuracy of DPAPP is heavily dependent upon the selection of an effective set of critical events for predicting performance and scalability along each dimension of parallelism. The events should accurately reflect, in a statistical sense, performance and scalability bottlenecks in the system. We have previously considered empirical selection of events that represent known performance-critical components of microprocessors [24]. We have also considered exhaustive searches of all possible sets of events and evaluated each in terms of its resulting correlation coefficient for the training set of benchmarks [23]. In this chapter, we present a rigorous statistical technique, that automates event selection and makes it applicable to any architecture.

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

Rather than exhaustively looking at each possible combination of events, our predictor training tool independently looks at the contribution of each event to $uIPC$. To gauge each event's significance, we initially use multivariate regression on data from the set of training benchmarks to predict $uIPC(t)$ for each target configuration, using all events that are available for monitoring on the processor. We model $uIPC$ as in Equation 4.3, with the exception that we use a set of N events where $N >> n$.

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

4.5 Process for Online Use of Coefficients

Performance prediction is performed by executing the application phase on the test configuration while recording the decided upon set of event counters. The model for each desired target configuration can then be applied to these events to produce the performance estimate. Since the model is a linear combination of each event's effects on IPC, each event rate observed during the test configuration must be multiplied first by the observed IPC and then by its corresponding coefficient. These products, as well as the observed IPC times its coefficient and the residual, are then added together to form the predicted IPC for a given target configuration.

4.6 Prediction on Architectures with Multiple Dimensions of Parallelism

On architectures with multiple dimensions of parallelism, resource sharing varies considerably across dimensions. For example, physical processors in an SMP share only the off-chip interconnection network and DRAM. Cores within a processor typically share an on-chip interconnection network and the outermost levels of the on-chip cache. Threads on a single core share most resources of the execution core, including pipelines, branch predictors, TLB and L1 cache. Contention for these shared resources is largely responsible for performance and scalability. An example architecture with three layers of parallelism is presented in Figure 4.1.

To capture the implications of multidimensional parallelism, DPAPP uses a distinct set of critical events and derives a distinct set of scaling factors for each dimension of parallelism in the system. Each set of events required is recorded on a particular threading configuration

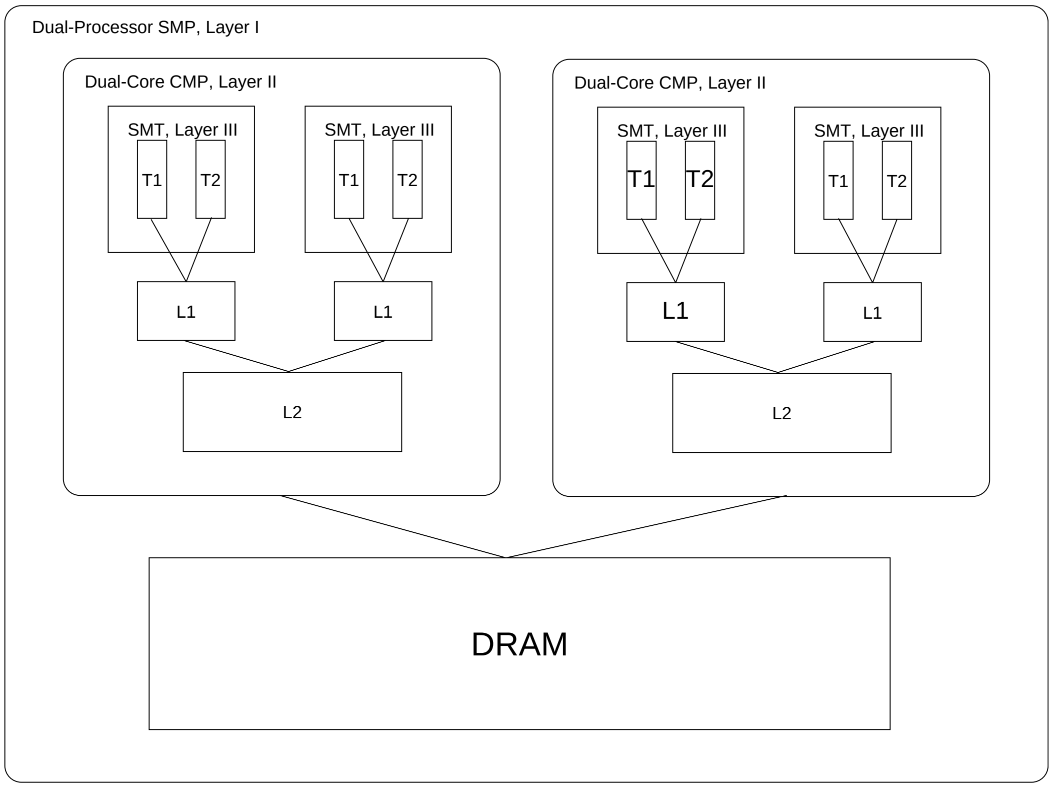


Figure 4.1: A shared-memory multiprocessor with three layers of parallelism and two elements at each layer.

for a single iteration. DPAPP repeats the processes outlined in Section 4.2 and Section 4.4, to obtain prediction event sets and coefficients for each dimension of parallelism. At actuation time, DPAPP makes predictions along each of the dimensions of parallelism and combines these predictions to yield a performance- and power-efficient concurrency operating point for each phase in the program.

4.7 Predictor Optimization

The accuracy of DPAPP is significantly improved by classifying code phases according to their observed $uIPC$ during the execution of the sample configuration. The justification for such an extension is twofold. First, grouping phases based on $uIPC$ allows training and prediction to occur separately for phases with different scalability slopes. As such, the division between buckets is selected such that it divides different degrees of scalability. Second, it is intuitive that the effects of events will vary depending on the original instruction throughput of each phase. Dividing the phases into buckets and creating separate $\alpha(t)$ scaling functions for each class of phases gives the predictor the opportunity to make more fine-grain and accurate predictions. In simpler terms, we train models independently for phases with high throughput at maximum concurrency versus low throughput. At runtime, the observed $uIPC$ on the sample configuration determines which set of coefficients will be used for prediction. We use this optimization in our implementation of DPAPP. Specifically, we divided phases into buckets with $uIPC$ greater than or equal to 1.0 and those less than 1.0 during the sample configuration. This division is not arbitrary, rather, it provides an approximate value to separate phases with low scalability characteristics versus those that scale well, in general, on this architecture. During prediction, each phase uses the coefficients derived from the $uIPC$ bucket corresponding to its observed $uIPC$ during the sample configuration.

4.8 Evaluation of Performance Prediction

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

Experiments were performed with 10 benchmarks that are representative of scientific and engineering applications typically requiring high performance. Nine of the benchmarks originate from the OpenMP version of the NAS Parallel Benchmarks suite, version 3.1 [58]. We use three different problem sizes, available in the NAS distribution (W, A, B). MM5 is an OpenMP implementation of a mesoscale weather prediction model [46]. The benchmarks include a wide variety of program properties, and in particular, widely varying $uIPC$ scalability across execution phases. Therefore, they are challenging targets for prediction. The benchmark suite includes several benchmarks with a small number of iterations (CG, FT, IS, MG), in which empirical search strategies may suffer due to a large percentage of total execution time being spent in exploration, as well as benchmarks with a large number of iterations (BT, LU, LU-HP, SP, UA, MM5), where search strategies stand to have their search overheads better amortized [20]. Results for FT are not included for class size B, because its working set does not fit in the available memory of our hardware platform.

Table 4.1 lists the benchmarks along with some pertinent information about their structure. The number of iterations, phases, and percentage of time spent in parallel regions shown are for class size A. The table also outlines the percentage of execution time during which at least one processor can be deactivated with non-negative impact on performance (i.e. the program runs optimally with at most 3 processors) and percentage of execution time during which one Hyperthread per processor can be deactivated with non-negative impact on performance (i.e. the program run optimally with at most one Hyperthread per processor), averaged over all three class sizes. This information is taken from static executions on all feasible configurations.

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

Table 4.1: The set of benchmarks we used to evaluate online performance predictors for power-performance adaptation, along with their main phase characteristics. % Time Disable CPU/SMT represents the percent of parallel execution time during which at least one element from the respective dimension can be disabled, averaged over all three input sizes.

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

4.8.1 Event Selection

Selection of an effective set of events to use for performance prediction requires data for all of the available hardware event counters on each of the sample configurations for all of the training benchmark phases. Further, the $uIPC$ values of all phases on each hardware configuration are necessary as well. There are 40 events on Pentium 4 processors that can be recorded using only a single register each, with further differentiation within each event through the use a of bitmask parameter specifying, for example, to record L2 cache misses, hits, or accesses. There is also an event to count memory accesses that requires two counter registers. We select one bitmask for each event representing the hardware parameter most likely to have the largest effect on performance, leaving 41 events to consider. Of these, 13 had rates near zero, and were thus removed as described in section 4.4. The performance monitoring unit of the Pentium 4 with Hyperthreading technology shares the 18 counter registers between the two co-executing threads, leaving 9 counters available for each thread.

Regression analysis was performed on the data from each phase to find the events that contributed the most to the resulting IPC prediction. Table 4.2 displays the set of events that was selected for each prediction on our platform. In this discussion, configuration $(nproc, nthr/proc)$ denotes a configuration with $nproc$ processors and $nthr/proc$ threads per processor. It should be pointed out that events with large contributions have been excluded due to conflicts with more dominant events. That is, the inclusion of one highly contributing event often eliminates other contributing events that interfere with it. All that can be done in these cases is to select the event with the largest contribution and ignore the conflicting events. Specifically, three of the top five events on this architecture cannot be included because they conflict with the top two events. This suggests that on architectures where there are reduced or no dependencies between events, our prediction approach will likely achieve higher accuracy.

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

Table 4.2: The Intel Pentium 4 hardware events selected for each prediction type. The second and third columns show the events for predicting the optimal configuration with 2 and 1 Hyperthreads activated per processor respectively.

The events that were selected for prediction make intuitive sense as well. *Cycles Active*, which is the dominant contributor for both predictions made, is a measure of the percent of time during which the processor is *not* halted due to inactivity. *L2 Cache Misses* and *Branches Retired* also contributed greatly as they have clear effects on performance. *TC Deliver Mode* is the percent of cycles that the trace cache was in deliver mode, approximating the hit rate of the in-

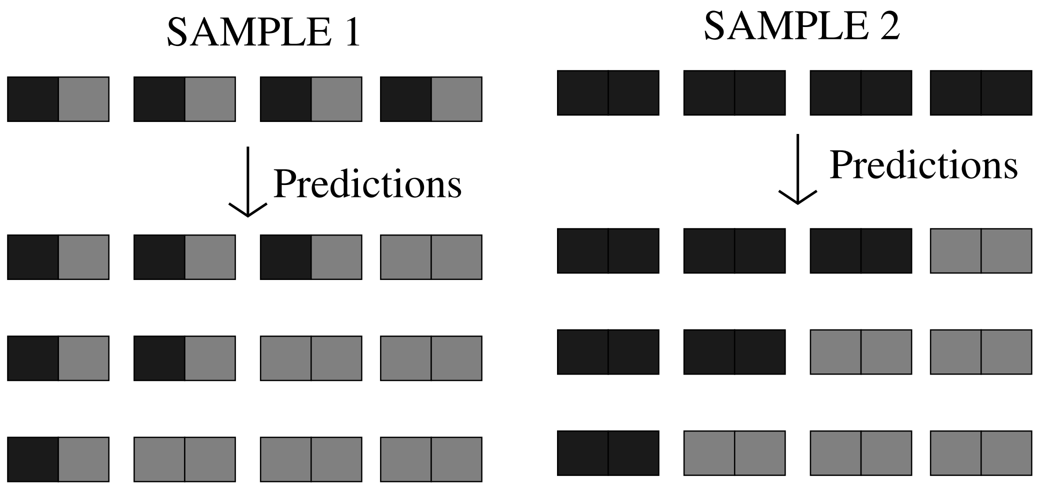


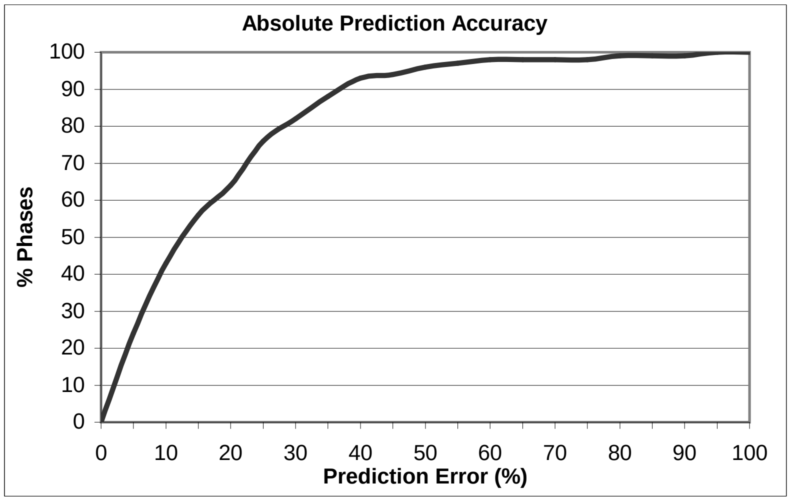
Figure 4.2: Specific configurations sampled and predicted. Actively used contexts are shown in black and idle cores shown in light grey.

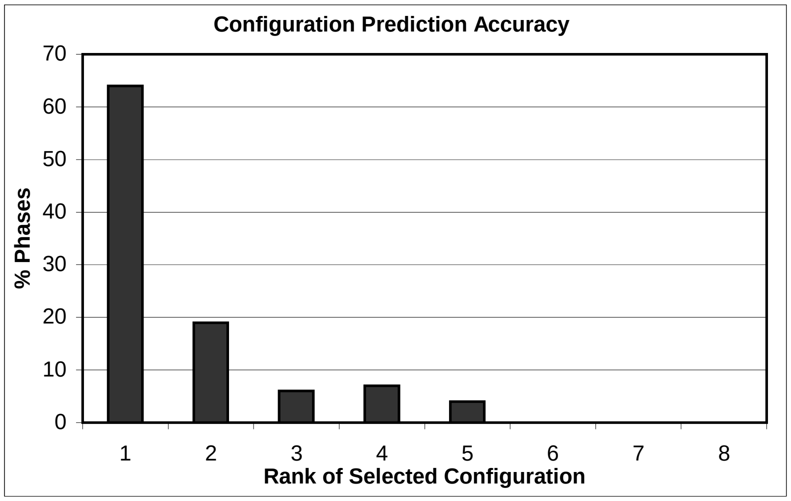
struction cache. *UOP Queue Writes* counts the number of instructions written to the instruction queue and represents an alternative measure of throughput to IPC. Accesses to memory that are canceled for various architectural reasons are recorded with *Memory Cancel*. *Packed SP/DP UOPs* reports the use of SIMD instructions with single and double precision. *Machine Clears* reports the number of pipeline flushes incurred due to memory ordering issues. The number of cycles during which no instructions can be executed is given by *Stall Cycles*. Finally, *Memory Accesses* requires two registers for recording of the number of memory accesses made by a region of code. This last event is only used by the two threads configuration prediction, as memory accesses can lead to increased contention in the shared cache.

4.8.2 Prediction Accuracy

We perform our evaluation of the accuracy of the online performance predictor using eight of our ten benchmarks, excluding the two benchmarks used for training the predictor. We consider the absolute prediction error and the configuration prediction error for each benchmark. We calculate the absolute prediction error as $|uIPC_{pred} - uIPC_{obs}|/uIPC_{obs}$, where $uIPC_{obs}$ is the observed IPC of useful instructions. On our experimental platform, there are six predictions made for each phase. Specifically, we predict for configurations with one and two Hyperthreads per processor on one, two, and three processors as shown in Figure 4.2. The average prediction error for each phase is taken across all target configuration predictions. Configuration prediction accuracy compares the predicted optimal configuration for each phase with the *local static optimal configuration*. The local static optimal configuration is obtained as follows: We execute the benchmarks with each of the eight possible hardware configurations statically, i.e. with no concurrency throttling between phases. For each phase, we designate as optimal the one configuration out of the eight possible that minimizes the execution time of the phase. Note that this definition of optimal configuration ignores inter-phase interference and that a local static optimal configuration may or may not be the global, program-wide static optimal configuration. Configuration prediction accuracy illustrates how often the predictor identifies the local static optimal configuration. It should be noted that $uIPC$ prediction is particularly challenging on our experimental platform, because often, $uIPC$ changes due to Hyperthreading cannot be approximated as a linear function of the number of processors and threads used. However, we should also note that the litmus test for our predictor is not $uIPC$ prediction accuracy but configuration prediction accuracy. As long as the predictor correctly predicts the optimal configuration for each phase, a potentially high $uIPC$ prediction error can be disregarded.

The $uIPC$ prediction accuracy can be seen in the top graph of Figure 4.3. This graph gives the cumulative distribution function of prediction error, that is, the percent of phases that experience error below each threshold with threshold samples taken every 5%. DPAPP achieves a median error of only 12.9% overall, with some applications seeing median errors below 5% across all predictions made. We note that 24% of all predictions have less than 5% error and 43% of all predictions have less than 10% error. On the other hand, only 4% of the predictions show error larger than 50%. Although our performance prediction model is purposefully simple to minimize the overhead of applying it at runtime, its results compare favorably with other reported statistical techniques for predicting IPC [33].





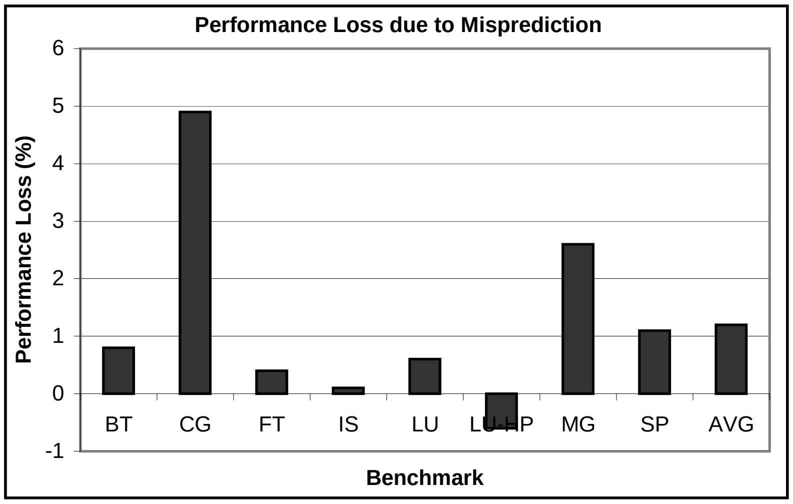


Figure 4.3: The top chart illustrates the CDF of prediction error. The middle chart illustrates the percent of phases for which each rank of configuration was selected. The rank of the selected configuration is taken from the list of configurations sorted by their IPCs on static executions of each phase, a value of 1 indicates that the optimal configuration was selected. The bottom chart shows the performance loss (>0) or gain (<0) resulting from configuration misprediction.

In terms of prediction of the optimal configuration for each phase, the middle chart of Figure 4.3 shows the percent of phases for which each possible ranking of configuration was selected. This value is calculated by sorting the configurations by IPC for each phase and identifying which entry was selected by the predictor. For example, a value of 1 indicates that the best configuration was selected and 2 indicates that the second best configuration was selected. This graph shows that in 64% of cases the single best configuration is identified by the predictor. An additional 19% of phases have the second best possible configuration selected.

As a result of the high configuration prediction accuracy, the performance loss in mispredicted regions is usually quite low. The bottom chart of Figure 4.3 shows the weighted performance loss observed for each benchmark during mispredicted phases. This value is calculated as $\sum_{i=1}^{N_B} w_i \cdot D_i$, where N_B is the number of mispredicted regions in benchmark B, w_i is the weight of each mispredicted region expressed as the percentage of the total parallel execution time of B that the specific region accounts for, and D_i is the absolute performance penalty suffered by the mispredicted region i. The average penalty across benchmarks is only 1.2%. The explanation for the negative performance loss (performance gain) of LU-HP is that by not changing configurations to the local optimal in all cases, the cache effects of altering configurations are reduced. These results show that our model is capable of identifying optimal configurations most of the time, and when it does not it still manages to find a competitive configuration to use, with minimal performance penalty.

4.9 Summary

In this chapter, we have presented a novel scalability prediction model of parallel scientific applications executing on multithreaded and multicore architectures. Specifically, the dynamic phase-aware performance predictor (DPAPP) works by applying multivariate regression analysis to hardware event rates collected at maximum concurrency to predict the degree to which changing concurrency and thread placement will reduce contention and affect performance.

The model is developed through an offline training process during which regression is used to capture the effects of varying event rates on scalability, so that the model can be applied online to previously unseen phases to accurately predict performance across threading configurations. Essentially, the relationship between particular event rates and scalability are “learned” offline, and then applied to individual phases online.

We evaluated DPAPP and found that it is able to characterize the performance and scalability of program phases with high accuracy on a four processor Intel Hyperthreaded system. Our predictor allows for the online identification of performance- and energy-effective concurrency levels and thread placements, while keeping the overhead at manageable levels. Over a range of multithreaded scientific benchmarks, the predictor was shown to be quite effective at locating the optimal configuration for each phase, due to a low median error of 12.6%. The high accuracy combined with low overhead makes DPAPP very well suited for use with dynamic concurrency throttling. In the next Chapter we discuss how we apply DCT based on the DPAPP model.

This page intentionally left blank.

Chapter 5

Dynamic Concurrency Throttling

The continued push to ever higher degrees of parallelism within a single chip is providing much potential for large performance gains from applications that are well suited for the architecture. However, as pointed out in Chapter 1, many parallel scientific applications fail to scale beyond a certain degree of concurrency on these systems due to contention over shared resources. Given the limited scalability observed, we extend the per-chip concurrency throttling discussed in Chapter 3 to occur system-wide and using performance prediction rather than direct search. That is, we develop a runtime system that exploits our DPAPP scalability model from Chapter 4 to identify the performance-optimal number of threads to use in a parallel application dynamically as well as the particular placement of those threads onto processing elements. By responding to limited scalability in such a way, considerable performance gains are possible because concurrency can be set at the knee of the scalability curve.

Dynamic concurrency throttling (DCT) also has interesting properties with respect to power management. In principle, DCT and dynamic voltage and frequency scaling (DVFS) are two techniques for reducing processor power consumption in software. By throttling processors, software can reduce both the dynamic and the static power consumption of the system at a faster rate than with DVFS. DVFS mainly targets dynamic power consumption. As the relative weight of static power consumption (due to leakage current) on future processors is expected to increase, the potential of DVFS for reducing power without performance penalty may diminish, whereas concurrency throttling techniques may still achieve substantial power savings [30].

Optimizing parallel codes for concurrency throttling can be achieved with compiler [63] or runtime phase analysis, using either direct search algorithms [25, 83] or performance prediction across system and program configurations with varying degrees of concurrency [21, 24]. Compiler methods are effective for codes with simple memory access and thread execution patterns, but they are constrained by well-known limitations of compiler analysis and compiler-based performance prediction. Runtime search methods can discover optimal or nearly-optimal concurrency levels for phases of parallel code separated by synchronization or communication operations. However, search methods may require a large number of executions of a phase to converge to an optimal operating point. In particular, the number of executions depends on both the number of processors and cores and the actual topology of processors and cores on the system [21]. The topology of processors is important as different mappings of a given number of threads on a given topology may yield dramatic performance variation. With tiled embedded processors with 64 to 512 cores, such as Tilera's Tile64 [114] and Rapport's Kilocore [81] already on the market, exhaustive or heuristic search of program and system configurations may become prohibitively expensive.

Runtime performance prediction overcomes the limitations of direct search methods at the potential cost of reduced accuracy in identifying optimal operating points. These approaches test fewer configurations, thus reducing online overhead. However, their efficacy depends on their prediction accuracy. We present our phase-aware concurrency throttling algorithm for a shared-memory multiprocessor, such as a multi-chip multiprocessor with multicore processors, as well as its implementation in a runtime library called ACTOR (for *Adaptive Concurrency Throttling Optimization Runtime system*). We show that our runtime system can effective identify improved concurrency levels and thread placements to achieve significant improvements in performance and power consumption simultaneously.

In the next section, we present the general structure of the ACTOR runtime system. In Section 5.2, we discuss the source of energy savings that are possible using ACTOR. We then present an extension to the basic adaptation approach whereby interphase interaction can be considered to reduce interference that occurs while performing DCT in Section 5.3. Section 5.4 provides details on the implementation of the ACTOR system including program instrumentation and transparent application controlling. In Section 5.5, we present the empirical results of performing dynamic concurrency throttling through the exploitation of the DPAPP scalability model, in terms of both performance and power consumption. Section 5.6 provides a summary of this chapter.

5.1 ACTOR Runtime System

Scientific codes are dominated by iterative execution of phases and ACTOR exploits this property to sample hardware event rates in the first few phase traversals and set the concurrency of each phase to the predicted optimal operating point, early during execution of the program. The live search of the optimization space for operating points of concurrency can also be performed by timing phases at different configurations and running search heuristics such as greedy hill-climbing [25, 83] or simulated annealing [69]. However, as the number of feasible hardware configurations increases with the introduction of more cores and threads per processor, direct search methods may spend most of the execution time sampling suboptimal configurations, rather than optimizing the program. This disadvantage manifests itself in codes where dominant multithreaded phases are traversed only a few times. Even if direct search methods are used for off-line auto-tuning by repetitive executions of the entire program [4], searching the optimization space for any input on any feasible configuration of processing units may be prohibitive. ACTOR prunes the search space for concurrency optimization to a constant number of samples by using performance prediction.

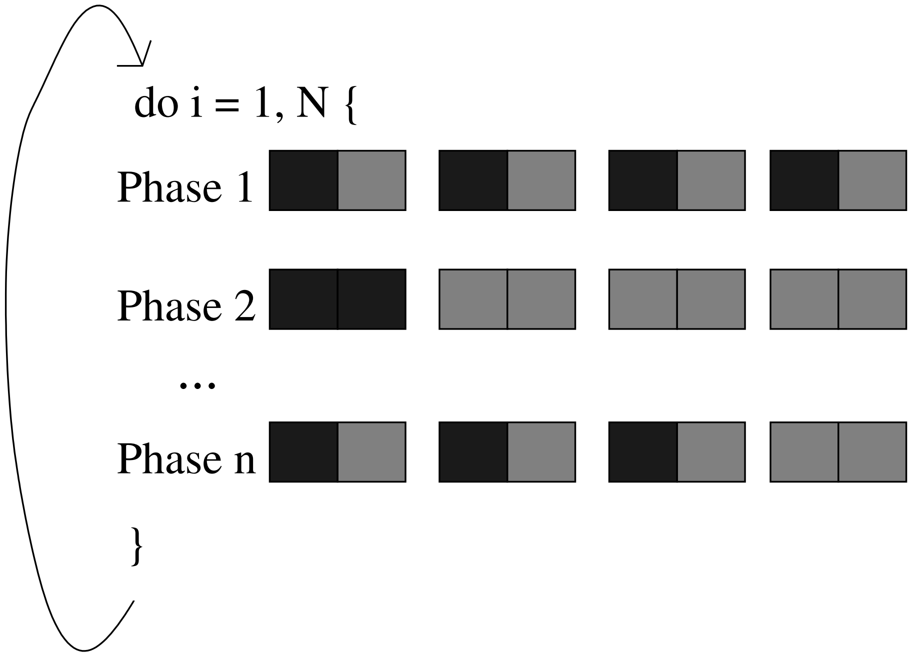


Figure 5.1: Example execution using DCT. Each phase uses the locally optimal number and binding of threads to processing elements. Black squares represent active threads and grey square are inactive.

Figure 5.1 presents an example of a standard parallel application execution using dynamic concurrency throttling on a four processor dual-SMT processor. Processing elements are represented as squares within the rectangular processors; black represents actively used threads and grey represents inactive threads. The diagram shows the iterative structure of the majority of parallel scientific and engineering applications, with multiple independent phases executed each iteration of an outermost loop. Also demonstrated is that concurrency decisions are made for each phase, allowing optimal concurrency to be identified and exploited for each program phase.

Figure 5.2 illustrates a DPAPP-driven concurrency throttling algorithm in ACTOR for a multiprocessor with two dimensions of parallelism. The DPAPP-based concurrency throttling algorithm has two parameters, the sampling rate and the dimension of parallelism along which the initial samples are taken. The sampling rate, S, corresponds to the number of times each phase needs to be executed before deriving a prediction for the optimal operating point and is

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

Figure 5.2: DPAPP-driven concurrency throttling algorithm for an architecture with 2-dimensional parallelism.

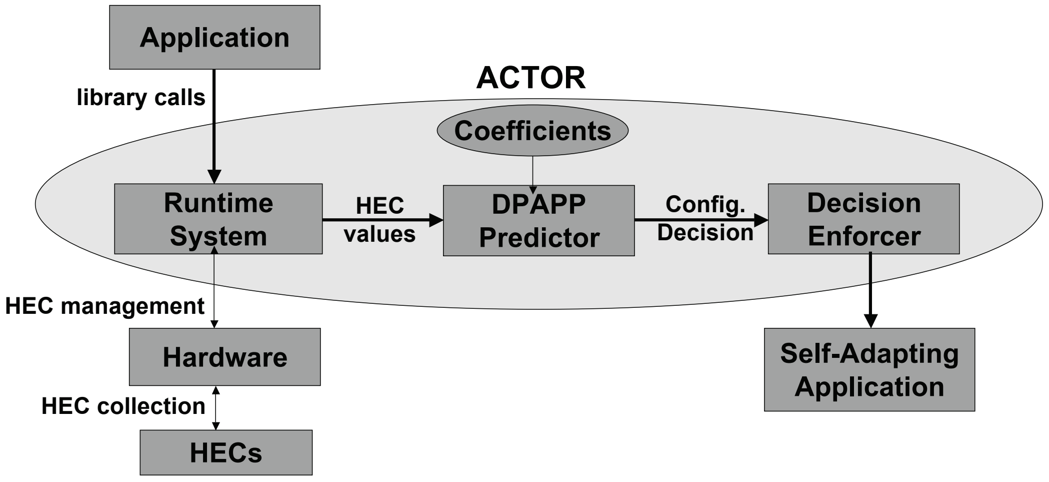


Figure 5.3: The overall structure of the ACTOR runtime system.

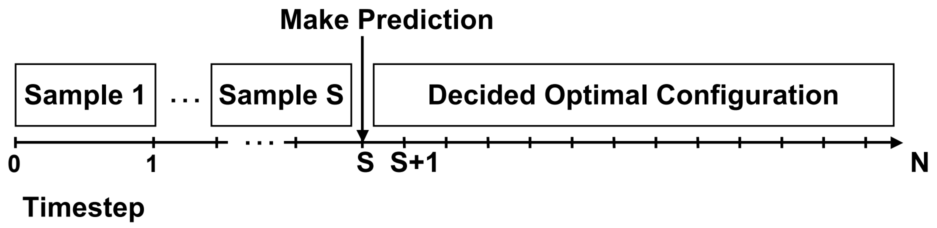


Figure 5.4: Sampling, prediction, and execution timeline for each phase.

used to control the sampling overhead. In our prototype, we use a sample rate of $S = 2$ taken along the innermost dimension of parallelism, i.e. threads within a processor, which provides the minimum number of samples needed to capture the effects of using more than one core or thread per processor. The second parameter is fixed at the training phase of the DPAPP predictor, during which all possible orderings of dimensions of parallelism can be tested. The algorithm in Figure 5.2 generalizes to more than two dimensions by repeating the loop in lines (11)–(17) for each dimension beyond the second.

The structure of the ACTOR system is given in Figure 5.3. The controller is dynamic, in the sense that it adapts the program as it executes, with no prior knowledge of program characteristics. Currently, ACTOR requires simple, formulaic instrumentation in the application, however this instrumentation could easily be provided within a simple preprocessor or OpenMP compiler. ACTOR estimates optimal operating points of concurrency using samples of critical hardware event rates from live executions of program phases. Specifically, the library controls the first S phase traversals to execute on the desired sample configurations and collect event rates, as shown in Figure 5.4. At the end of each sample, collected event rates are used by DPAPP to predict the $uIPC$ of each phase on alternative configurations. Once predictions for a phase are obtained, all subsequent traversals of a phase are executed at the predicted optimal operating point of concurrency. ACTOR enforces configuration decisions through the Linux processor affinity system call, *sched_setaffinity()*, and threading library specific calls for changing concurrency levels, such as *omp_set_num_threads()* in OpenMP. The library executes at user-level and so does not require administrator privileges. The overhead of using ACTOR in terms of the time spent not executing application code is approximately five hundred thousand cycles per program phase (250 microseconds on a 2 GHz processor), which is negligible for any realistic application.

While both concurrency throttling and DVFS target improved energy-efficiency, concurrency throttling has the advantage that it will often improve performance, whereas DVFS generally sacrifices performance to reduce power consumption. Further, DVFS relies on program phases with high memory access rates to avoid degrading performance significantly, while concurrency throttling may be applied in other cases as well. In general, however, the two approaches are likely to be highly synergistic and can be applied together to achieve even greater energy-efficiency. For example, DVFS could be applied using existing approaches to cores kept active by concurrency throttling. More sophisticated techniques could be devised to optimize both DVFS and concurrency, and we consider such approaches in Chapter 6.

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

While we have specifically designed ACTOR for use with iterative scientific applications, the approach may apply to other categories of applications as well. The basic principle of ACTOR can be used with any definition of a phase where concurrency can be dynamically adjusted. For example, in non-iterative, synchronization-intensive, or heterogenous multithreaded codes, if an existing phase identification technique can be employed to identify repetitive behavior where concurrency is modifiable, then our approach can be applied. For server workloads the application may be treated as one large phase and a limited timeframe can be monitored to decide concurrency for the entire application.

5.2 Energy Savings Possibilities

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

5.3 Cross-Phase Decision Making

The processes of prediction, decision making, and adaptation are not performed at whole-program granularity, rather, each phase of an application is analyzed independently. This allows phases with different execution properties in the same application to execute with their own, locally optimal hardware configurations. Since many programs have behavior that varies across phases [102], overall performance can be improved compared to using a single configuration for the entire program. However, a non-negligible performance penalty may be paid as a result of changing the hardware configuration across adjacent phases at runtime. This performance penalty stems primarily from migration of working sets of threads between caches [65]. To avoid negative inter-phase interference, we consider variants of our adaption scheme that are aware of this interference.

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

5.4 Adaptation Runtime System Design and Implementation

We have implemented a full prototype of our concurrency throttling approach in the *ACTOR* (Adaptive Concurrency Throttling Optimization Runtime) runtime system. The runtime system controls OpenMP codes at the granularity of parallel loops and regions. ACTOR uses PACMAN [19], a customized, thread-aware performance event monitoring library for Intel processors that we wrote to overcome limitations in all other existing HEC collection libraries.

5.4.1 Program Instrumentation for Adaptation

The runtime adaptation library that we describe has been designed to operate on codes that have been parallelized using OpenMP [96]. OpenMP codes have the property that parallel regions are demarcated with directives serving to notify the compiler that a given section of code can be executed in parallel by multiple threads. Our runtime system exploits the information provided by these directives to perform adaptation at the granularity of OpenMP parallel regions, essentially taking parallel regions to serve as program phases for our purposes. While it is true that loops, or perhaps an even finer granularity, might better serve as program phases in general, parallel regions in OpenMP tend to have stable execution characteristics and they are the finest granularity at which the number of threads can be changed.

In order for an application to make use of ACTOR, a few calls must first be inserted into the original code. One of our primary goals was to limit the intrusion on the programmer necessary to instrument the code, so we have made every effort to simplify this process as much as possible. In addition to being simple, the instrumentation has to ensure that parallelization code as well as synchronization instructions at barriers are not included in the monitored regions. Figure 5.5 presents an example parallel region that has been instrumented with hooks into our runtime system, with modifications to the original source code shown in boldface.

This sample code shows all of the necessary modifications. Calls to *start_region()* and *stop_region()* delimit the boundaries of a parallel region and allow ACTOR to keep track of

```
1 call start_region(1)
2 !$omp parallel default(shared) private(i,j,k)
3 call start_loop()
4 !$omp do
5 do k = 1, d(3)
6     do j = 1, d(2)
7         do i = 1, d(1)
8             u1(i,j,k) = u0(i,j,k)*ex(t*indexmap(i,j,k))
9         end do
10    end do
11 end do
12 !$omp end do nowait
13 call stop_loop()
14 !$omp barrier
15 !$omp master
16 call accumulate()
17 !$omp end master
18 !$omp end parallel
19 call stop_region(1)
```

Figure 5.5: Example parallel region from the FT application of the NAS parallel benchmarks suite. The additions/modifications to the code required to activate and use our adaptation runtime system are shown in boldface.

phase specific information, including the number of threads and thread bindings to use during exploration and enforcement iterations. Within each phase, calls to *start_loop()* and *stop_loop()* control the starting and stopping of performance counter collection during the exploration iterations. All counters collected by each thread are tallied in *accumulate()* after synchronization has occurred. Parallelization code is not monitored because *start_loop()* is called after the *parallel* directive and synchronization is avoided because *stop_loop()* is called before synchronization is performed at the end of the parallel loop (because of the *nowait* added to line 12 specifying that synchronization should not be performed there). As some parallel regions contain multiple parallel loops, we have allowed multiple *start_loop()-stop_loop()-accumulate()* sequences to occur within a single region while still avoiding synchronization costs between multiple loops in a parallel region. The formulaic structure of these calls makes instrumentation easy even with no previous knowledge of the source code, and a candidate for automation by a compiler.

5.4.2 Thread Control Details

An important part of the functionality of the adaptation library is its support for autonomous management of executing threads. ACTOR controls the number of threads that are used to execute each phase as well as the bindings of these threads to specific processing elements during both the configuration exploration and decision enforcement stages. The number of threads used is set with the *omp_set_num_threads()* OpenMP library call, which specifies the number of threads that will be used for all subsequently executed parallel regions (until the function is called with a different value). Binding of threads is facilitated with the *sched_set_affinity()* Linux system call, which specifies on which processing elements a given thread is allowed to run. These functions are called from within *start_region()* ACTOR function, which is made directly before each phase executes, to set the number of threads and specific bindings for the upcoming phase. Within the *start_region()* function, a parallel region is created with the desired number of threads where each thread simply binds itself to the desired processor. This way, each thread will be bound correctly upon executing the parallel region. The specific binding scheme used has been designed to minimize any negative effects in the cache of changing the number of threads and processing elements, wherever possible.

First, individual threads are bound to specific processing elements to avoid uncontrolled thread migration, thereby maximizing cache warmth when no concurrency changes are made. Second, specific bindings for each configuration have been selected such that the maximum number of threads remain executing on the same processing element. Threads are assigned such that changes in the number of threads without changing the number of processors or cores (the most common case) disturb none of the already active threads. Changes in the number of cores and/or processors minimizes interference similarly, though to a larger degree. Specifically, threads are bound in increasing order of thread number and spread evenly over all layers of parallelism, starting with processors and working down. An example binding is shown in Figure 5.6.

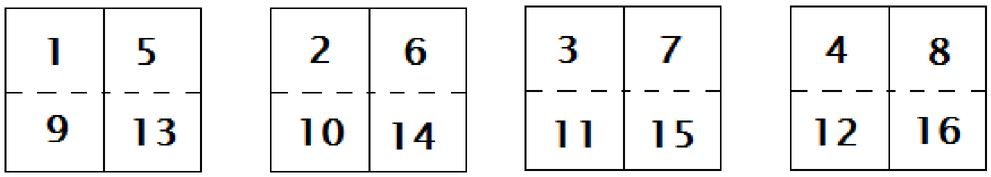


Figure 5.6: Example bindings of threads by number on a 4-way SMP of dualcore 2-way SMTs.

5.4.3 Performance Counter Collection

To provide access to the hardware performance monitors, we used PACMAN [19] (PerformAnce Counters MANager), a library that provides low-overhead access to the performance monitoring hardware. PACMAN is built on top of lower-level libraries such as Perfctr [98] on Intel Pentium processors. On Intel Hyperthreaded processors, the performance monitor unit is shared between co-executing threads. To allow each thread to execute with its own set of counters, PACMAN partitions the counter registers between the threads. Further, PACMAN removes the restriction from Perfctr that disallows the use of the second execution context on each processor while performance counters are active. Counters are collected using per-thread collection because it allows a much finer granularity and does not record events incurred by other executing processes, allowing more accurate information to be used for prediction. PACMAN is described in more detailed in Appendix A.

5.5 Adaptive Concurrency Throttling Evaluation

Evaluation of concurrency throttling was performed using the experimental platform and benchmarks as used for performance prediction in Section 4.8.2. To measure the power consumption of the benchmarks under various hardware configurations we utilize a power measurement methodology based on hardware event counters [57] that has proven to be highly accurate. This methodology works by first partitioning the processor into components and then determining the maximum power consumption of each component based on the die area it consumes. The runtime power consumption of each component is the maximum power adjusted by an activity factor. The latter is estimated by looking at corresponding hardware event counters. This amount is added to a non-gated clock power associated with each component that grows non-linearly with activity. Finally, the power consumptions of all components are summed along with a constant base idle power. It should be noted that we focus only on processor power consumption. For the well-tuned scientific applications we consider in this work, processor power is the dominant portion of the total system power consumption [101].

5.5.1 Motivating Examples

Figure 5.7 depicts the execution times and energy consumption of each benchmark under class size A for each static configuration. Static configurations use a single configuration for the entire execution. These graphs show that on our experimental platform, very little additional performance gain is seen through adding additional processors once two processors are active. Particularly interesting is the IS benchmark, which sees its best performance using a single thread on only one processor. Further, sometimes there is a large gain through using the second execution context on each processor, and sometimes a substantial loss. It can be observed that while performance levels out, the energy consumption increases at rather steep rates with more processors.

The reader may note that the observed scalability bottlenecks are an artifact of hardware bottlenecks, such as limited memory bandwidth. While this statement is correct, it also reflects a property of a large number of real systems, including state-of-the-art platforms that outdate our experimental system. For example, we performed experiments with the NAS benchmarks on a newly released quad-core Intel Xeon processor (QX 6700) that have shown that applications still tend not to scale well on even the latest hardware. In particular, several of the benchmarks fail to scale beyond two cores, with maximum speedups saturating well below 2 (see Figure 5.8). As a result, opportunities for concurrency throttling still exist even in the newest hardware platforms.

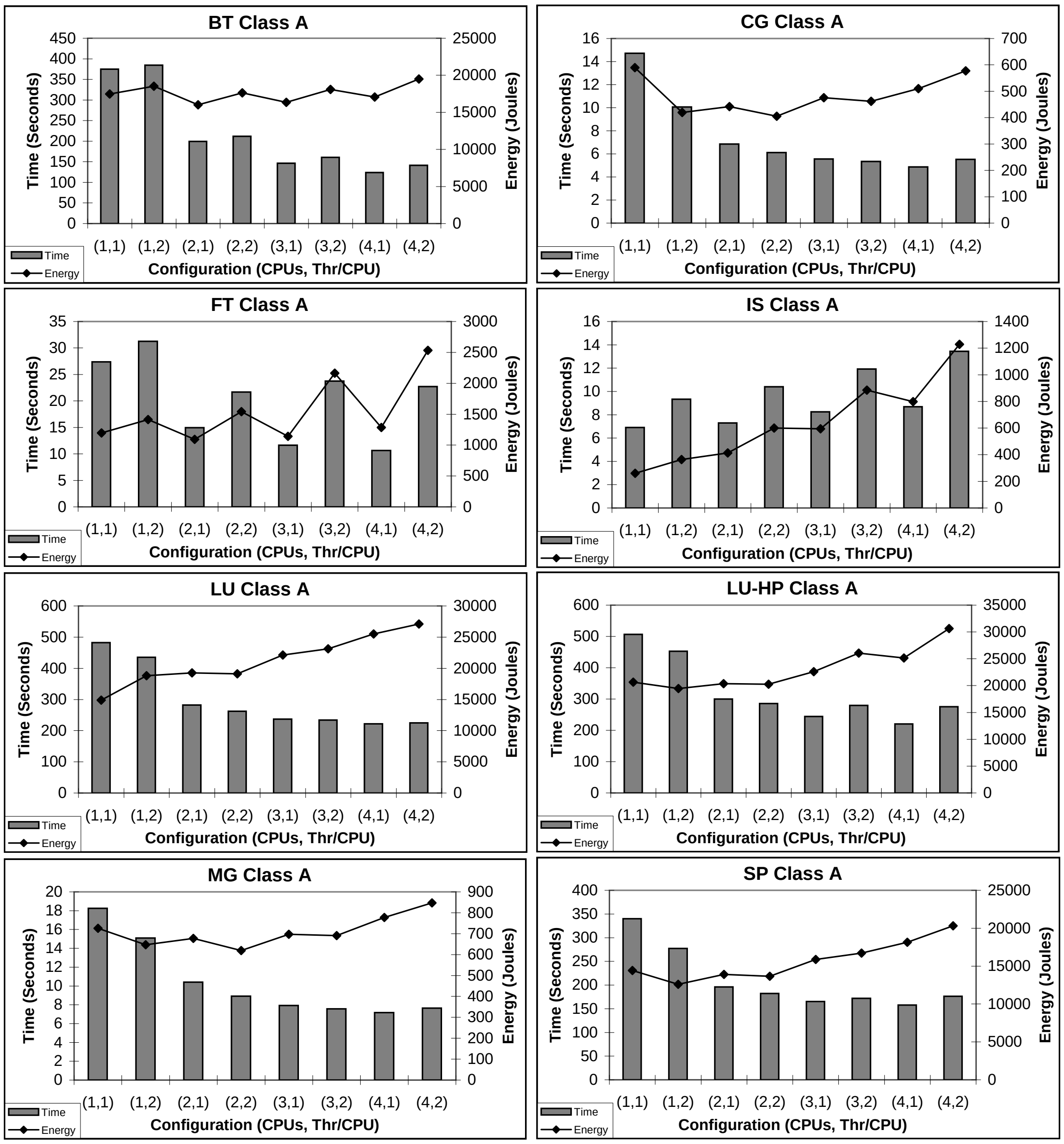


Figure 5.7: The execution times and energy consumption of each static configuration.

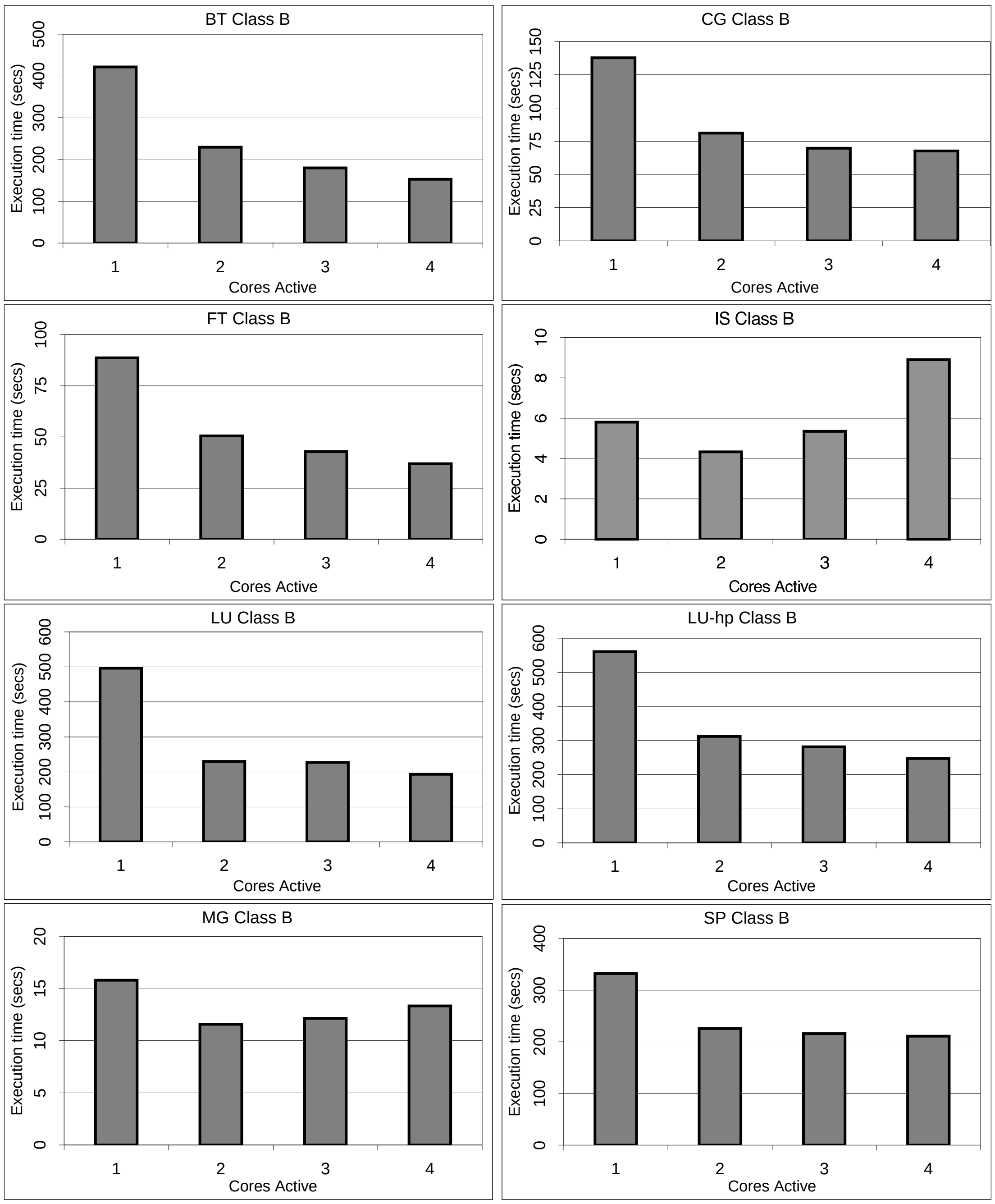


Figure 5.8: Scalability characteristics of the NAS benchmarks on a state-of-the-art quad-core Intel processor.

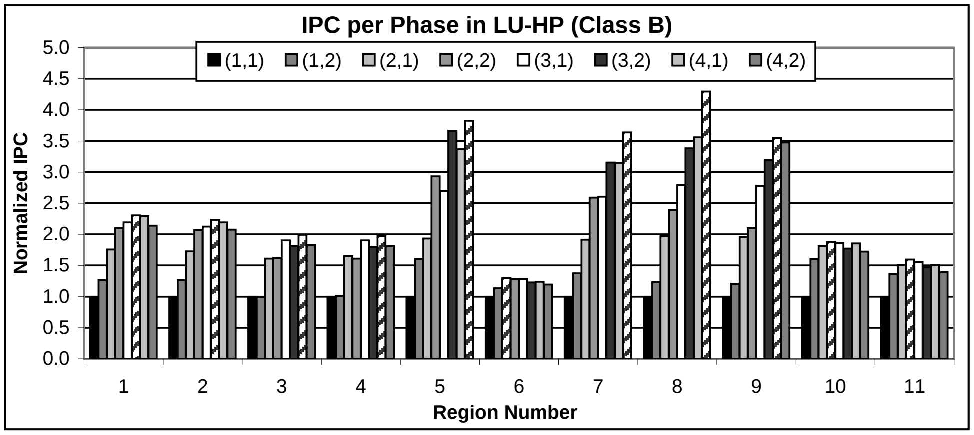


Figure 5.9: IPCs for each phase of the LU-HP benchmark under each static configuration, normalized by the IPC on (1,1). The best performing configuration for each phase is marked with stripes.

As further evidence of the importance of phase-level adaptation, Figure 5.9 displays the IPCs for each phase of the LU-HP benchmark at class size B under each static configuration normalized by the IPC of (1,1). It is evident from the chart that a single application can have optimal configurations varying greatly between phases. LU-HP in particular experiences five different optimal configurations across different phases, specifically (2,1), (2,2), (3,2), (4,1), and (4,2). Therefore, using a technique to execute each phase at its local optimal operating point stands to improve performance. In cases where the optimal configuration occurs on fewer than the available number of processing elements, power savings can occur during the execution of these phases. The goal of our adaptation approach is to exploit these properties with no *a priori* knowledge of the codes and achieve both power and performance benefits.

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

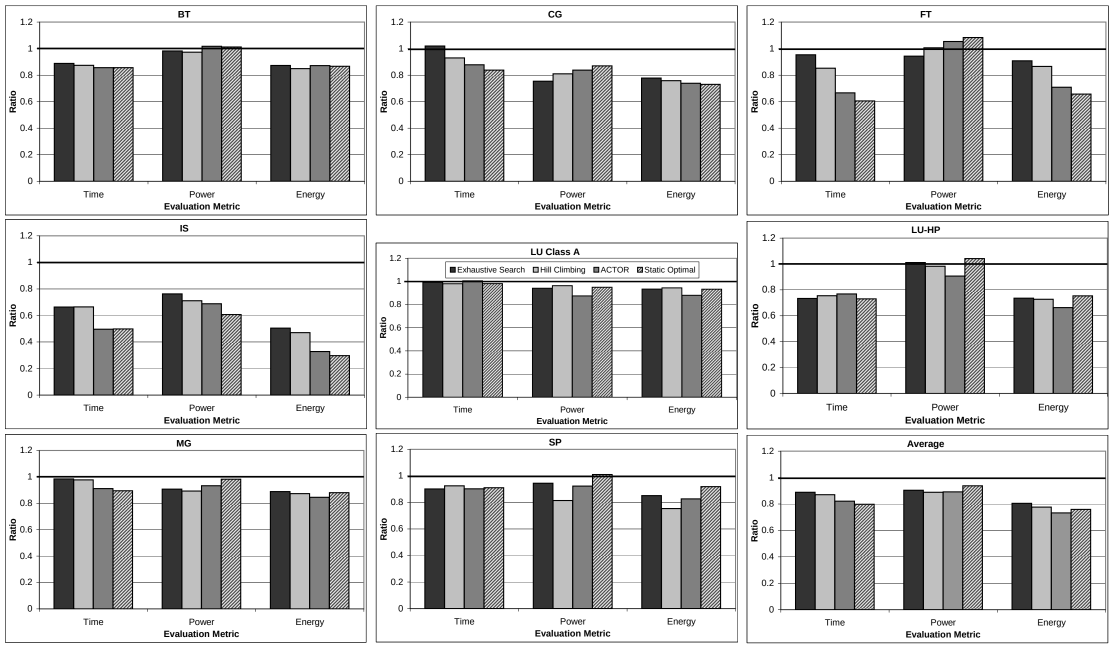


Figure 5.10: Performance of the adaptation strategies in terms of execution time, power, and energy for all evaluation benchmarks.

than *static optimal*. We limit our following evaluation to comparing adaptive strategies to *static optimal*.

The two offline approaches that we consider have the disadvantage that the optimal configuration may change with different input sizes. For example, IS executes statically optimally on (3,1) for class size W, but (1,1) and (2,1) for class sizes A and B respectively. For individual phases, the optimal configuration varies by problem size as well. Specifically, only 52.5% of the program phases in our benchmarks experience the same optimal configuration regardless of input size. This means that use of these static techniques requires offline analysis that is specific to the application and the input size. By contrast, the online adaptive approaches adapt autonomically at runtime for the current application execution and require no application/input-size specific offline analysis.

5.5.2 Empirical Search-based Strategies

For purposes of comparison, we have implemented two alternative dynamic adaptation strategies based on empirical search of the configuration space at runtime. The first of these is the most straightforward form of adaptation, exhaustive search, where each possible configuration is tested and the one that provides the lowest execution time is selected for each phase. Figure 5.10 illustrates the normalized arithmetic means of three metrics: execution time, average power consumption during execution, and energy consumption. These metrics are derived for each benchmark under different execution strategies. Each metric is first normalized to the corresponding metric of the (4,2) configuration for the specific benchmark, which exploits all available execution contexts on our experimental platform. We then calculate the arithmetic means of the normalized metrics.

Occasionally, the power consumption actually *increases* through the use of adaptation. This result is counterintuitive since adaptation is always expected to either keep the number of processors and Hyperthreads used constant, or reduce it. Recall that our starting assumption for adaptation is that deactivating threads inside a processor reduces power. This is actually true in the majority of phases, specifically 79% of all phases. However, in certain cases, the use of Hyperthreading introduces long stall times in the processor, due to contention for shared resources, and therefore long periods of processor inactivity, during which power consumption is low. By contrast, deactivating Hyperthreading in these situations, reduces stall times, increases processor utilization and therefore increases the average power consumption. However, overall energy consumption is still reduced, due to the reduction in execution time experienced in these cases. Therefore, the overall result is positive for applications where the Hyperthreading anomaly occurs.

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

The second empirical search technique that we implemented is a heuristic search algorithm, which we have previously devised to reduce the overhead of exhaustive search [25]. This algorithm works by applying a hill-climbing heuristic search to find the optimal number of processing elements to use at each dimension of parallelism, one dimension at a time. The algorithm begins by executing the phase on all available processors with all Hyperthreads active. Then, the number of processors is successively reduced until an increase in execution time is observed. The lowest number of processors that results in a decrease in execution time is used for the corresponding phase. This process is then repeated on the decided upon number of processors to determine the number of Hyperthreads to use on each processor.

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

5.5.3 Performance Prediction-based Adaptation

Through the use of performance prediction, the number of iterations required for adaptation can be further reduced using the algorithm presented in Section 5, to only two iterations in the case of our experimental platform, thereby minimizing overhead. Further, performance prediction reduces the effects due to changing configurations during the exploration process that can lead to suboptimal decisions by the direct-search strategies.

As discussed in Section 5.3, changes in the hardware configuration used to execute an application can result in unintended cache effects that end up hurting performance [65]. To quantify the performance degradation of cache locality distortion caused by dynamic concurrency throttling, we first executed LU-HP Class A under all eight static configurations. We identified the optimal configuration for each phase and found the execution time of each under its optimal configuration. We then executed the application with the optimal configuration used for each phase and found a 19% average performance loss compared to the static execution. This loss can only be attributed to the dynamic variation of the configuration used for execution.

We first compare a strategy whereby the predicted optimal configuration for each phase is used blindly, to strategies that consider cross-phase analysis to make decisions. The best strategy is selected for use with ACTOR, and is compared to the offline and direct-search approaches

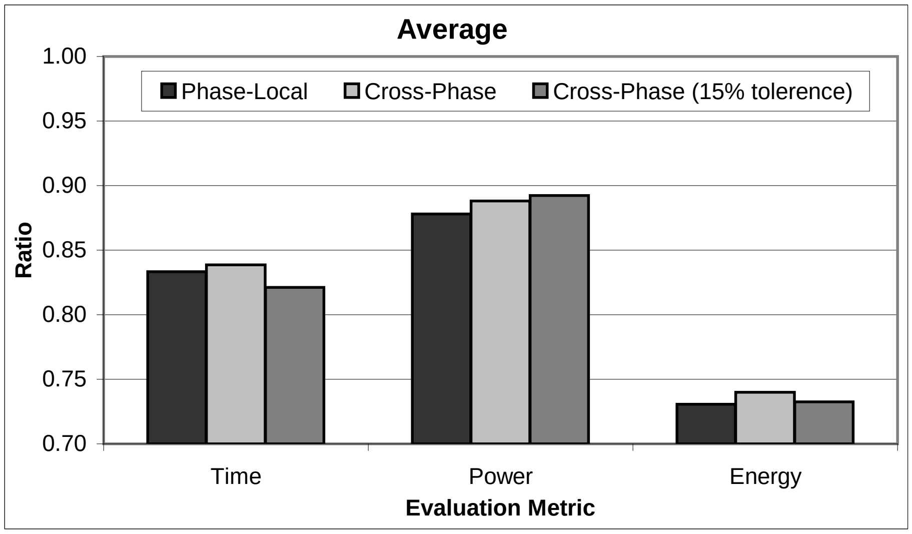


Figure 5.11: Execution time, power, and energy effects of utilizing the three different prediction based adaptation strategies, with all numbers normalized with respect to the (4,2) execution.

already presented. First, we evaluate our approaches to minimize the harmful effects of using the local optimal configuration for each phase, which occur if changes in the configuration of adjacent phases result in redistribution of working sets between caches [65]. We compare the results to the greedy local optimal approach to find the best prediction-based adaptation approach. Our experimental results, shown in Figure 5.11, indicate that simply attempting to avoid cache interference is not inherently effective. Using an approach whereby the configuration selected as the best for the majority of execution time (i.e., the dominant configuration) is enforced for all phases produces a slowdown of 1.5% compared to the local optimal approach, with an additional 0.9% energy consumption. This happens because, in many cases, the benefits of executing a phase with its optimal configuration outweigh the loss suffered as a result of cross-phase interference.

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

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

The power-related results for ACTOR are just as substantial as those for performance. Energy consumption is the product of power consumption and execution time, and concurrency throttling attempts to reduce both, decreasing energy consumption by a still larger margin. We observe 10.8% and 26.7% reductions in power and energy consumption, respectively, compared to using all execution contexts. When compared to using the static optimal configuration, a 2.9% average reduction in power is seen and a 0.9% reduction in energy. This result may seem surprising, however it can be explained by the fact that the static optimal uses only a sin-

	All Procs	Static Opt	Phase Opt	Exh Search	Hill Climbing	ACTOR
BT	3.99	3.96	3.93	3.80	3.57	3.99
CG	3.69	3.01	3.39	2.45	2.61	2.86
FT	3.22	3.42	3.50	3.02	3.28	3.48
IS	3.05	1.12	0.76	1.82	1.45	1.42
LU	3.99	3.99	4.00	3.91	4.00	3.34
LU-HP	4.00	3.99	3.72	3.77	3.59	3.28
MG	3.61	3.51	3.47	3.16	2.90	3.24
SP	4.00	3.99	3.52	3.64	2.79	3.45
AVERAGE	3.69	3.37	3.29	3.20	3.02	3.13

Table 5.1: The average number of processors kept active for each benchmark under each execution strategy.

gle configuration for the entire program execution, rather than further decreasing the number of active processors for individual phases below the global optimal level. ACTOR also sees a 1.1% reduction and a 0.8% increase in power consumption compared to exhaustive search and hill climbing respectively However, ACTOR reduces total energy consumption by 10.2% and 6.3% because of its performance advantages. These results indicate that prediction-based adaptation is able to make effective decisions, both in terms of improving execution time and in terms of reducing energy consumption.

As a further comparison between the alternative approaches, we consider the average number of processors used (versus deactivated) under each strategy, as shown in Table 5.1. The explanation for why using all processors does not always have 4.0 processors active is that portions of each benchmark are not parallel, and therefore use only a single processor at times. This analysis shows that there is a large potential to deactivate processors in many of the benchmarks. For example, ACTOR executes IS with an average of only 1.42 processors, while improving performance by 50.4%. On the other hand, certain other applications do not appear to be an amenable to deactivating entire processors. BT, for example, executes best using all 4 available processors, though an improvement of 14.3% is still observed by occasionally using only one thread per processor.

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

5.6 Summary

The performance and power characteristics of applications on emerging systems demand the consideration of throttling concurrency. In this chapter, we first showed the extremely limited scalability characteristics of a range of parallel scientific applications on a system composed of four Intel Hyperthreaded processors. Additionally, we found highly phase-variant behavior, in that individual phases within applications experience considerably different scalability properties, motivating the consideration of phase-level adaptation. Despite the limited scalability, power consumption did continue to increase, leading to higher power expenditure for reduced performance.

Based on the observed scalability limitations, we have presented a novel approach to dynamic concurrency throttling that uses the model presented in Chapter 4 based on information collected at runtime to predict the performance of an application across various hardware configurations. The library, called ACTOR, is a prediction-based adaptive concurrency throttling system, which we show to outperform adaptation strategies based on empirical searches of the configuration space due to reduced exploration overhead. The observed advantage for prediction-based adaptation over that based on empirical searches motivates the adoption of prediction-based strategies for DCT. The use of ACTOR requires only minor instrumentation to the original source code, which could easily be automated by a preprocessor or OpenMP compiler.

We optimize the adaptation system by introducing cross-phase awareness into the decision making process, thereby allowing it to consider potential cache effects of changing configurations between phases. This optimization proves successful, identifying many cases where phase-local adaptation would suffer from extensive cache-effects, and is able to improve performance relative to the phase-local approach. Adaptive concurrency throttling is shown to be significantly more effective than simply using all available execution contexts for all phases, with improvements of 17.9% in performance, 10.8% in power, and 26.7% in energy consumption. ACTOR yields performance results comparable to offline-derived application/input-size specific decisions, without requiring application/input-size specific offline analysis that makes offline approaches unrealistic.

Chapter 6

Integrating DVFS with DCT

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 [83]. In workload execution 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 [5]. 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 [59]. 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 [74], 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 advanced the understanding of performance and power implications of DVFS [56, 85] and processor/core conservation [16], 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 on multicore processors [83]. 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.

This chapter presents a software framework for multi-dimensional, software-controlled, and HPC-constrained power-performance adaptation on systems based on multi-core processors [26]. 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 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. 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.

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 [87, 118].

Through derivation and evaluation of our prototype, this chapter 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 (17% mean for the NAS benchmarks). In nearly all applications we tested, the energy gains come with simultaneous power savings (6% mean) and performance improvement (12% mean). 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 unified 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.

In the next section, we present background information on applying DCT in conjunction with DVFS. In Section 6.2, we provide a detailed description of our extension to the DPAPP model already described in Chapter 4 with support for simultaneously predicting the effects of applying DCT and DVFS. We describe the implementation of our extension to the ACTOR runtime system for simultaneous adaptation of DCT and DVFS in Section 6.3. In Section 6.4, we present the experimental analysis of the unified adaptation model. Section 6.5 summarizes this chapter.

6.1 Preliminaries

We provide the theoretical foundation for our performance prediction model for systems with multiple multi-core processor dies. 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 that 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.

We assume that we can set each die of the system to execute at an independent voltage/frequency level chosen from a predetermined set by a privileged instruction. 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. Our modeling methodology does not preclude and can be generalized to local (per-core) DVFS schemes.

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 chapter (NAS benchmarks). Ongoing research efforts are addressing DCT in other programming models, such as MPI [52].

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.

6.2 Empirical Model Derivation

We present runtime performance predictors that estimate performance in response to changing the settings of two power-performance knobs, DCT and DVFS. We refer to each combination of frequency and concurrency configuration available on the system as a hardware configuration, or more simply a *configuration*. The predictors use input from execution samples collected at runtime on specific configurations to predict the performance on other, untested configurations. We estimate performance for each phase in terms of useful instructions per second, or $uIPC$, which is the IPC with instructions used for parallelization or synchronization omitted. By using $uIPC$ predictions, we exploit opportunities to save power primarily by scaling 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 $uIPC_t$.

6.2.1 Baseline Prediction Model

Our prediction model uses $uIPC_s$ to estimate the effect of the observed event rates 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. 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 the model represents through positive or negative coefficients. While the relationship between event rates and $uIPC$ may not be strictly linear, a linear model can represent it well [24, 67, 95]. We estimate the specific coefficients through multivariate linear regression as discussed further in Section 6.2.3. We use the following model already presented in Section 4.2 as our starting point:

$$uIPC_t = uIPC_s \cdot \sum_{i=1}^{n}(x_{(i,t)} \cdot e_{(i,s)}) + uIPC_s \cdot \gamma_t + \epsilon_t \tag{6.1}$$

6.2.2 Model Extensions

While the baseline prediction model can be effective for DCT [21, 24], 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, as opposed to a single configuration as described in Chapter 4. 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 6.2 presents the model extended to two samples, with an additional term λ to capture interaction between samples, which we describe next.

$$\begin{aligned} uIPC_t = uIPC_{s1} \cdot \alpha_{(t,s1)}(e_{(1..n,s1)}) + \\ uIPC_{s2} \cdot \alpha_{(t,s2)}(e_{(1..n,s2)}) + \\ \lambda_t(e_{(1..n,S)}) + \epsilon_t \end{aligned} \tag{6.2}$$

Using multiple samples allows us to analyze the relationship between each configuration. We include an interaction term for the product of two events in the linear model to capture the relationship statistically. For simplicity, we only consider possible interactions between the same event across multiple configurations, including the product of $uIPC$ on each sample configuration.Thus, our model considers the interplay between multiple configurations. Specifically, we define the interaction term for a model using two samples as Equation 6.3 shows:

$$\begin{aligned}\lambda_t(1..n, S) = \sum_{i=1}^{n}(\mu_{(t,i)} \cdot e_{(i,s1)} \cdot e_{(i,s2)}) + \\ \mu_{(t,IPC)} \cdot uIPC_{s1} \cdot uIPC_{s2} + \iota_t\end{aligned} \tag{6.3}$$

The interaction term λ_t linearly combines the products of each event across configurations, as well as that of $uIPC$. In Equation 6.3, 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 very large complex configuration spaces, we may need to use even more sample configurations. We can extend our model to an arbitrary collection of samples, S, of size $|S|$, to support such a situation, as follows:

$$uIPC_t = \sum_{i=1}^{|S|}(uIPC_i \cdot \alpha_{(t,i)}(e_{(1..n,i)})) + \lambda_t(e_{(1..n,S)}) + \epsilon_t \tag{6.4}$$

While using more samples generally increases model accuracy, it also increases sampling overhead. We address the selection of S in terms of specific configurations as well as its size in Section 6.2.5.

We generalize the term λ_t further to account for the interaction between events across $|S|$ samples as follows:

$$\lambda_t(e_{(1..n,S)}) = \sum_{i=1}^{n}(\sum_{j=1}^{|S|-1}(\sum_{k=j+1}^{|S|}(\mu_{(t,i,j,k)} \cdot e_{(i,j)} \cdot e_{(i,k)}))) + \sum_{j=1}^{|S|-1}(\sum_{k=j+1}^{|S|}(\mu_{(t,j,k,IPC)} \cdot uIPC_j \cdot uIPC_k)) + \iota_t \quad (6.5)$$

To further improve model accuracy, we apply variance stabilization 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. [77]. 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.

6.2.3 Offline Model Training

We use multivariate linear regression on phases from a set of training benchmarks to approximate the coefficients in our model. We record the $uIPC$ and a predefined collection of event rates while executing each training benchmark's phases on all configurations. We use multiple linear regression on these values to learn the patterns in the effects of sample configuration 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, deriving sets of coefficients independently. We select the set of training benchmarks to include variation in properties such as scalability and memory-boundedness.

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. We also assume that the voltage/frequency of all dies in the system is set simultaneously to the same setting, to support parallel codes better and to avoid load imbalance during parallel execution phases. On emerging architectures that feature hundreds of cores, it may become necessary to reduce the search space further during model training to limit offline overhead, for example by uniform sampling of the system configurations used for training. At current multi-core system scales, the training process using a fully automated system for our approach takes on the order of hours – specifically, approximately 5 hours on our experimental platforms – and scales up linearly with the number of possible configurations.

6.2.4 Predicting Across Multiple Dimensions

We can apply our model to predict the performance effects of DCT and DVFS independently, or across simultaneous changes in the settings of both 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.

6.2.5 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 [21,24]. 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 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). This technique allows us to limit the number of samples while still providing significant input for the predictor along each dimension.

6.3 Implementation

We have implemented our multi-dimensional prediction model within a runtime library to perform online adaptation of DVFS and DCT. We target parallel applications from the HPC domain with iterative structure, such that each program phase is executed many times. We exploit this property to collect hardware event rates during the first few executions of each phase to serve as input for the model. We hardcode the model itself into the runtime system by programming the coefficients derived during the training process for a particular model into the library. The runtime system facilitates online predictions of performance based on the collected hardware event rates.

Our library targets power-performance adaptation of OpenMP applications and is implemented using only portable components. To use our library, applications are trivially instrumented with function calls around each adaptable phase, which are delimited as OpenMP parallel regions. At runtime, the library controls the execution of each phase in terms of the number of threads, their placement on cores, and the DVFS level selected by the predictor for global use. During the sampling phase, configurations are set appropriately and event rates are collected automatically by the library. 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 record hardware event rates with PAPI [9]. We set DFVS levels using the *cpufrequtils* library, specifically using the *sysfs_set_frequency()* call. 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. 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 *unified prediction model*.

When adapting DCT, the library can compare configurations simply using the predicted $uIPC$. However, when considering DVFS, including the hybrid approaches, we must be careful to ensure valid performance comparisons. A problem arises here because at lower frequencies each cycle lasts longer, which causes higher IPCs to occur at lower frequencies while the program actually runs slower. This situation occurs because each memory access requires fewer cycles at lower processor frequencies. For this reason, we calculate instructions per second before making comparisons using the known frequency levels.

A program may have phases that are of too fine granularity to benefit from adaptation using either DVFS or DCT, as the overhead of performing adaptation can exceed its benefits. We have empirically identified a threshold of one million cycles, below which we simply 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.

6.4 Experimental Analysis

In this section, we evaluate our multi-dimensional prediction model. We begin with a brief description of the experimental setup. Next, we analyze the scalability of the benchmarks on our target machine. Then, we evaluate the model of performance prediction used to apply DVFS and DCT. Finally, we compare the benefits of applying DVFS and DCT independently and synergistically, in terms of both performance and energy benefits.

6.4.1 Experimental Setup

Our experimental platform has two Intel Xeon E5320 quad-core processors, for a total of eight cores. 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. This asymmetry is illustrated in Figure 6.1. Each core operates at a maximum frequency of 1.86GHz, with the possibility of reducing to 1.60GHz. The

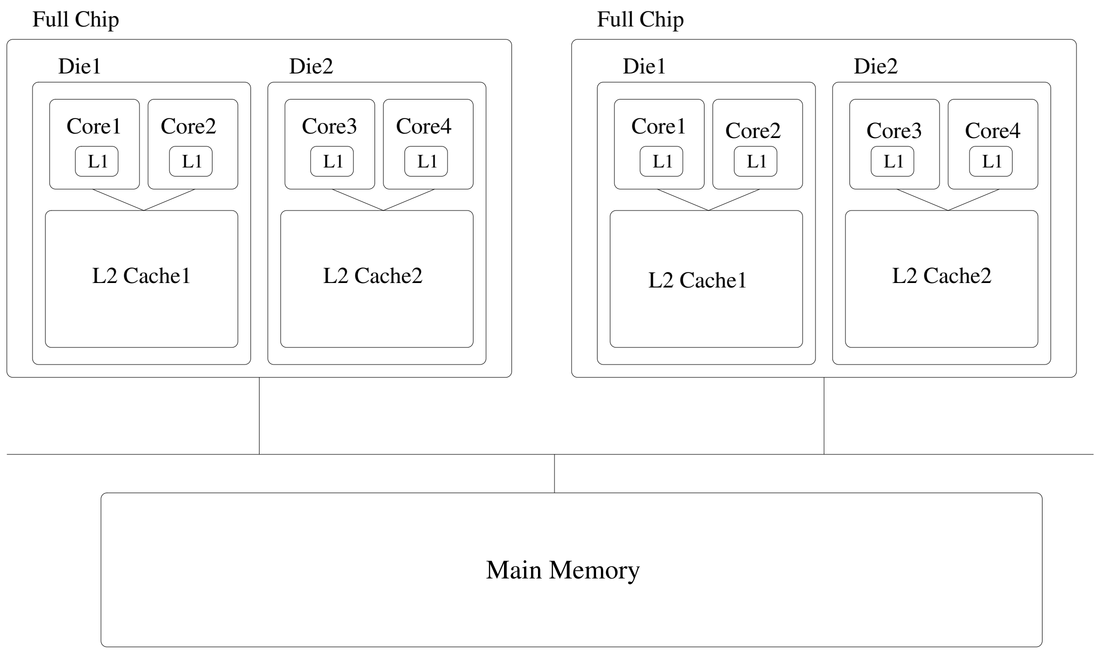


Figure 6.1: Representation of experimental platform containing two Intel E5320 processor dies. Note the sharing of each L2 cache between pairs of cores.

system contains 4GB of memory and runs Linux kernel version 2.6.22. In all experiments, full system energy is collected per run using a Watts Up Pro power meter, and the average power consumption is computed based on the execution time and total energy consumption.

We experimented with benchmarks representative of parallel applications from the HPC domain. Specifically, we use seven benchmarks from the OpenMP version of the NAS Parallel Benchmarks suite (3.1). We use the class B input size, which consumes between 5% and 30% of system memory. The codes, implemented in C or Fortran and parallelized using OpenMP, have been extensively optimized for parallelism and locality [58]. The benchmarks have large variation in several interesting execution properties, including number of phases, scalability (global and per phase), compute- and memory-boundedness of phases, number of loop iterations, and computational intensity, thus making prediction challenging.

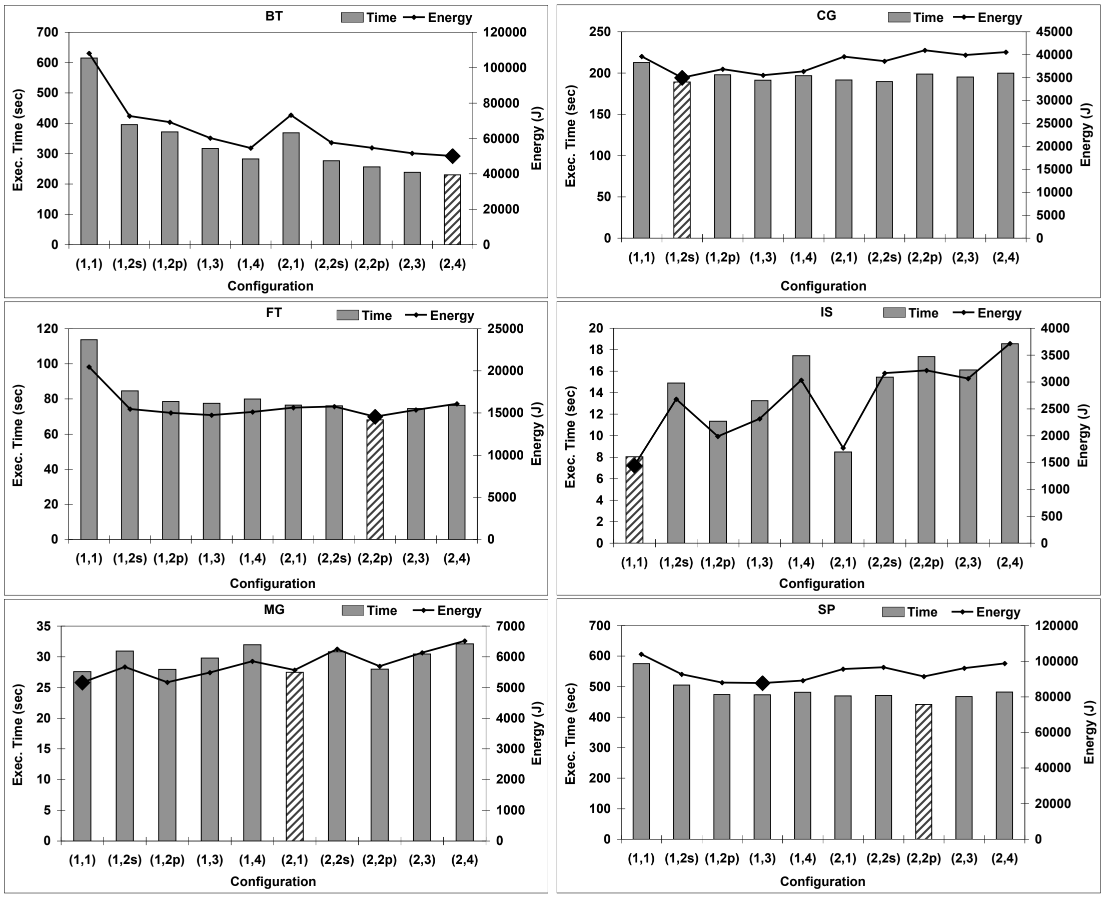
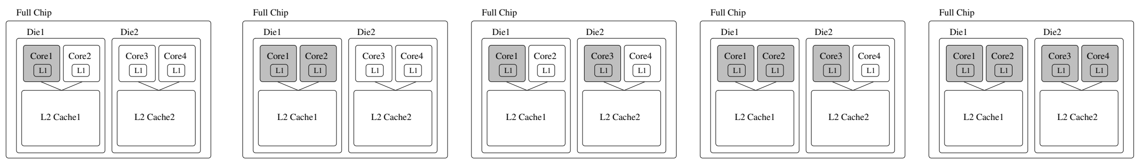


Figure 6.2: Execution time (bars) and energy consumption (lines) of the benchmarks across all configurations. The notation $(X,\ Y)$ denotes non-adaptive execution with X processors and Y cores per processor. The configurations with the best performance and energy for each benchmark are marked with a stripe and a large diamond respectively.



(a) 1 (b) 2s (c) 2p (d) 3 (e) 4

Figure 6.3: The five possible configurations on each processor.

6.4.2 Application Scalability Analysis

Before evaluating DVFS- and DCT-based adaptation using performance prediction, we briefly analyze the scalability of the applications on our platform. To do so, we execute each application under all symmetric configurations on the experimental platform and record the execution time. Figure 6.2 presents the scalability results on our dual-processor, quad-core Intel Xeon system. As stated earlier, two threads on a single chip can execute with shared or private caches on this architecture. We use the notation $2s$ to indicate a shared cache and $2p$ to indicate private caches on our graphs. The notation (X, Y) denotes non-adaptive execution with X processors and Y cores per processor, and later an additional term Z is included to indicate the DVFS level used. Figure 6.3 displays the five possible configurations using a single processor. Symmetric configurations exist using two processor yielding a total of ten configurations on our experimental platform.

The figure shows that, in general, applications are far from scaling perfectly on the target platform. In particular, only one application 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). Second are applications that incur a non-negligible performance *loss* when using *more* cores (IS and MG). Third are 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.

BT illustrates that scalability on the quad-core processors is not inherently limited, if the application exhibits a high ratio of computation to memory accesses. In BT, the lowest energy consumption also occurs at full concurrency, because scaling achieves significantly higher performance with only incremental additional power consumption. The limited scalability of other applications occurs due to particular execution properties, notably high memory access rates. In particular, BT sees a speedup of 2.68 through maximum concurrency compared to sequential execution.

IS and MG exhibit 2.31x and 1.17x slowdowns respectively from using all cores due to memory access intensity and the limited memory bandwidth available on our platform. Furthermore, IS observes a 31.5% performance improvement through the allocation of the entire processor cache space to a single core, compared to sharing the caches between two or more cores per processor, suggesting that destructive interference in the shared L2 results in memory bandwidth saturation and limited scalability. Interestingly, IS and MG both achieve minimal energy consumption using only a single thread, with additional concurrency increasing energy consumption by 157.1% and 26.3%, respectively.

CG, FT, and SP exhibit limited scaling from additional concurrency and their speedups have a plateau when concurrency crosses the boundary of a single quad-core processor. These benchmarks have a mean speedup of 24.9% through the use of all available cores compared to single-threaded execution. FT and SP obtain speedups of 42.2% and 19.5% respectively, from the four cores of a single processor, however minimal benefit is seen through the use of the second processor. In these three benchmarks, using fewer cores reduces energy consumption, as comparable performance is usually achieved at lower power consumption levels.

Collectively, the benchmarks see a geometric mean slowdown of 4.7% scaling to maximum concurrency. At the same time, total system power consumption increases by 13.9% due to increased resource utilization. These effects combine to yield an average increase in energy consumption of 17.6%. Though multicore architectures are being marketed as an energy-efficient solution, clearly the efficiency in practice depends heavily on the scalability provided by the architecture for a given application. If multicore processors are to be adopted for use in the HPC arena, either the system will need to be improved for the known properties of HPC applications, or the applications themselves will need to be re-engineered for better performance on multicore architectures.

The most energy-efficient configuration coincides with the most performance-efficient configuration for 4 out of the 6 benchmarks (BT, CG, FT, and IS). For 2 benchmarks (MG, SP), the user can use fewer than the performance-optimal number of cores, to achieve substantial energy savings, at a marginal performance loss. We also observe that for a given number of threads, performance can be very sensitive to the mapping of threads to cores (e.g. BT, FT, and SP when executed with 2 or 4 threads). Even if performance is insensitive to the mapping of threads to cores, power can be sensitive to the mapping of threads to cores. In MG, for example, distributing two threads between two sockets on the same die is imperceptibly less performance-efficient, but significantly more energy-efficient than distributing two threads between two dies.

Execution properties are not static within a given application [102]. Rather, applications experience phased behavior where program characteristics vary at repeating intervals. In our test cases, program phases have widely varying scalability and energy-efficiency characteristics, even within a single application. For example, the SP benchmark alone contains phases that experience optimality at six distinct configurations, with full concurrency yielding speedups ranging from $0.68\times$ to $3.24\times$. Thus, the optimal configuration for any given program phase may differ from surrounding phases. Runtime identification of poorly scalable phases could support fine-grain dynamic concurrency throttling that executes them with more efficient threading configurations.

We attribute the observed limited scalability of several benchmarks to memory contention, at all levels of the memory hierarchy. Specifically, two cores sharing a cache rarely benefit from data sharing, but rather suffer from destructive interference in the form of increased conflict misses between threads. Further, additional threads produce a higher demand on main memory, producing contention at the shared front-side bus. These issues combine to limit scalability most in applications that are memory intensive and have primary or secondary working sets too large to fit into on chip cache space.

6.4.3 Performance Prediction Evaluation

We can train our prediction model with various training sets of one or more benchmarks. We selected a single benchmark to train the model, trading potentially higher prediction accuracy for less training time. Specifically, we used NAS-UA to perform training. UA has a large number of phases and widely varying execution characteristics on a per phase basis, including IPC, scalability, locality, and granularity. We have also used extended training sets with more benchmarks, but we do not present those results, since they did not notably improve model accuracy compared to our reduced training set. We selected sample configurations for each model to maximize the amount of information available to the model. For the DVFS model, we selected a single sample at maximum frequency within each given threading configuration. We selected two samples for DCT: (1,3) and (2,4). Finally, for the unified DVFS-DCT model, we selected three sample configurations: (1,2p,2), (2,2s,1), and (2,4,2), where the third term indicates the frequency level. These sample configurations were selected as outlined in Section 6.2.5 to provide data along each dimension of adaptation. In all cases, we made predictions for all configurations not sampled.

Using many events can benefit the model, however current architectures severely limit the maximum number of events that we simultaneously record, while multiplexing many events on the available event registers has a significant overhead and limited accuracy. Thus, we set the number of events used in our model to the number supported in the hardware. On our experimental platform, only two event registers are available, and one must always be used to collect $uIPC$, which is mandatory with our model. For all three models, the auxiliary event with the highest correlation with target IPC in the training data was L1 data cache accesses.

We derived the model coefficients offline using linear regression on samples of event rates and $uIPC$ on each configuration from the training benchmark. Figure 6.4 shows the percent of predicted samples for each model with error less 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). We note that prediction is performed with input from 1, 2, or 3 sample configurations for the remaining of 20 possible configurations on our platform. The higher accuracy of predicting DVFS than DCT results from a simpler set of effects in changing DVFS level that our model captures easily, while DCT has complicated performance effects, due to the irregular, non-monotonic scalability patterns of many phases. Of the 20 possible configurations, the unified model correctly identifies the single best configuration in 35% of cases 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 model is 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 [24, 77].

6.4.4 Evaluation of DCT and DVFS Adaptation

In this section, we evaluate the use of our prediction models in conjunction with runtime adaptation of multithreaded scientific codes. We begin by comparing the use of only DVFS or DCT. We then analyze two schemes for adapting both DVFS and DCT, specifically applying them sequentially or in a unified manner. Finally, we compare prediction-based adaptation against the use of empirical search in identifying optimal configurations. Figure 6.5 presents the results of adaptation through the various mechanisms for each benchmark and Figure 6.6 shows the geometric mean of the normalized energy and execution time results.

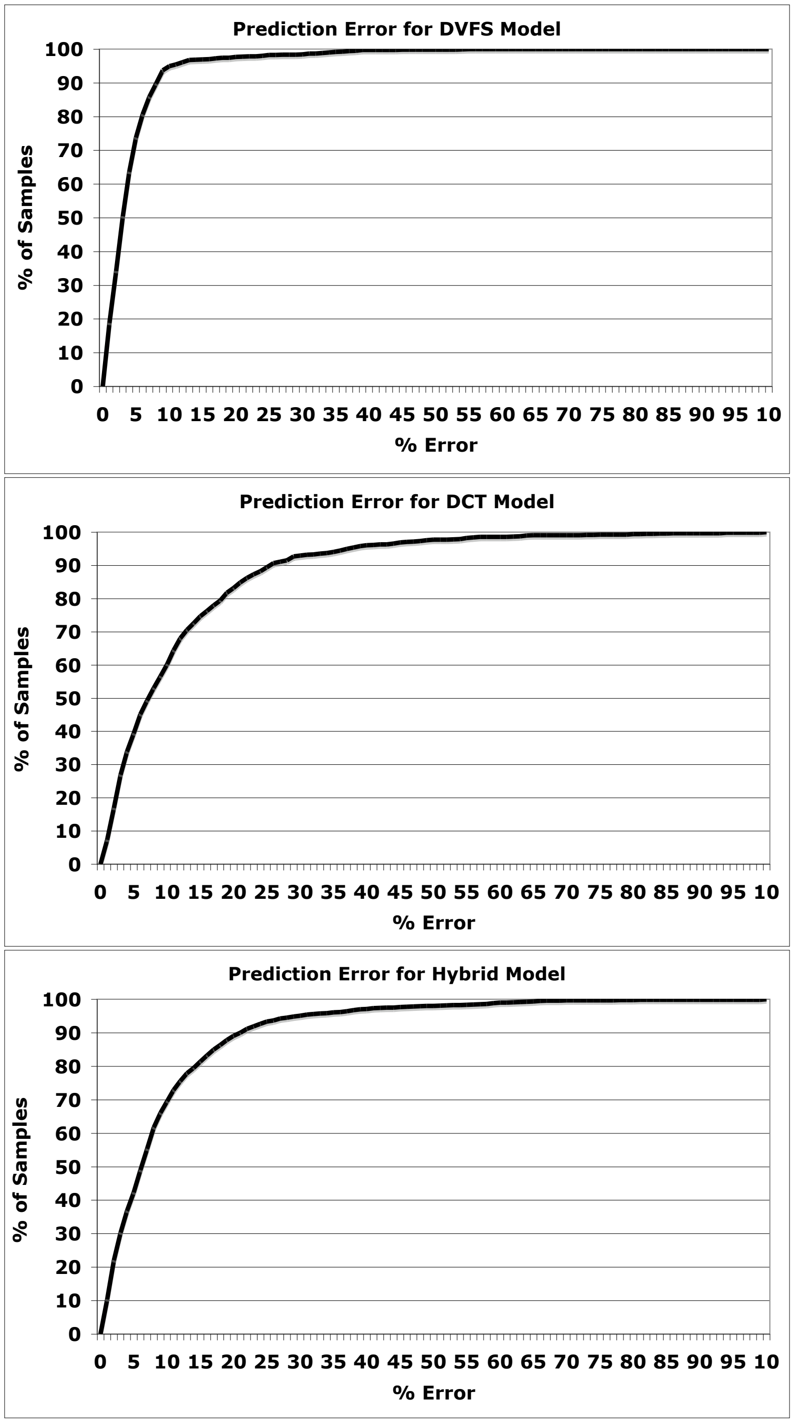


Figure 6.4: Cumulative distribution functions of prediction accuracy of the three prediction models.

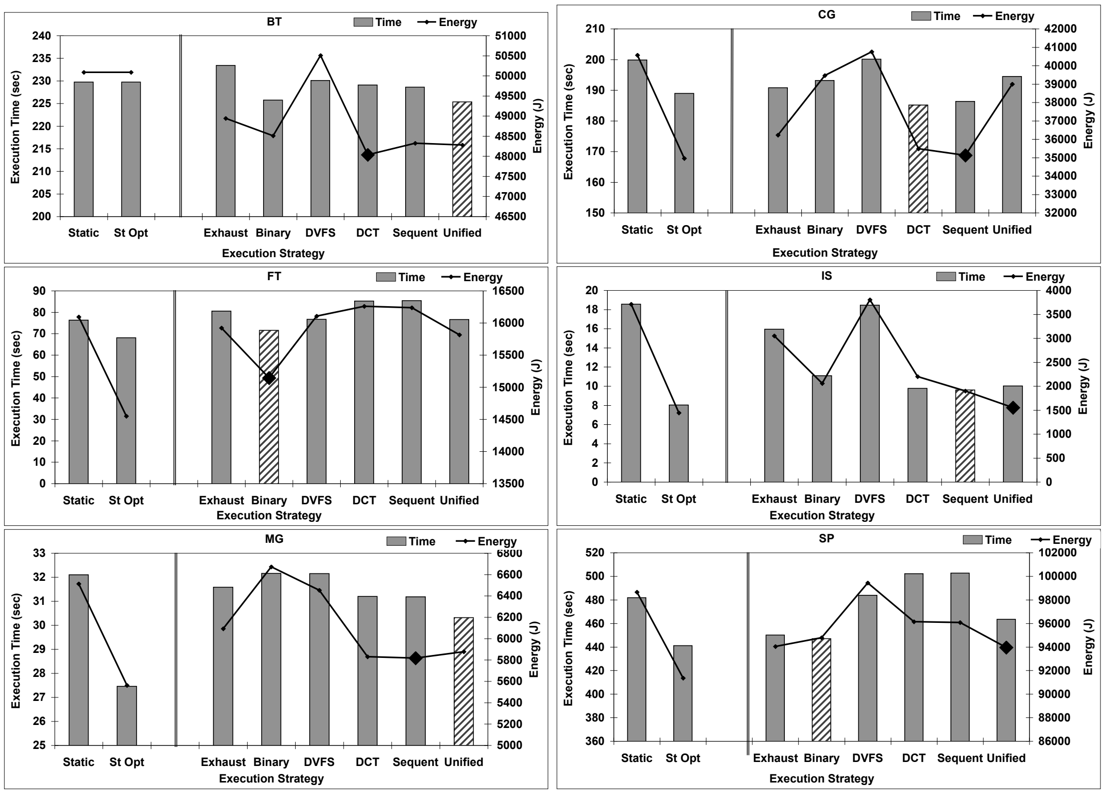


Figure 6.5: Results of adaptation through various techniques. The group of bars left of the divider represent static configurations and those right of the divider are the adaptive strategies. The adaptive configurations with the best performance and energy for each benchmark are marked with a stripe and a large diamond respectively.

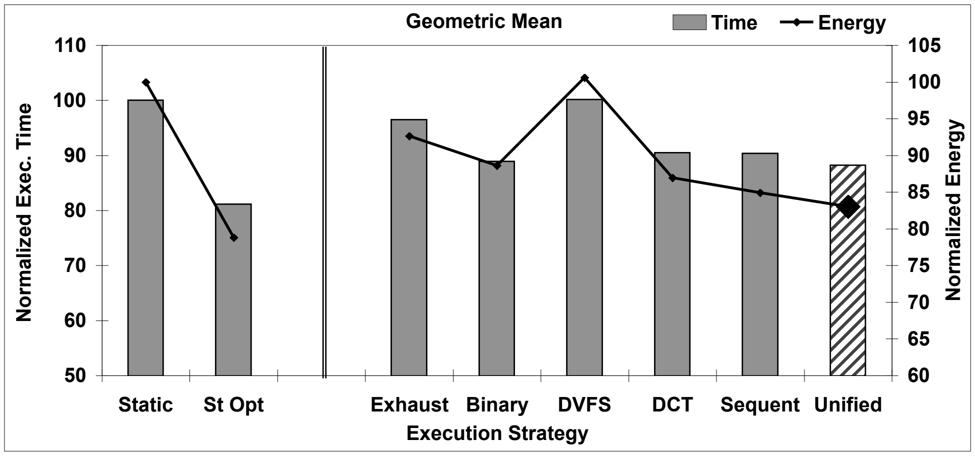


Figure 6.6: Geometric means of the benefits of adaptation through various strategies. The adaptive configurations with the best performance and energy for each benchmark are marked with a stripe and a large diamond respectively.

We compare the various strategies against the results of "static executions" that use a single configuration for the entire execution. We specifically compare to using full concurrency and frequency (*static*) and to the best performing of all static executions (*static optimal*). Clearly, we derive the static optimal post-facto: we could not know it online in practice without exhaustive offline execution of each application on all configurations, for each specific input. 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, as each phase may have its own optimal configuration.

Adaptation using one knob

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. The results of applying DVFS using the model support the intuition that DVFS is generally unable to improve performance compared to simply using all available cores at maximum frequency. There are cases in memory-bound phases where this assumption is violated and scaling down frequency can actually improve execution time [42], but these phases are relatively rare. Specifically, our experiments reveal no benefit in terms of performance or energy from adapting to the DVFS level with the highest predicted performance. Without allowing for some loss in performance, DVFS alone is not generally able to significantly benefit energy consumption. We can attribute this result to some extent to our system having only two voltage/frequency levels available to software, where the lower level is not substantially more power-efficient than the higher level (1.6 vs. 1.86 GHz).

On the other hand, using DCT with the prediction model provides substantial benefits in execution time (9.5% mean savings), power consumption (3.7% mean savings), and energy consumption (13.1% mean savings) compared to the static execution with all cores active. Despite the overall success of DCT, mispredictions for two benchmarks result in an observed increase in execution time – FT by 11.6% and SP by 4.2%. However, SP still manages an energy savings of 2.5% because of the reduced power consumed by the fewer active cores, and FT increases energy by only 1.0% and is the only benchmark not to have energy consumption reduced through DCT. In contrast, the largest benefit occurs with IS which sees a 40.7% reduction in energy consumption. When compared to the static optimal execution, DCT is within 8.1% mean performance and even surpasses that performance with CG by 2.0% due to phase-awareness. These results indicate that at least at the scale of a few multi-core processors integrated on a single node, DCT is the more suitable of the two power-performance knobs for use in the HPC domain, since it can improve or sustain performance, and improve energy-efficiency by substantial margins simultaneously.

Adaptation using two knobs

We can combine the two power-performance knobs in multiple ways. First, we consider applying them sequentially. We 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. Since we make decisions to maximize predicted IPC and our platform has only two DVFS levels with a small frequency difference, DVFS adds very little benefit to DCT alone, and no reduction in execution time compared to DCT alone occurs. However, DVFS reduces power and energy consumption by 2.3% and 2.0% respectively beyond DCT alone on average, by identifying several isolated phases to reduce frequency without negatively impacting performance.

The major advantage of the 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.

Because of its advantages over the sequential approach, the unified scheme improves performance by 2.2% and reduces energy by 1.9% (geometric mean improvements over the sequential approach). When compared to the default execution using maximum concurrency and frequency, the advantages of unified adaptation become even clearer. Specifically, we see an 11.8% speedup simultaneous with a 5.9% reduction in power consumption, resulting in an overall reduction in energy consumption of 17.0% (geometric mean improvements over static execution). In fact, all benchmarks experience improved performance with the exception of FT, which slows down by a mere 0.3%. Similarly, all benchmarks benefit from reduced energy consumption. Even when compared to the oracle-derived executions on the static optimal configuration for each benchmark, unified adaptation achieves energy consumption within 4.2% and performance within 7.0% (geometric means), and better performance by 2.0% 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. search approaches

We have also implemented two empirical search approaches to identify optimal DVFS and DCT 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. This approach does not require any offline training, so the programmer can use it with minimal effort. However, the online overhead of testing many possible configurations stands to reduce any potential benefit of adaptation considerably, which is what occurs in practice. The exhaustive search method reduces execution time by 3.5%, power by 4.0%, and energy by 7.3%, well below the savings of our prediction-based techniques. Exhaustive search proves superior to prediction schemes in only SP, which executes 400 workload-invariant iterations and where prediction sees higher than usual configuration prediction error.

For comparison purposes we also consider a heuristic search approach, based on a binary search of the configuration space, similar to the approach evaluated by Li and Martinez [83]. A fair direct comparison between our prediction models and the approach discussed previously [83] is not possible, since contrary to the simulation-based study in previous work, our evaluation was conducted on a real system with a different workload (NAS benchmarks), and 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 useful insight on the appropriateness of heuristic search and prediction-based approaches to dynamic program adaptation.

Our implementation of binary search begins by executing at full concurrency and frequency, then sequentially performs binary searches of the concurrency and DVFS dimensions. During the searches, if a sample is tested with worse performance that the first sample, concurrency or DVFS is increased in the next tested sample. This approach has considerably reduced overhead compared to the exhaustive search, because many configurations need not be tested, resulting in 7.6% better performance and 4.1% lower energy consumption (geometric mean improvements over exhaustive search). Compared to the static execution, performance is improved by 11.1% and energy by 11.4%, however power consumption is only reduced by 0.4%. 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 7.7% for performance and 9.8% for energy.

The most interesting comparison is between the unified prediction model and binary search. Binary search achieves performance 0.7% worse than the unified prediction approach while consuming 5.6% more energy (geometric mean differences). Binary search suffers from blind-spots that prevent identification of effective configurations at low concurrency or DVFS levels, which tend to consume less power, so the unified prediction model can reduce power further, on average by 5.5%. Binary search does have better performance than the unified model in three of six cases (CG, FT, and SP), however energy consumption is higher in all but one case (SP). In particular, the results of binary search suffer 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. 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.

6.5 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 chapter, 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 synergistic fashion, can simultaneously improve performance and energy-efficiency relative to either approach in isolation. Specifically, a unified adaptation model achieves performance improvements of 12%, power savings of 6%, and energy savings of 17% compared to using all cores at full frequency, outperforming an approach that 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. 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.

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, and we explore this in the next chapter. 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 chapter, it is far more noticeable on large-scale multiprocessors, such as NUMA systems. 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 runtime environments for parallel programming 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.

Chapter 7

Evaluating ANNs for Performance Modeling

In Chapter 5, we evaluated phase-sensitive concurrency throttling on a system of multiple simultaneous multithreaded processors. In prior work, we rely on regression techniques for predicting optimal levels of concurrency and thread placement. Here we leverage machine learning, specifically artificial neural networks (ANNs). We use ANN-based performance prediction to identify the desired level of concurrency and the optimal thread placement [27]. The ANNs are trained offline to model the relationship between performance counter event rates observed while sampling short periods of program execution and the resulting performance with various levels of concurrency. The derived ANN models allow us to perform online performance prediction for phases of parallel code with low overhead by sampling performance counters. The use of our ANN approach removes the burden of managing the training phase and providing domain-specific knowledge, two steps that are crucial to regression-based prediction strategies [77].

This work is the first to compare different prediction techniques directly in the context of scalability prediction [28]. We explore two primary techniques: multiple linear regression and artificial neural networks (ANNs). We compare ANNs to linear regression within ACTOR, ensuring an accurate reflection of the trade-off between prediction speed and accuracy. We conduct this study on a medium-scale SMP, with two quad-core processors, for a total of eight cores, which provides increased phase-sensitivity of the optimal concurrency levels.

In this chapter, we make five primary contributions. First, we analyze the effectiveness of ANN-based concurrency throttling on a quad-core Intel Xeon. Second, we rigorously compare the efficacy of different prediction techniques for online concurrency throttling. Third, we provide an improved understanding of the phase-sensitivity of the scalability and energy-efficiency of multithreaded scientific applications on multicore SMPs. Fourth, we evaluate the effectiveness of concurrency throttling in such an environment through direct measurement of power consumption. Fifth, we conclude that a linear model is sufficiently accurate to achieve significant performance gains, and outperforms, on average, a model derived using ANNs in terms of overall performance and power consumption in an online DCT system, although ANN-based models reach a slightly larger number of optimal DCT decisions.

This chapter has two primary sections. In Section 7.1, we describe the use of ANNs for scalability prediction within the DPAPP model. Section 7.1.1 discuss what ANNs are and how they may be used with DPAPP. We then present the results of DCT using ANNs in Section 7.1.2. We also perform a comparison of ANNs to multiple linear regression in Section 7.2. We discuss the way each modeling technique is used in Section 7.2.1 and present the prediction accuracy and results of DCT using each technique in Section 7.2.2. In Section 7.3, we provide a summary of the entire chapter.

7.1 Evaluation of ANNs for Performance Modeling

7.1.1 Concurrency Throttling Using Neural Networks

We now describe the performance prediction component of ACTOR, our runtime system that dynamically throttles concurrency to improve performance and energy efficiency. ACTOR adapts applications by identifying better-performing numbers of threads and thread placements for each phase. Phases are collections of parallel loops or basic blocks assigned for execution to different threads. We focus on a novel approach to concurrency throttling based on runtime performance prediction using ANNs on observed performance counter event rates.

Overview of Artificial Neural Networks

Machine learning studies algorithms that *learn* automatically through experience. For our problem, we focus on a particular class of machine learning algorithms called *artificial neural networks* (ANNs). Their many previous uses include microarchitectural design space exploration [54], workload characterization [121], and compiler optimization [32]. ANNs automatically learn to predict one or more targets (here, IPC) for a given set of inputs. We choose ANNs because they are flexible and well suited for generalized nonlinear regression, and their representational power is rich enough to express complex interactions between variables: any function can be approximated to arbitrary precision by a three-layer ANN [94]. They require no knowledge of the target function, take real or discrete inputs and outputs, and deal well with noisy data.

An ANN consists of layers of *neurons*, or switching units: typically, an input layer, one or more hidden layers, and an output layer. Input values are presented at the input layer and predictions are obtained from the output layer. Figure 7.1 shows an example of a fully connected feed-forward ANN. Every unit in each layer is connected to all units in the next layer by weighted edges. Each unit applies an *activation function* to the weighted sum of its inputs and passes the result to the next layer. Figure 7.2 [94] shows a unit with a sigmoid activation function. One can use any nonlinear, monotonic, and differentiable activation function. We use the sigmoid activation function for our models.

Training the network involves tuning edge weights via backpropagation, using gradient descent to minimize error between predicted and actual results. In this iterative process, the train-

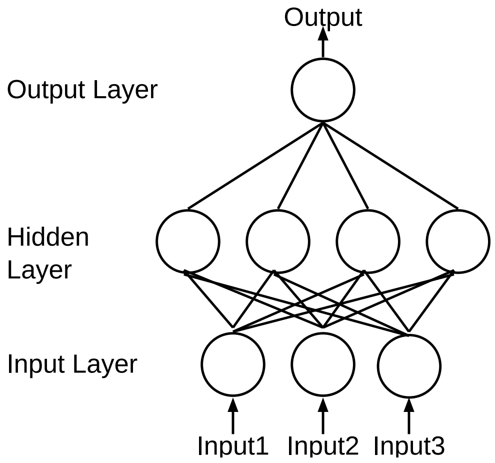


Figure 7.1: Simplified diagram of fully connected, feed-forward ANN.

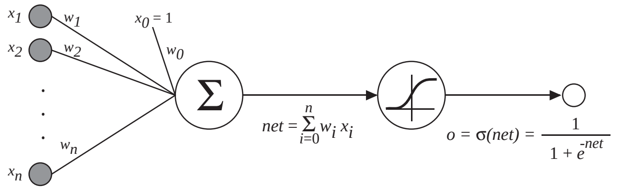

x_1
w_1
$x_0 = 1$
x_2
w_2
w_0
w_n
x_n
Σ
$net = \sum_{i=0}^{n} w_i x_i$
$o = \sigma(net) = \frac{1}{1 + e^{-net}}$

Figure 7.2: Example of a hidden unit with a sigmoid activation function.

ing samples are repeatedly presented at the input layer, and the error is calculated between the prediction and the actual target. The weights are initialized near zero and are updated using an update rule (similar to the one shown in Equation 7.1) in the direction of steepest decrease in error. As weights grow, the network becomes increasingly nonlinear.

$$w_{i,j} \leftarrow w_{i,j} - \eta \frac{\partial E}{\partial w_{i,j}} \tag{7.1}$$

ANNs have a tendency to *overfit* on training data, leading to models that generalize poorly to new data despite their high accuracy on the training data. This is countered by using *early stopping* [13], where we keep aside a validation set from the training data and halt training as accuracy begins to decrease on this set. However, this means we lose some of our training data to the validation set. To address this, we use an ensemble method called *cross validation* to help improve accuracy and mitigate the risk of overfitting the ANN. This technique consists of splitting the training set into n equal-sized *folds*. Taking n=10, for example, we use folds 1-8 for training, fold 9 for early stopping to avoid overfitting, and fold 10 to estimate performance of the trained model. We train a second model on folds 2-9, use fold 10 for early stopping, and estimate performance on fold 1, and so on. This generates 10 ANNs, and we average their outputs for the final prediction. Each ANN in the ensemble sees a subset of training data, but the group as a whole tends to perform better than a single network because all data has been used to train portions of it. Cross validation reduces error variance and improves accuracy at the expense of training multiple models.

We model the effects of changing concurrency and thread placement. Hardware performance counter values collected during a brief sampling period at maximal concurrency become input to our ANN ensemble that predicts IPC for each phase on alternative configurations. The online sample period runs on as many cores as available to represent the greatest possible interference among threads, and resulting predictions estimate the degree to which contention will be reduced by throttling concurrency. Our modeling approach produces the following function for each desired target configuration, T, mapping observed event rates (e_i) on the sample configuration, S, to IPC on the target configuration.

7.1.2 Results of ANN-based Adaptation

Evaluation of ANN-based Performance Prediction

Our experimental platform is a Dell Precision Workstation 390n running Linux kernel version 2.6.18. The particular machine has a single Intel Xeon quad-core processor (QX 6600). The processor is designed as two dual-core processors placed on a single chip. As such, there are two 4MB L2 caches, each shared between two of the cores. Hereafter, we refer to the two cores sharing a single cache as tightly coupled, and as loosely coupled otherwise. Additionally, the system has 2GB of main memory and a 1066MHz frontside bus.

In our evaluations, we use benchmarks from the NAS Parallel Benchmark suite version 3.2 [58] to represent modern scientific applications. The codes are implemented in either C or Fortran, have been parallelized using OpenMP, and have been extensively optimized for parallelism and locality [58]. We execute them under various levels of concurrency and under specific bindings of the threads to cores, performing experiments with five different threading configurations: first, a single thread bound to a single core (configuration 1), two threads bound to two tightly coupled cores (configuration 2a), two threads running on two loosely coupled cores (configuration 2b), three threads (configuration 3), and four threads running on all four cores (configuration 4). Figure 3.6 shows the limited scalability of parallel applications on our testbed.

Performance counters were collected using PAPI version 3.5. We use each benchmark for evaluation by training as many models as there are applications, each time leaving one particular application out of the training process. In this way, we perform prediction for each application with a model that has never seen data from the target application. In practice, the model would generally be trained a single time with a given set of training applications, and would subsequently be used for any desired application, with possible refinements to reflect data from the current workload.

In our evaluation of the ANN-based predictor, we have selected a set of twelve hardware events representing the cache and bus behavior of the application. Our experimental platform only allows the simultaneous recording of two events. As a result, we employ collection across multiple timesteps to record all necessary events. However, several of our benchmarks contain very few iterations, in which case the sample execution period can consume a significant fraction of the overall execution time, thereby limiting the potential benefits of adaptation. In response to this situation, we limit the number of monitored timesteps to at most 20% of the total execution. While reducing the number of counters used in prediction will likely have some minimal effect on the prediction accuracy, the benefits of using the improved concurrency level for a larger percentage of execution time is likely to outweigh the negative effects on accuracy.

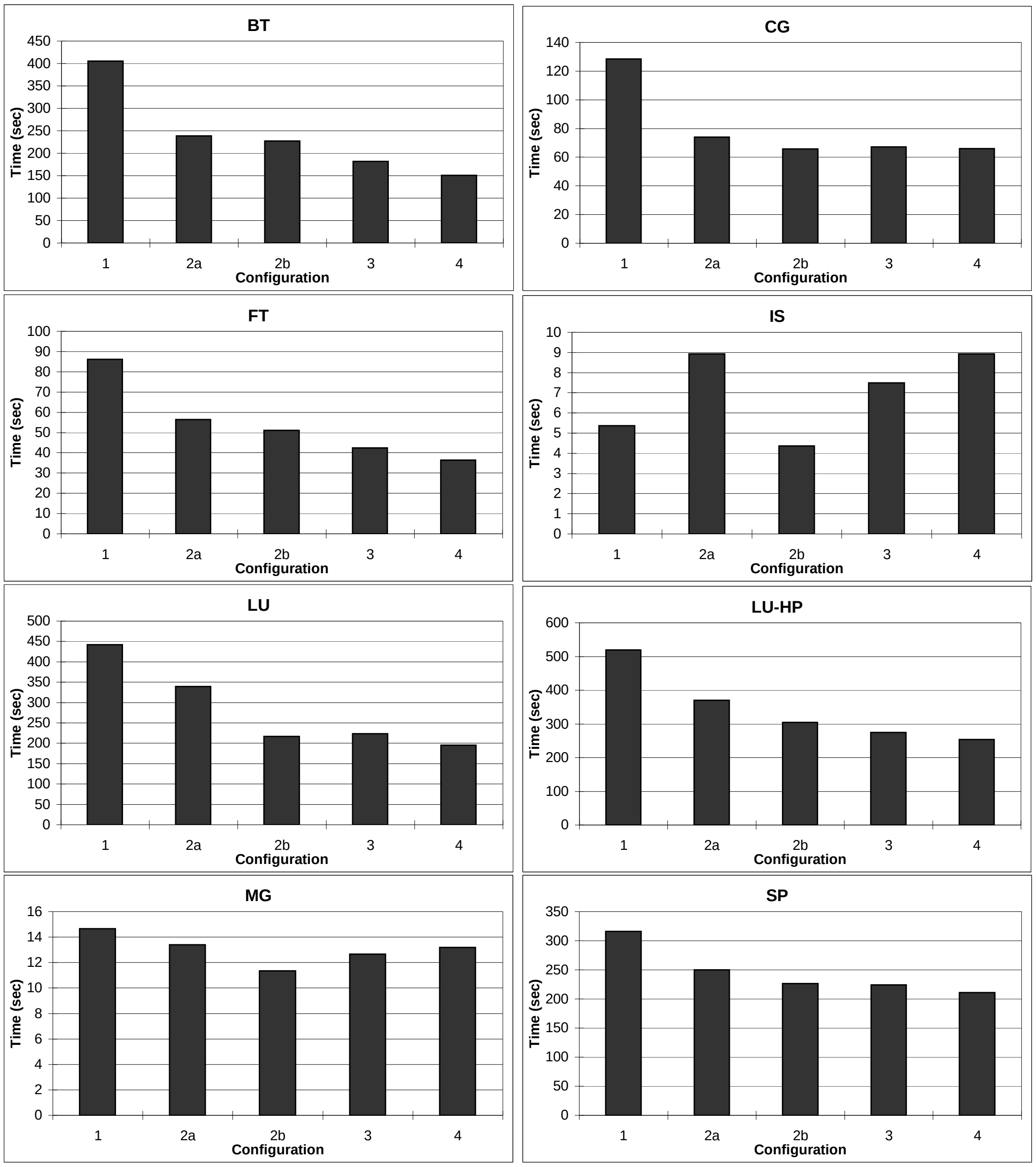


Figure 7.3: Execution times by hardware configuration.

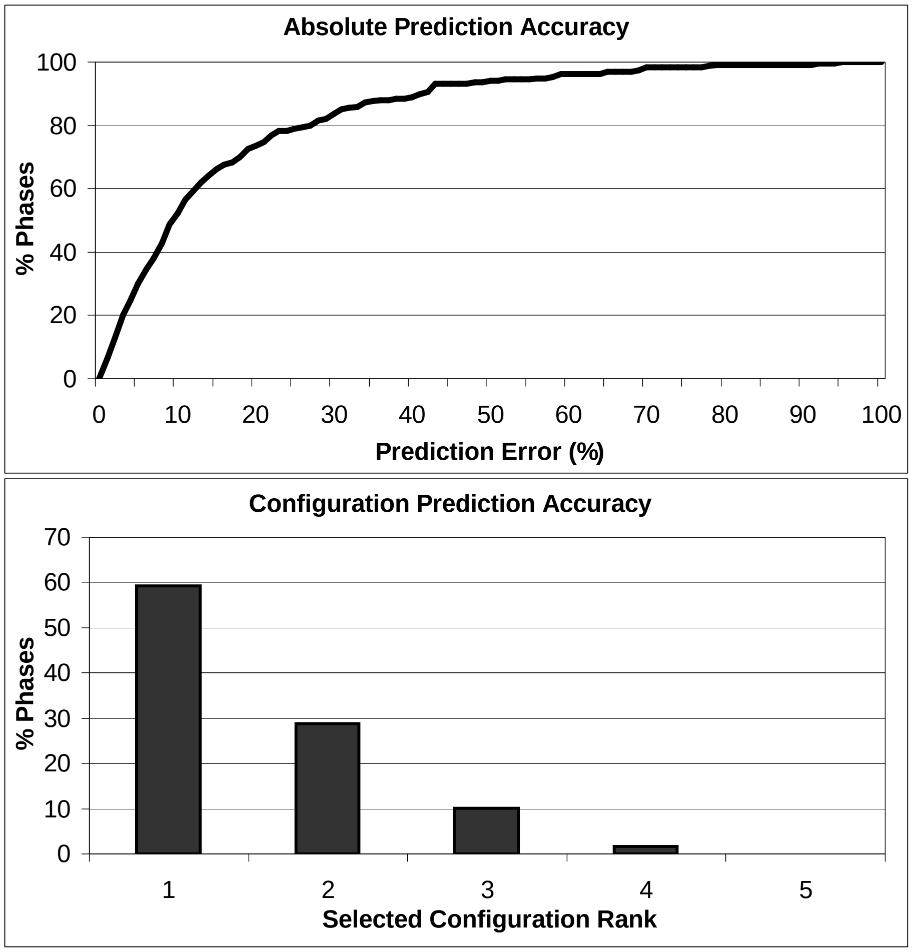


Figure 7.4: Cumulative distribution function of prediction error (top) and percent of phases for which each ranking configuration is selected (bottom).

In the following evaluation we use a reduced number of events for the applications with fewer iterations (FT, IS, and MG).

Figure 7.4 gives a cumulative distribution function of the error of our ANN-based predictor, showing the percentage of samples that fall within increasingly higher levels of observed error. Specifically, we make predictions for four target configurations (1, 2a, 2b, and 3) and these results are accumulated over all predictions made. For each sample, error is calculated as $|(IPC_{obs} - IPC_{pred})/IPC_{obs}|$. Overall, the median error is only 9.1%. Further, 29.2% of the predictions exhibit errors of less than 5%.

An alternative metric for evaluating the accuracy of the predictor in the context of concurrency throttling is the rate at which the optimal configuration is selected. Figure 7.6 shows the percentage of phases where each ranking configuration is selected. In 59.3% of phases, the single best configuration is correctly identified, and the second best configuration is selected in an additional 28.8%. In only one case out of 59 is the second worst configuration selected, and the worst is, in fact, never identified as optimal. These results show that ANN-based performance prediction is effective at identifying optimal or near-optimal concurrency levels.

Concurrency Throttling Evaluation

Figure 7.5 displays the results of our prediction-based concurrency throttling approach normalized by the four core execution, as well as those of the alternative execution strategies. We compare against using all available cores for multithreaded execution, which would normally be the default for a performance-oriented developer. We present results for two approaches based on the use of oracle-derived configurations. The one that we call the global optimal uses the best static configuration for an entire application. The second, the phase optimal, uses the best configuration for each phase. In each of these cases, this information would not normally be available, however they serve as points of comparison to evaluate the effectiveness of the library.

By using our approach for low overhead identification of improved concurrency levels, we see an average performance gain of 6.5% compared to the default strategy of simply using all available cores. Even BT, which scaled well on the four core machine, sees a substantial gain of 4.7% through our phase-aware adaptation strategy, which successfully identifies phases in BT that can be improved by concurrency throttling. Additionally, SP sees minor gains from more cores, however ACTOR is able to improve its performance by 5.2%.

When compared to the two oracle-derived strategies, we can see that ACTOR falls short of these oracular approaches, coming in 2.5% and 4.9% slower on average than the global and

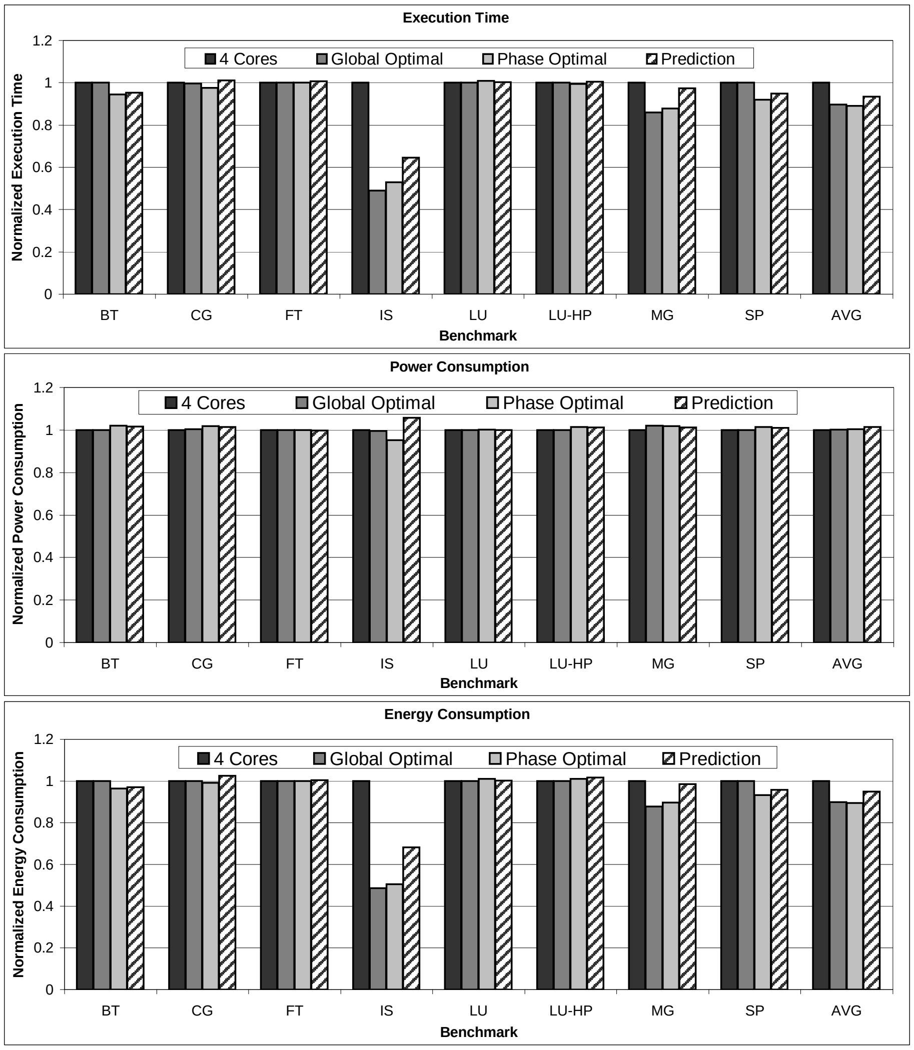


Figure 7.5: Execution time, power consumption, and energy consumption prediction-based adaptation compared to alternative execution strategies.

phase optimals, respectively. This shows potential benefits of improving prediction accuracy. Further, reduced online overhead of sampling is possible on architectures with more counter registers to reduce the number of rotations necessary for event collection.

One surprising result is that no power is saved through concurrency throttling, on average. We successfully leave cores idle, but it is likely that by changing the binding of threads, we are interfering with cache warmth. This, in turn, causes increases in bus and memory accesses, thereby increasing off-chip power consumption. So, while on-chip power consumption is reduced by small amounts, this is overcome by the off-chip increase. There are also cases, as pointed out in Section 6.4.2, where power is increased through selecting reduced threading configurations with better performance. Together, these situations result in an average increase in power consumption of 1.5%. However, given the considerable improvement in execution time, the total energy consumption goes down by an average of 5.2%.

A popular metric in power-aware HPC is energy-delay-squared (ED^2), which considers power consumption but is more influenced by performance, commensurate with the heavy emphasis on performance in HPC. Given the large improvements in execution time, with very minor increases in power consumption, we experience significant reductions in ED^2, saving an overall 17.2%. The most significant result occurs with IS, which shows that for applications that scale poorly, concurrency throttling is imperative to achieve energy-efficiency with a 71.6% improvement in ED^2. However, it is clear that further gains are possible through this approach as the phase optimal execution sees a 29.0% improvement compared to using four cores.

7.2 Comparing ANNs to Regression

To compare artificial neural networks with multiple linear regression, we utilize a different experiment platform than for the evaluation of ANNs alone in the previous section. Instead, we perform physical experimentation on a state-of-the-art dual-processor quad-core Intel Xeon platform as described in Section 6.4.1. Though not a ”many-core” system, this larger scale platform further tests the scalability of parallel applications and better represents future systems with a larger number of cores.

7.2.1 Strategies for Scalability Prediction

Training with Linear Regression

To derive a linear regression-based model, we use the training process outlined in Section 6.2. Linear regression has some limitations. First, it only considers terms that are explicitly included in the input and model accuracy requires that we encode detailed architectural knowledge in the model. Second, we must explicitly model interactions between multiple events or between a particular event across sample configurations. The required specification of architectural knowledge and event interactions increases the burden on the modeler. Further, it can be hard to decipher the correlation between event rates and IPC. We have found event rates to have a multiplicative effect on the sample configuration IPC, rather than an additive one, so we included a term representing the product of each event rate with IPC. Another limitation is that the derived model cannot capture non-linear effects of predictor variables on the response variable.

Training with ANNs

We also consider models trained using the technique outlined in Section 7.1.1. The ANN-based approach automatically develops a model based on a collection of samples without requiring user-input or domain-specific knowledge. Instead, the modeling process automatically determines possible interactions, which greatly reduces user effort. Further, ANNs are able to capture complicated and non-linear relationships between inputs and outputs. However, as a result, while regression-based models for performance prediction have very low overhead due to their simplicity, ANNs require a relatively large amount of computation. Further, ANNs are treated as black-boxes, which prevents insight into the architecture based on the model.

7.2.2 Results of Comparing ANNs to Regression

In this section, we compare the performance of models trained using multiple linear regression and ANNs. Specifically, we begin by comparing the resulting accuracy of each model, in terms of both absolute accuracy and optimal configuration identification. Then, we compare the resulting performance and energy improvements observed by performing dynamic concurrency throttling using each approach. We performed the experiments presented in this section on the dual processor, Intel quad-core platform described in Section 6.4.1 and evaluated in Section 6.4.2.

Performance Prediction Evaluation

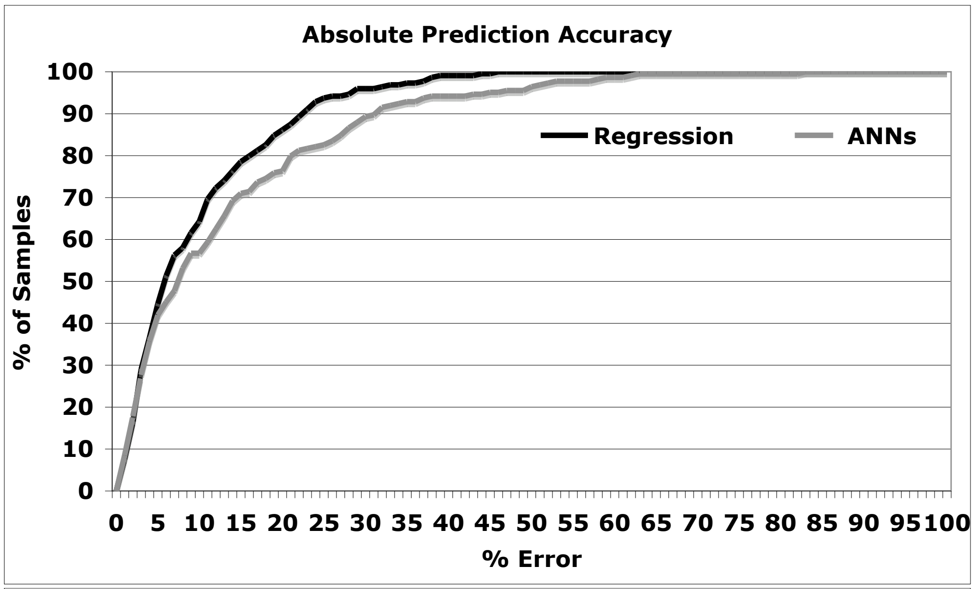
We begin by evaluating the accuracy of our prediction approaches. For model training we use a collection of phases from the NAS-UA benchmark, with the phases from all remaining benchmarks used for evaluation. As sample configurations, we use one at full concurrency (two processors, four cores per processor) with a second using a single processor with three cores active, providing variation in both the number of processors and cores, with predictions made for all of the remaining eight configurations. Our experimental platform only has two hardware event counter registers, on which we record instructions retired and L1 data cache accesses, which we identified as the event with the strongest correlation to IPC.

Figure 7.6 presents the empirical results of both prediction methodologies. The left graph in the figure gives a cumulative distribution function of prediction error, showing the percentage of samples that fall within increasingly higher levels of observed error. For each sample, error is calculated as $|(IPC_{obs} - IPC_{pred})/IPC_{obs}|$, where IPC_{obs} corresponds to the actual measured cumulative IPC and IPC_{pred} corresponds to the cumulative IPC prediced by the model. In the graph, it can be seen that regression is slightly more accurate for scalability predictions in that the percent of samples for regression within any given threshold is always greater than or equal to that for ANNs. Overall, the median error for regression is 5.6% (9.4% mean) and that for

ANNs is 7.5% (12.8% mean), demonstrating a modest advantage for regression. Furthermore, these low error rates occur despite very complex scalability patterns.

An alternative metric for our models is the rate at which they correctly identify the best configuration (or one with very similar performance). The right graph of Figure 7.6 presents the percent of samples for which the approaches select each ranking configuration. This graph shows comparable accuracy for both training approaches. In each case, the predictor identifies nearly optimal configurations in most cases. Specifically, regression correctly identifies the best configuration in 28.6% of phases and selects one of the top five for 85.7% of phases. Similarly, ANNs select the best configuration for 32.1% of phases and on of the top five for 75.0%. ANNs are more effective in identifying the single best operating point, but less so for the top n configurations for all n larger than 1. In phases with limited scalability it becomes more difficult for the models to differentiate between multiple configurations with near-identical performance. At the same time, in such phases, misprediction of the optimal configuration does not harm performance significantly, therefore the overall impact of misprediction is tolerable.

One interesting difference between regression and ANNs is the computational overhead of model evaluation. We find that performing all eight necessary predictions takes approximately 100K cycles with regression and 1M cycles using ANNs for each application phase. While this ten-fold difference is quite large, in neither case is the overhead likely to impact the overall performance of long-running parallel applications. We conclude that in the context of scalability prediction, while both approaches achieve quite high accuracy, regression is slightly more accurate. This rather surprising result can be explained by the relatively small training size, which proves sufficient to derive an accurate model using the simple linear approach, but insufficient for the complicated modeling performed by ANNs.



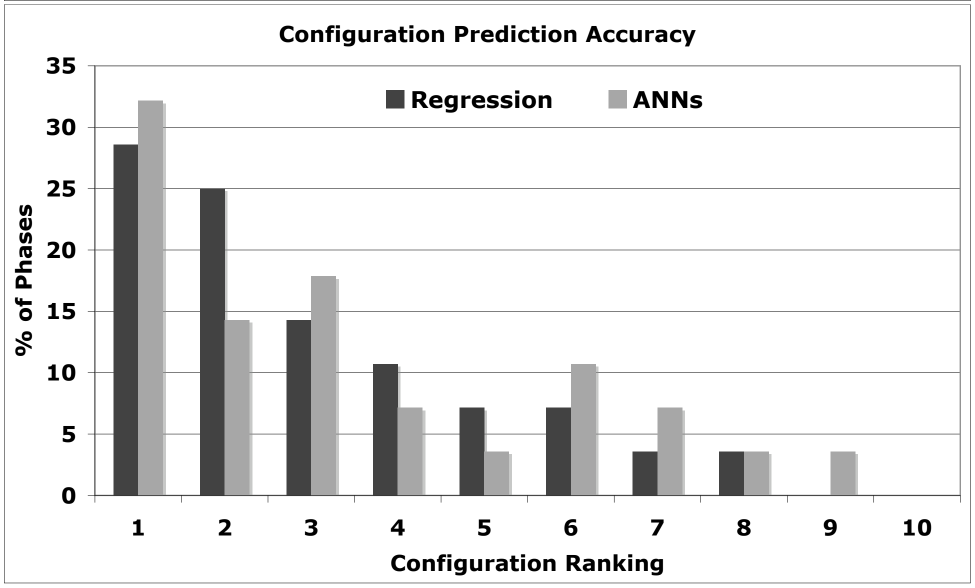


Figure 7.6: Cumulative distribution function (left) and percent of phases for which each ranking configuration is correctly selected for each prediction strategy.

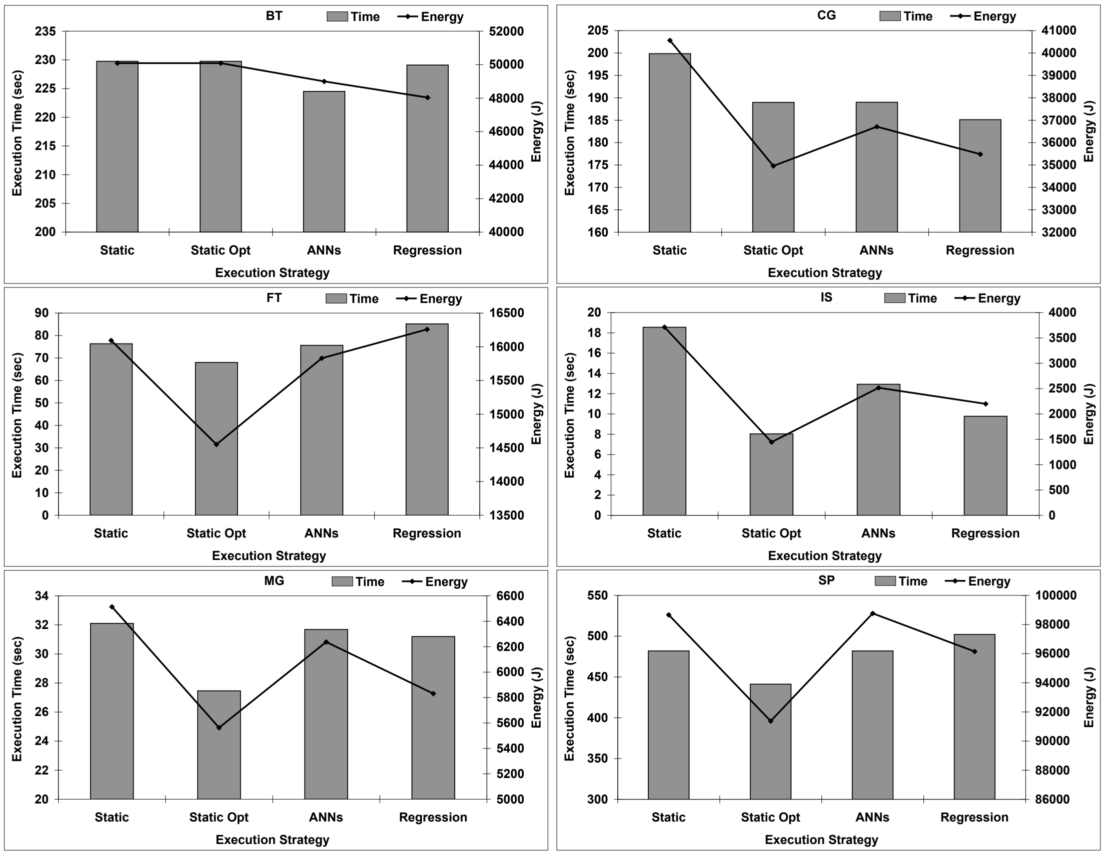


Figure 7.7: Execution time and energy consumption of various execution strategies.

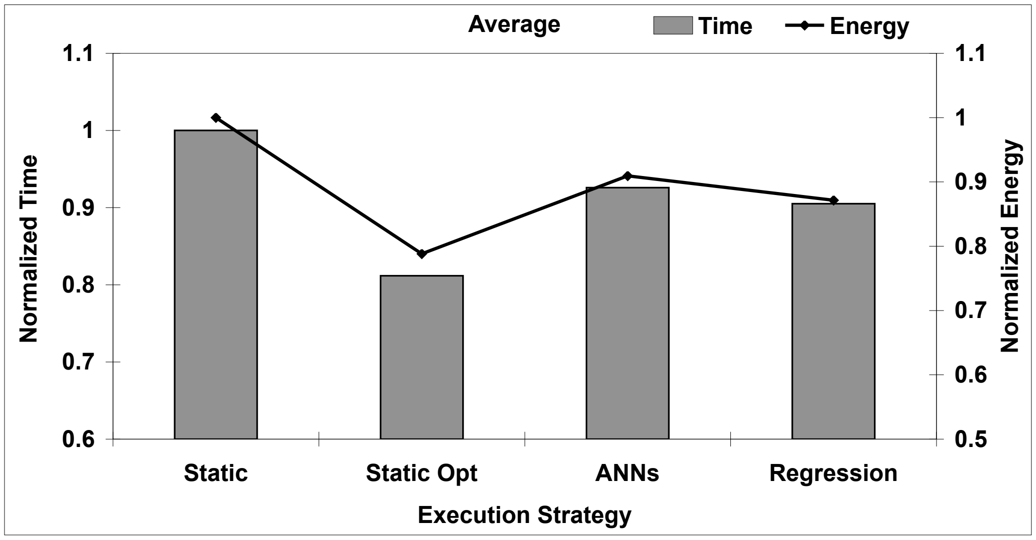


Figure 7.8: Geometric mean of execution time and energy consumption.

Concurrency Throttling Evaluation

Figure 7.7 displays the results of each prediction-based concurrency throttling approach for each evaluation benchmark. Additionally, Figure 7.8 presents the geometric mean of the results normalized by the execution with all available cores. We compare against using all available cores for multithreaded execution, which would be a natural choice for a performance-oriented developer. We also present results for an approach based on the use of oracle-derived configurations (*static optimal*), where we use the best static configuration for an entire application. This information is not easily available since it requires an exhaustive search of the configuration space using complete executions of the applications, however it serves as a point of comparison to evaluate the effectiveness of the library.

DCT using a linear regression predictor yields a mean performance improvement of 9.5% over full concurrency. By comparison, ANNS provide a 7.4% improvement. The small performance advantage for regression-based prediction is justified by the higher (1.9% on average) prediction accuracy that it achieves. The performance of regression-based prediction suggests that a linear model can accurately capture the effects of event rates on performance. If the data did not follow a linear trend, then ANNs would have an advantage even with a small training size and would achieve higher accuracy and better performance than regression. This finding further suggests that the relationship between event rates and parallel scalability on multicore systems is highly linear.

Despite the accuracy and performance advantage of regression overall, each approach is better in three of the six cases. The mean difference between model performance is largely attributed to the IS benchmark, where regression achieves a 47.3% performance improvement, compared to 30.3% for ANNs. In all cases ANNs maintain or improve performance relative to maximum concurrency, while regression sees a 4.2% and 11.6% slowdown for SP and FT respectively. ANNs are also able to achieve a modest 2.3% speedup for BT, which is the one benchmark that scaled fairly well to all cores, demonstrating the potential advantage of performing adaptation at the phase level. Similarly, the linear regression predictor brings the performance of CG beyond the performance achieved by the static optimal configuration. The results of the study of individual benchmarks are inconclusive with respect to the predictors, although it appears that ANNs predict higher scalability better while linear regression predicts low scalability better.

Regression-based prediction optimizes energy consumption better than ANNs. Specifically, the average energy consumption is reduced 9.1% using ANNS versus 12.9% using regression. The relative energy savings of regression over ANNs are based on the 1.9% relative performance gain reported above combined with a 1.9% average power reduction (not shown). The regression-based model achieves lower energy consumption in five of the six benchmarks, suggesting that it more effectively identifies opportunities to conserve power, even if execution time is not always reduced.

Two reasons lead both prediction-based adaptive approaches to fall short of the static optimal configuration in all but one benchmark. First, even though the prediction-based approaches have relatively minimal overhead (the two sample configurations), this overhead can be significant for applications with few iterations; an oracle derives the static optimal so it has no overhead. Second, any prediction-based approach has some error, which limits the potential savings relative to a static offline approach, though offline approaches are infeasible due to large search spaces that much be evaluated for any given program input.

Overall, the regression-based model does outperform the ANN-based model by all metrics. We believe that the modest model specification cost associated with regression is worthwhile for the advantages it provides. Despite slightly better results using regression for model training, these results show that both modeling approaches are quite effective and can provide substantial performance and energy savings.

7.3 Summary

In this chapter, we compared multiple linear regression and artificial neural networks for the prediction of parallel application scalability on a multicore multiprocessors. We based our models on hardware event counters collected at maximum concurrency, using different techniques to train the models and to capture the relationship between hardware counters and scalability. Our experiments yield median error rates of only 5.6% and 7.5% and performance improvements of 9.5% and 7.4% for linear regression and ANNs respectively, with simultaneous improvements in power consumption in both cases. These results demonstrate the potential of dynamic concurrency throttling on multicore systems, and further that the high accuracy achieved by each training methodology results in effective configuration selection. Across all evaluation metrics multiple linear regression proved slightly more effective on average, given the limited training size, which indicates that a linear model sufficiently captures the relationship of hardware events to performance.

This page intentionally left blank.

Chapter 8

Summary and Future Work

8.1 Summary

The performance and power characteristics of applications running on emerging computing systems composed of multithreaded and multicore processors demand consideration of throttling concurrency when scalability bottlenecks result in no performance gain, or a performance loss, from using additional processing elements. In this dissertation, we have presented an approach to dynamic concurrency throttling (DCT) that uses information collected at runtime to predict the performance of an application across various hardware configurations. The system collects hardware event counters to provide insight into the interaction of the hardware and software, allowing the predictor to characterize the performance and scalability of a given program phase. We have considered the use of both multiple linear regression and artificial neural networks for use in model training, and found that both were highly accurate and each had particular advantages over the other. Overall, in the context of scalability prediction, we found multiple linear regression to be slightly more accurate. Further, we have developed a nearly automated training system to provide portability of our prediction model. When tested on a range of architectures using real parallel scientific applications with a wide variety of execution characteristics, the model achieves accuracy approaching 95%. Further, we show that the model is able to identify near-optimal concurrency levels and thread placements in most cases with minimal overhead relative to alternative adaptation strategies.

We have presented an autonomic runtime system (ACTOR) that employs the described performance predictor and have shown that adaptive concurrency throttling can be performed with performance- and energy-effective decisions being made, while keeping the overhead at manageable levels. The described system is shown to outperform adaptation strategies based on empirical searches of the configuration space due to reduced exploration overhead. Finally, the approach is shown to be significantly more effective than simply using all available execution contexts for all phases, in terms of performance, power, and energy consumption. It yields performance results comparable to offline-derived application/input-size specific decisions, and improvements in power and energy, without requiring additional application/input-size specific analysis. Specifically, DCT based on scalability prediction achieves performance gains in the range of 5%–18% across a range of multithreaded and multicore architectures, while simultaneously reducing power consumption by 0%–11%, resulting in energy savings of 5%–27%.

We extended the ACTOR system to provide support for runtime adaptation of DVFS in conjunction with DCT. We accomplished this by expanding the DPAPP predictor to model the effects of adapting DVFS levels as well. In particular, we developed a unified model that is capable of predicting the performance of applying DVFS and DCT simultaneously, capturing the interacting effects of both approaches. We find that adapting DCT and DVFS at the same time using a unified model results in better performance and energy-efficiency than approaches where they are applied sequentially or in isolation.

The approaches presented in this dissertation provide a set of thoroughly evaluated solutions to improve the performance and energy-efficiency of parallel scientific applications executing on multithreaded and multicore systems. Further, we have implemented a runtime system that exploits the various presented techniques to achieve considerable performance improvements and power savings from within real applications on a wide variety of actual machines.

8.2 Future Work

There is a wide variety of future projects based on the work we have performed to-date. In the remainder of this section, we discuss future directions that this work could take.

8.2.1 Extending the Notion of Phases to MPI

In the work performed thus far, we have exploited explicit phase markers in the form of OpenMP directives which specify the occurrence of parallel regions. However, this limits both performance prediction as well as adaptive concurrency throttling to codes parallelized using OpenMP. While a large number of codes destined for SMP execution are written with OpenMP, a majority are written in MPI. To increase the applicability of our work, we could locate effective phase markers within MPI codes for use with prediction and adaptation. Previous work has considered all code between two MPI library calls [38] or between two collective operations to be part of a single phase. Future work should explore strategies for phase identification in terms of their effectiveness in the context of performance prediction and adaptive concurrency throttling.

While performance prediction can be performed at essentially any granularity, concurrency throttling inherently relies on being able to change the number of threads/processes used to execute an application, which is not possible at all granularities. If code regions between two MPI calls or between two MPI collective operations are used to approximate program phases, then the number of MPI processes will not necessarily be able to be changed for each phase. One promising approach is to identify a program-wide optimal level and change the number of MPI processes at runtime using a system such as that presented by Sudarsan, et al. [108].

8.2.2 Thermal Optimization with DCT

When using DCT, cores that are left idle will tend towards a lower operating temperature. This property can be exploited to control the temperature of processors during thermal emergencies, that is, in times when the processor temperature reaches undesirable levels. Specifically, idle cores could be rotated around to all cores in the system at some time interval, and the same could be done at the entire processor level when processors are left idle. This would allow each core to cool-off for some period of time by being left idle. Similar work has been performed by Merkel, et al. [93] to rotate jobs around already idle cores to reduce thermal emergencies. Heo, et al. [49] have also considered migrating two jobs with different thermal properties between cores to prevent prolonged stress on a single CPU component. An alternative target metric could be system temperature, and DCT could be used selectively to minimize this value or keep it below some threshold.

8.2.3 Alternative Uses for Idle Cores

While DCT currently exploits the deactivation of processors, cores, and threads to improve the energy-efficiency of an application, the cores may well be used to improve the performance of the application in some way other than multithreading. Much work has gone into optimizing single-threaded application execution on multicore and multithreaded architectures to take advantage of otherwise idle cores. This work on "helper-threading", as it is called, could be used to take advantage of cores left idle by concurrency throttling. We have already shown in Section 3.4 that the potential performance gains through the use of speculative precomputation on SMT contexts that yield significant slowdowns when used for TLP [29], and it can be expected that similar gains will be seen on other architectures with other techniques. Possible techniques that could be explored include communication offloading, data prefetching, and application self-monitoring. Further, an optimal allocation of cores between the various execution strategies (TLP vs. helper-threading) could be found based on observed execution properties.

8.2.4 Prediction-based Dynamic Space Sharing

As modern systems continue to increase in the number of cores they contain, scalability limitations will make it more efficient to share the system between multiple parallel applications via space-sharing. Here there would be two possibilities for exploration. First, concurrency throttling could proceed as already described for a given target application, and any execution contexts that were found to be detrimental to its execution could be assigned to an alternative application's threads. In this strategy, care would have to be taken to ensure that the supplementary applications do not interfere excessively with the target application. Second, if increased throughput is the goal, rather than peak single application performance, then performance prediction and concurrency throttling could be employed in a different context entirely. Dynamic space sharing of multithreaded applications could be performed by making predictions for all available applications and finding the schedule that yields the highest overall throughput, giving more cores to the application that will benefit the most from them. One difficulty in this approach would be attaining accurate performance predictions of each application on different configurations in the face of interference from competing applications, which requires fully understanding the interaction between parallel applications through detailed performance analysis and modeling.

8.2.5 Transparent Mapping of Applications to Underlying Architecture

In the near future, systems with multiple forms of extremely powerful computing resources – such as massive-scale/heterogenous CMPs, graphics processors, and FPGAs – will be commonplace. Even now, Intel has already announced a working prototype of an 80-core chip [114] and Los Alamos National Lab is acquiring a system, called RoadRunner, composed of both traditional multi-core processors and the Cell B.E. However, applications will have to be able to adapt their execution along many dimensions to achieve peak performance or desired thermal/power properties on these complex systems. An autonomous system could be used to monitor applications at runtime and transparently adapt all aspects of execution, including identifying the optimal computing substrate from those available, managing power/performance tradeoffs, and finding effective uses for idle resources. Such an approach is capable of defeating many of the limitations of modern systems, including the memory, power, and thermal walls, by altering execution to overcome these issues. Further, this research can reduce the burden on the developer of being aware of architectural details, as compiler analysis, automatic binary instrumentation, and a sophisticated runtime system will manage all necessary optimizations.

Appendix A

Implementation of PACMAN

Hardware event counters (HECs) are registers within a processor that can be programmed to count the occurrences of particular processor events, such as L2 cache misses, stall cycles, etc. Due to the insight they provide into the execution of an application on a given architecture, hardware event counters are seeing increasing popularity in both the research [9] and industrial communities [34].

Despite the pervasiveness of Intel Hyperthreaded processors [72], the support for collection of hardware event counters on this architecture is limited. Tools designed to work on single-threaded processors fail to provide sufficient functionality when ported over. The difficulty stems from the sharing of the performance monitoring unit (PMU) between the two execution contexts on Hyperthreaded Pentium 4 processors. Perfctr [98], the standard interface to Pentium 4 event counters for Linux, overcomes this problem by disallowing the use of the second execution context on each processor when collecting events in *per-thread* mode [1]. PAPI [9], being built on top of Perfctr, suffers from the same problems. Intel's VTune Performance Analyzer [53] provides thread-local event counter statistics offline, however it does not provide functionality for online and accurate event counter collection. It is important that applications

[1]In *per-thread* mode event counters are recorded separately for each thread, whereas in *global* mode counters are recorded for all processors in the system with all executing threads included.

be able to use all available contexts while still exploiting the full set of hardware event counting features at runtime.

In order to use event counters for online adaptation of applications, it is necessary to be able to retrieve per-thread counter values from within the target application during its execution. Beyond this, it is also necessary to have fine-grain access to counter values to support monitoring of short-lived regions of code. In the following section, we discuss how we enabled such functionality in PACMAN (available at http://people.cs.vt.edu/˜mfcurt).

A.1 PACMAN Implementation

An Intel Hyperthreaded processor has 18 HEC registers that are shared between the two co-executing threads. Each register can record a single event. Should two threads on the same processor attempt to use the same register, only one thread's configuration would ultimately be used. PACMAN prevents overlapping of HEC register usage between co-executing threads by introducing a logical partitioning of the registers when events are recorded in per-thread collection mode. Within each processor, half of the HECs are provisioned to each execution context. Intel has divided the registers into four sets and any given event can only be recorded within its designated set. We create our partition such that each set of registers is divided evenly between the two threads. There are also configuration registers associated with the HECs which we have partitioned similarly. During event counter initialization each thread is configured to use the partition for the execution context on which it is currently executing. To prevent a thread from migrating away from the execution context for which its HECs were set up, PACMAN appropriately binds threads to execution contexts.

PACMAN uses the interface provided by Perfctr for low-level access to the hardware event counters, after removal of the internal checks from Perfctr that enforce usage and monitoring of only the first execution context on each processor. In addition to the extended support for Hyperthreaded processors, PACMAN retains the full functionality already present in Perfctr.

This includes the use of event counters with only a single thread per processor using all 18 available HECs, as well as global mode collection.

Although global collection does allow events to be collected on both execution contexts, even in Perfctr, there are two shortcomings of this approach for online use. First, event counts summarize collective performance of all executing applications, not just the one to be monitored. Second, the granularity of monitored regions must be very coarse (on the order of hundreds of milliseconds at least) since results are stored in the operating system and are only periodically updated between consecutive time quanta. These were motivating factors in the development of PACMAN.

The process of initializing Pentium 4 event counters can be cumbersome. One hurdle to using hardware event counters is the creation of the bitmasks written to specific registers as part of the configuration process. Although Perfctr does abstract away the manual process of loading the registers, it must be given the exact contents for each desired register. These values specify what events to record, but they also provide additional constraints on recording, and creating the desired bitmasks can be a very complicated process. Further, for any event to be recorded, a configuration register must be specified and correctly initialized, however, only certain configuration registers are legal for a given event. PACMAN simplifies the initialization process by having predefined values for any desired event, which allow the user to specify an event to be recorded by an intuitive name, such as STALL_CYCLES, and handles the low-level details internally. Another difficulty is that, due to sharing of the PMU, a given HEC register cannot be allocated to both co-executing threads. However, since registers are allocated in PACMAN according to the partitioning scheme described above, this problem is overcome as well. In these ways, PACMAN greatly reduces the difficulty of configuring HEC registers.

This page intentionally left blank.

Bibliography

[1] NR Adiga, G Almasi, GS Almasi, Y Aridor, R Barik, D Beece, R Bellofatto, G Bhanot, R Bickford, M Blumrich, AA Bright, J Brunheroto, C Ca?caval, J Castaos, W Chan, L Ceze, P Coteus, S Chatterjee, D Chen, G Chiu, TM Cipolla, P Crumley, KM Desai, A Deutsch, T Domany, MB Dombrowa, W Donath, M Eleftheriou, C Erway, J Esch, B Fitch, J Gagliano, A Gara, R Garg, R Germain, ME Giampapa, B Gopalsamy, J Gunnels, M Gupta, F Gustavson, S Hall, RA Haring, D Heidel, P Heidelberger, LM Herger, D Hoenicke, RD Jackson, T Jamal-Eddine, GV Kopcsay, E Krevat, MP Kurhekar, AP Lanzetta, D Lieber, LK Liu, M Lu, M Mendell, A Misra, Y Moatti, L Mok, JE Moreira, BJ Nathanson, M Newton, M Ohmacht, A Oliner, V Pandit, RB Pudota, R Rand, R Regan, B Rubin, A Ruehli, S Rus, RK Sahoo, A Sanomiya, E Schenfeld, M Sharma, E Shmueli, S Singh, P Song, V Srinivasan, BD Steinmacher-Burow, K Strauss, C Surovic, R Swetz, T Takken, RB Tremaine, M Tsao, AR Umamaheshwaran, P Verma, P Vranas, TJC Ward, M Wazlowski IBM Research W Barrett, C Engel, B Drehmel, B Hilgart, D Hill, F Kasemkhani, D Krolak, CT Li, T Liebsch, J Marcella, A Muff, A Okomo, M Rouse, A Schram, M Tubbs, G Ulsh, C Wait, J Wittrup, M Bae, K Dockser, L Kissel, MK Seager, JS Vetter, and K Yates. An Overview of the BlueGene/L Supercomputer. In *Proc. of the IEEE/ACM Supercomputing'2002: High Performance Networking and Computing Conference*, Baltimore, MD, November 2002.

[2] D. Albonesi. Selective Cache Ways: On-demand Cache Resource Allocation. In *Proc. of the 32nd International Symposium on Microachitecture*, November 1999.

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

[4] K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W.L. Plishker, J. Shalf, S.W. Williams, and Katherine A. Yelick. The Landscape of

Parallel Computing Research: A View from Berkeley. Technical report ucb/eecs-2006-183, EECS Department, University of California at Berkeley, December 2006.

[5] 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.

[6] L. Barroso. The Price of Performance: An Economic Case for Chip Multiprocessing. *ACM Queue*, 3(7), September 2005.

[7] C. Bekas and E. Gallopoulos. Cobra: Parallel Path Following for Computing the Matrix Pseudospectrum. *Parallel Computing*, 27(8):1879–1896, July 2001.

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

[9] 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 the International Conference on High-Performance Computers and Communication (Supercomputing)*, November 2000.

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

[11] L. Carrington, A. Snavely, X. Gao, and N. Wolter. A Performance Prediction Framework for Scientific Applications. In *Workshop on Performance Modeling - ICCS*, 2003.

[12] L. Carrington, N. Wolter, A. Snavely, and C.B. Lee. Applying an Automatic Framework to Produce Accurate Blind Performance Predictions of Full-Scale HPC Applications. In *Department of Defense Users Group Conference*, June 2004.

[13] R. Caruana, S. Lawrence, and C.L. Giles. Overfitting in Neural Nets: Backpropagation, Conjugate Gradient, and Early Stopping. In *Proc. Neural Information Processing Systems Conference*, October 2000.

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

[15] J. Cavazos, C. Dubach, F. Agakov, E. Bonilla, M. O'Boyle, G. Fursin, and O. Temam. Automatic Performance Model Construction for the Fast Software Exploration of New Hardware Designs. In *Proc. International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, Seoul, Korea, October 2006.

[16] 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.

[17] K. Choi, R. Soma, and M. Pedram. Fine-grained Dynamic Voltage and Frequency Scaling for Precise Energy and Performance Trade-off Based on the Ratio of Off-chip Access to On-chip Computation Times. In *Proc. of Conference on Design, Automation, and Test in Europe*, Washington, DC, February 2004.

[18] J. Collins, H. Wang, D. Tullsen, C. Hughes, Y. Lee, D. Lavery, and J. Shen. Speculative Precomputation: Long-Range Prefetching of Delinquent Loads. In *Proc. of the 28th Annual International Symposium on Computer Architecture (ISCA–2001)*, pages 14–25, Goteborg, Sweden, July 2001.

[19] M. Curtis-Maury, C. Antonopoulos, and D. Nikolopoulos. PACMAN: A PerformAnce Counters MANager for Intel Hyperthreaded Processors. In *Proc. of the IEEE International Conference on the Quantitative Evaluation of Systems*, Riverside, CA, September 2006.

[20] M. Curtis-Maury, C. Antonopoulos, and D. Nikolopoulos. A Comparison of Online and Offline Strategies for Program Adaptation. In *Proc. of the 45th ACM Southeast Conference*, Winston-Salem, NC, March 2007.

[21] 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.

[22] M. Curtis-Maury, X. Ding, C. Antonopoulos, and D. Nikolopoulos. An Evaluation of OpenMP on Current and Emerging Multithreaded/Multicore Architectures. In *Proc. of the First International Workshop on OpenMP*, Eugene, Oregon, June 2005.

[23] M. Curtis-Maury, J. Dzierwa, C. Antonopoulos, and D. Nikolopoulos. On the Design of Online Predictors for Autonomic Power-Performance Adaptation of Multithreaded Codes. *Journal of Autonomic and Trusted Computing*, to appear.

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

[25] M. Curtis-Maury, J. Dzierwa, C. Antonopoulos, and D. Nikolopoulos. Online Strategies for High-Performance Power-Aware Thread Execution on Emerging Multiprocessors. In *Proc. of the Second Workshop on High-Performance Power-Aware Computing*, Rhodes, Greece, April 2006.

[26] M. Curtis-Maury, A. Shah, F. Blagojevic, D. Nikolopoulos, B. R. de Supinski, and M. Schulz. Prediction Models for Multi-Dimensional Power-Performance Adaptation on Many Cores. In *Submitted*.

[27] M. Curtis-Maury, K. Singh, S. A. McKee, F. Blagojevic, D. S. Nikolopoulos, B. R. de Supinski, and M. Schulz. Identifying Energy-Efficient Concurrency Levels Using Machine Learning. In *Proc. of the International Workshop on Green Computing*, September 2007.

[28] M. Curtis-Maury, K. Singh, A. Shah, F. Blagojevic, D. Nikolopoulos, S. A. McKee, B. R. de Supinski, and M. Schulz. Comparing Scalability Prediction Strategies on an SMP of CMPs. In *Submitted*.

[29] M. Curtis-Maury, T. Wang, C. Antonopoulos, and D. Nikolopoulos. Integrating Multiple Forms of Multithreaded Execution on multi-SMT Systems: A Study with Scientific Workloads. In *Proc. of the Second International Conference on the Quantitative Evaluation of Systems*, Turin, Italy, September 2005.

[30] P. de Langen and B. Juurlink. Leakage-Aware Multiprocessor Scheduling for Low Power. In *Proc. of the 20th International Parallel and Distributed Processing Symposium*, Rhodes, Greece, April 2006.

[31] A. Dhodapkar and J. Smith. Managing Multi-Configuration Hardware via Dynamic Working Set Analysis. In *Proc. of the 29th Annual International Symposium on Computer Architecture*, May 2002.

[32] C. Dubach, J. Cavazos, B. Franke, G. Fursin, M. F.P. O'Boyle, and O. Temam. Fast Compiler Optimisation Evaluation Using Code-Feature Based Performance Prediction. In *CF '07: Proceedings of the 4th International Conference on Computing Frontiers*, pages 131–142, New York, NY, USA, 2007. ACM Press.

[33] L. Eeckhout and K. De Bosschere. Statistical Simulation of Superscalar Architectures using Commercial Workloads. In *Proc. of the Fourth Workshop on Computer Architecture Evaluation using Commercial Workloads*, Monterrey, Mexico, January 2001.

[34] S. Eranian. The Perfmon2 Interface Specification. Technical Report HPL-2004-200R1, HP Labs, February 2005.

[35] F. Dahlgren and H. Grahn and M. Karlsson and F. Larsson and F. Lundholm and A. Moestedt and J. Nilsson and P. Stenström and and B. Werner. Simics/sun4m: A virtual workstation. In *Proc. of the 1998 USENIX Annual Technical Conference*, New Orleans, LA, June 1998.

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

[37] X. Feng, R. Ge, and K. Cameron. Power and Energy Profiling of Scientific Applications on Dsitributed Systems. In *Procȯf the 19th IEEE International Parallel and Distributed Processing Symposium*, Denver, CO, April 2005.

[38] 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.

[39] V. Freeh, F. Pan, D. Lowenthal, N. Kappiah, R. Springer, B. Rountree, and M. Femal. Analyzing the Energy-Time Tradeoff in High-Performance Computing Applications. *Transactions on Parallel and Distributed Systems*, 5(11), May 2006.

[40] R. Ge and K. W. Cameron. Power-Aware Speedup. In *Proc. of the 21st IEEE International Parallel and Distributed Processing Symposium*, March 2007.

[41] R. Ge, X. Feng, and K. Cameron. Improvement of Power-Performance Efficiency for High-End Computing. In *Proc. of the Workshop on High-Performance, Power-Aware Computing*, Denver, CO, April 2005.

[42] R. Ge, X. Feng, and K. Cameron. Performance-contrained Distributed DVFS Scheduling for Scientific Applications on Power-aware Clusters. In *Proc. of the 17th IEEE/ACM High-Performance Computing, Networking, and Storage Conference (SC'05)*, Seattle, WA, November 2005.

[43] R. Ge, X. Feng, W. Feng, , and K. 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.

[44] K. Ghose and M. Kamble. Reducing Power in Superscalar Processor Caches using Subbanking, Multiple Line Buffers and Bit-line Segmentation. In *Proc. of the International Symposium on Low Power Electronics and Design*, August 1999.

[45] M. Gonzalez, J. Oliver, X. Martorell, E. Ayguad'e, J. Labarta, and N. Navarro. OpenMP Extensions for Thread Groups and Their Run-time Support. In *Proc. of the 13th International Workshop on Languages and Compilers for Parallel Computing (LCPC'2000)*, pages 317–331, New York, NY, August 2000.

[46] G. A. Grell, J. Dudhia, and D. R. Stauffer. A Description of the Fifth-Generation Penn State/NCAR Mesoscale Model (MM5). NCAR Technical Note NCAR/TN-398 + STR, National Center For Atmospheric Research (NCAR), June 1995.

[47] S. Gurumurthi, A. Sivasubramaniam, M. Kandemir, and H. Franke. Reducing Disk Power Consumption in Servers with DRPM. *IEEE Computer*, December 2003.

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

[49] S. Heo, K. Barr, and K. Asanovic. Reducing Power Density through Activity Migration. In *Proc. of the International Symposium on Low Power Electronics and Design*, Seoul, Korea, August 2003.

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

[51] C. Hsu and W. Feng. A Power-Aware Run-Time System for High-Performance Computing. In *Proc. of the the ACM/IEEE International Conference on High-Performance Computing, Networking, and Storage (Supercomputing)*, Seattle, WA, September 2005.

[52] 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.

[53] Intel Inc. Intel VTune Performance Analyser. http://www.intel.com/software/products/vtune, 2003.

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

[55] C. Isci, A. Buyuktosunoglu, C. Cher, P. Bose, and M. Martonosi. An Analysis of Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget. In *Proc. of the 39th International Symposium on Microarchitecture*, December 2006.

[56] C. Isci, G. Contreras, and M. Martonosi. Live, Runtime Phase Monitoring and Prediction on Real Systems with Application to Dynamic Power Management. In *Proc. of the 39th International Symposium on Microarchitecture*, December 2006.

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

[58] H. Jin, M. Frumkin, and J. Yan. The OpenMP Implementation of NAS Parallel Benchmarks and its Performance. Technical report nas-99-011, NASA Ames Research Center, October 1999.

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

[60] P. Joseph, K. Vaswani, and M. Thazhuthaveetil. A Predictive Performance Model for Superscalar Processors. In *Proc. of the 39th International Symposium on Microarchitecture*, December 2006.

[61] P. Joseph, K. Vaswani, and M. Thazhuthaveetil. Contruction and Use of Linear Regression Models for Processor Performance Analysis. In *Proc. of the 12th International Symposium on High-Performance Computer Architecture*, February 2006.

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

[63] I. Kadayif, M. Kandemir, N. Vijaykrishnan, M. Irwin, and I. Kolcu. Exploiting Processor Workload Heterogeneity for Reducing Energy Consumption on Chip Multiprocessors. In *Proc. of the Design Automation and Test in Europe Conference*, February 2004.

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

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

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

[67] 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.

[68] D. Kerbyson, H. Alme, A. Hoisie, F. Petrini, H. Wasserman, and M. Gittings. Predictive Performance and Scalability Modeling of a Large-Scale Application. In *Proc. International Conference on High Performance Computing and Communications*, November 2001.

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

[70] M. Kondo and H. Nakamura. A Small, Fast and Low-Power Register File by Bit-Partitioning. In *Proc. of the 11th International Symposium on High-Performance Computer Architecture*, San Francisco, CA, February 2005.

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

[72] D. Koufaty and D. Marr. Hyperthreading Technology in the Netburst Microarchitecture. *IEEE Micro*, 23(2):56–65, March 2003.

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

[74] 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.

[75] A. Lebeck, X. Fan, H. Zeng, and C. Ellis. Power Aware Page Allocation. In *Proc. of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[76] B. Lee. An Architectural Assessment of SPEC CPU Benchmark Relevance. Technical Report TR-02-06, Harvard University, January 2006.

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

[78] B. Lee and D. Brooks. Regression Modeling Strategies for Microarchitectural Performance and Power Prediction. Technical Report TR-08-06, Harvard University, March 2006.

[79] B. Lee and D. Brooks. Illustrative Design Space Studies with Microarchitectural Regression Models. In *Proc. of the 13th International Symposium on High-Performance Computer Architecture*, February 2007.

[80] B. Lee, D. 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 International Symposium on Principles and Practices of Parallel Programming*, March 2007.

[81] B. Levine. Kilocore: Scalable, High-Performance, and Power Efficient Coarse-Grained Reconfigurable Fabrics. In *Proc. of the International Symposium on Advanced Reconfigurable Systems*, December 2005.

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

[83] J. Li and J. Martínez. Dynamic Power-Performance Adaptation of Parallel Computation on Chip Multiprocessors. In *Proc. of the 12th International Symposium on High-Performance Computer Architecture*, Austin, TX, February 2006.

[84] T. Li, A. Lebeck, and D. Sorin. Spin Detection Hardware for Improved Management of Multithreaded Systems. *IEEE Transactions on Parallel and Distributed Systems*, 17(6), November 2006.

[85] Y. Li, B. C. Lee, D. Brooks, 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.

[86] M. Lim, V. Freeh, and D. Lowenthal. Transparent Frequency and Voltage Scaling of Communication Phases in MPI Programs. In *Proceedings of IEEE/ACM Supercomputing*, Tampa, FL, November 2006.

[87] C. Liu, A. Sivasubramaniam, M. Kandemir, and M. Irwin. Exploiting Barriers to Optimize Power Consumption on CMPs. In *Proc. of the 19th International Parallel and Distributed Processing Symposim*, Denver, CO, April 2005.

[88] J. Lo, J. Emer, H. Levy, R. Stamm, D. Tullsen, and S. Eggers. Converting Thread-Level Parallelism to Instruction-Level Parallelism via Simultaneous Multithreading. *ACM Transactions on Computer Systems*, 15(3):322–353, August 1997.

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

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

[91] G. Marin and J. Mellor-Crummey. Cross-Architecture Performance Predictions for Scientific Applications Using Parameterized Models. In *Proc.ACM International Conference on Measurement and Modeling of Computer Systems*, pages 2–13, June 2004.

[92] D. Marr, F. Binns, D. Hill, G. Hinton, K. Koufaty, J. Miller, and M. Upton. Hyperthreading Technology Architecture and Microarchitecture. In *Intel Technical Journal, 1(1)*, February 2002.

[93] A. Merkel and F. Bellosa. Balancing Power Consumption in Multiprocessor Systems. In *First ACM SIGOPS EuroSys Conference*, Leuven, Belgium, April 2006.

[94] T.M. Mitchell. *Machine Learning*. WCB/McGraw Hill, Boston, MA, 1997.

[95] T. Moseley, J. Kim, D. Connors, and D. Grunwald. Methods for Modelling Resource Contention on Simultaneous Multithreaded Processors. In *Proc. of the 2005 International Conference on Computer Design*, pages 373–380, San Jose, CA, October 2005.

[96] OpenMP Architecture Review Board. OpenMP Application Programming Interface. Version 2.5, May 2005.

[97] D. Parikh, K. Skadron, Y. Zhang, M. Barcella, and M. Stan. Power Issues Related to Branch Prediction. In *Proc. of the 8th International Symposium on High-Performance Computer Architecture*, Cambridge, MA, February 2002.

[98] M. Pettersson. A Linux/x86 Performance Counters Driver. Technical report. http://user.it.uu.se/∼mikpe/linux/perfctr/.

[99] 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.

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

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

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

[103] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder. Discovering and Exploiting Program Phases. *IEEE Micro*, 23(6):84–93, 2003.

[104] T. Sherwood, S. Sair, and B. Calder. Phase Tracking and Prediction. In *Proc. of the 30th Annual International Symposium on Computer Architecture*, May 2003.

[105] K. Singh, E. Ipek, S.A. McKee, B.R. de Supinski, M. Schulz, and R. Caruana. Predicting Parallel Application Performance via Machine Learning Approaches. *Concurrency and Computation: Practice and Experience*, 19(17), May 2007.

[106] A. Snavely, L.Carrington, N.Wolter, J.Labarta, R.Badia, and A.Purkayastha. A Framework for Application Performance Modeling and Prediction. In *Supercomputing*, 2002.

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

[108] R. Sudarsan and C. J. Ribbens. ReSHAPE: A Framework for Dynamic Resizing and Scheduling of Homogeneous Applications in a Parallel Environment. In *Proc. of the Proceedings of the International Conference on Parallel Processing*, September 2007.

[109] M. A. Suleman, M.K. Qureshi, and Y.N. Patt. Feedback-Driven Threading: Power-Efcient and High-Performance Execution of Multi-threaded Workloads on CMPs. In *Proc. of the Proceedings of the International Symposium on Architectural Support for Programming Languages and Operating Systems*, March 2008.

[110] M. Tolentino, J. Turner, and K. Cameron. Memory-MISER: A Performance-Constrained Runtime System for Power-Scalable Clusters. In *Proc. of ACM Computing Frontiers*, Ischia, Italy, May 2007.

[111] N. Tuck and D. Tullsen. Initial Observations of the Simultaneous Multithreading Pentium 4 Processor. In *Proc. of the 2003 International Conference on Parallel Architectures and Compilation Techniques (PACT'2003)*, New Orleans, LA, September 2003.

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

[113] D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proc. of the 22nd Annual International Symposium on Computer Architecture*, June 1995.

[114] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain S. Venkataraman, Y. Hoskote, and N. Borkar. An 80-tile 1.28TFLOPS Network-on-Chip in 65nm CMOS. In *Proc. of the International Solid State Circuits Conference*, pages 5–7, San Francisco, CA, 2007.

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

[116] H. Wang, P. Wang, R. Weldon, S. Ettinger, H. Saito, M. Girkar, S. Liao, and J. Shen. Speculative Precomputation: Exploring the Use of Multithreading for Latency. *Intel Technology Journal*, 6(1), February 2002.

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

[118] 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), January 2006.

[119] T. Yang, X. Ma, and F. Mueller. Cross-platform performance prediction of parallel applications using partial execution. In *Proc. IEEE/ACM Supercomputing International Conference on High Performance Computing, Networking, Storage and Analysis*, November 2005.

[120] J. Yi, D. Lilja, and D. Hawkins. Improving Computer Architecture Simulation Methodology by Adding Statistical Rigor. *IEEE Transactions on Computers*, 54(11), November 2005.

[121] R. M. Yoo, H. Lee, K. Chow, and H.-H. S. Lee. Constructing a Non-Linear Model with Neural Networks for Workload Characterization. In *IISWC*, 2006.

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

[123] Y. Zhang, M. Burcea, V. Cheng, R. Ho, V. Cheng, and M. Voss. An Adaptive OpenMP Loop Scheduler for Hyperthreaded SMPs. In *Proc. of PDCS-2004: International Conference on Parallel and Distributed Computing Systems*, San Francisco, CA, September 2004.

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