

ABSTRACT

LIM, MIN YEOL. Improving Power and Performance Efficiency in Parallel and Distributed Computing Systems. (Under the direction of Dr. Vincent W. Freeh.)

For decades, high-performance computing systems have focused on increasing maximum performance at any cost. A consequence of the devotion towards boosting performance significantly increases power consumption. The most powerful supercomputers require up to 10 megawatts of peak power – enough to sustain a city of 40,000. However, some of that power may be wasted with little or no performance gain, because applications do not require peak performance all the time. Therefore, improving power and performance efficiency becomes one of the primary concerns in parallel and distributed computing. Our goal is to build a runtime system that can understand power-performance tradeoffs and balance power consumption and performance penalty adaptively.

In this thesis, we make the following contributions. First, we develop a MPI runtime system that can dynamically balance power and performance tradeoffs in MPI applications. Our system dynamically identifies power saving opportunities without prior knowledge about system behaviors and then determines the best p-state to improve the power and performance efficiency. The system is entirely transparent to MPI applications with no user intervention. Second, we develop a method for determining minimum energy consumption in voltage and frequency scaling systems for a given time delay. Our approach helps to better analyze the performance of a specific DVFS algorithm in terms of balancing power and performance. Third, we develop a power prediction model that can correlate power and performance data on a chip multiprocessor machine. Our model shows that the power consumption can be estimated by hardware performance counters with reasonable accuracy in various execution environments. Given the prediction model, one can make a runtime decision of balancing power and performance tradeoffs on a chip-multiprocessor machine without delay for actual power measurements. Last, we develop an algorithm to save power by dynamically migrating virtual machines and placing them onto fewer physical machines depending on workloads. Our scheme uses a two-level, adaptive buffering scheme which reserves processing capacity. It is designed to adapt the buffer sizes to workloads in order to balance performance violations and energy savings by reducing the amount of energy wasted on the buffers. Our simulation framework justifies our study of the energy benefits and the performance effects of the algorithm along with studies of its sensitivity to various parameters.

Improving Power and Performance Efficiency in Parallel and Distributed Computing
Systems

by
Min Yeol Lim

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2009

APPROVED BY:

Dr. George N. Rouskas

Dr. Gregory T. Byrd

Dr. Vincent W. Freeh
Chair of Advisory Committee

Dr. Xiaosong Ma

Dr. Robert J. Fowler

DEDICATION

To my beautiful wife, *Younghee Park*,
my adorable son, *Jaehyun*,
and
my parents for their support and love . . .

BIOGRAPHY

Min Yeol Lim was born in Daegu, South Korea in 1974. He begun studying computing engineering at Kyungpook National University in Daegu, South Korea in 1993, and, after serving Korea military duty from 1995 to 1997, graduated with a Bachelor of Engineering degree in 2000. Then, he continued his graduate study for a master degree at Korea Advanced Institute of Science and Technology in Daejeon, South Korea. For two years, he focused on researching and developing a load-balancing framework at Web application server farm, which is also the topic of his Master's thesis. After receiving the Master of Science degree in 2002, he worked at Supercomputing Center of Korea Institute of Science and Technology Information where he conducted Grid computing research. He began his doctorate study in the Department of Computer Science at North Carolina State University in 2004. Since he joined the Operating System Research lab directed by Dr. Vincent W. Freeh in 2005, he has focused his research on power/performance analysis and optimization techniques for parallel and distributed computing systems. On August 14, 2009, he successfully defended his Ph.D. dissertation.

ACKNOWLEDGMENTS

I would like to thank all the people who helped me with this work. This dissertation could not have been finished without their support.

First and foremost, I would like to appreciate Dr. Vincent W. Freeh who served as my advisor as well as a mentor. It was an honor working with such a nice professor like Dr. Freeh. His enthusiasm for the work kept me motivated and he always guided me to stay in the right direction to my goal. He also supported me with kind advice and guidance on how to improve my spoken and written skills. I would not have finished my degree on time without him. I truly appreciate his tireless support at all levels.

I also would like to thank Dr. Xiaosong Ma, Dr. George Rouskas, Dr. Gregory Byrd, and Dr. Robert Fowler for being members of my committee. They provided relevant, insightful guidance to make good progress throughout my research. Their valuable comments helped me focus on important issues and look at the big picture in my research.

I am grateful to many collaborators during my Ph.D. pursuit. Dr. David Lowenthal at University of Arizona helped me take the first step to this dissertation. Through the collaboration with him for my first paper, I learned how to conduct research in my area. Dr. Allan Porterfield was a great mentor during my summer internship and academic year support at Renaissance Computing Institute. I was able to make a significant progress and to broad my knowledge in my research by his support. I also thank Daniel Bedard at Renaissance Computing Institute who developed PowerMon device for his support in my research. I am grateful to Dr. Freeman Rawson at IBM who helped the PADD research. I sincerely appreciate his invaluable advice and comments.

Finally, I would like to thank Tyler Bletsch during the years. He is a smart person who has a lot of knowledge in computing systems. He was always willing to discuss my research and gave great advice on this dissertation. He helped me understand and utilize all the systems in our lab when starting my research from scratch.

I sincerely acknowledge the funding resources of my dissertation work: (1) NSF grants (CCF-0234285 and CCF-0429643), (2) Dr. Vincent Freeh's University Partnership award from IBM, (3) PERI - Performance Engineering Research Center (PERC-3) DE-FC02-06ER25764, and (4) Dr. Vincent Freeh's RENCI Award.

TABLE OF CONTENTS

LIST OF TABLES	vii
LIST OF FIGURES	ix
1 Introduction	1
1.1 Power and performance tradeoffs	2
1.2 Accomplishments	3
1.3 Contributions	5
2 Background.....	6
2.1 Energy savings in server systems	6
2.2 Energy savings in HPC systems	7
2.3 Energy savings in server farms	9
3 Adaptive, Transparent CPU Scaling Algorithms in MPI Communications.....	12
3.1 Motivation	13
3.2 Design and implementation	18
3.2.1 The training component	18
3.2.2 The shifting component	22
3.3 Results	24
3.3.1 Overall results	28
3.3.2 Detailed analysis	32
3.4 Summary	40
4 Determining the Minimum Energy Consumption	41
4.1 Approach	42
4.1.1 Modeling	43
4.1.2 Scheduling algorithm	44
4.2 Implementation	46
4.2.1 Time and energy data collection	46
4.2.2 Power measurements	48
4.2.3 Finding regions	49
4.3 Results	50
4.3.1 Detailed analysis	51
4.3.2 Discussion	54
4.4 Summary	55

5	Fine-grained and Accurate Power Measurements	56
5.1	Design	58
5.2	Evaluation	60
5.2.1	Access latency and accuracy	60
5.2.2	Fine-grained power measurement	62
5.2.3	Power monitoring overhead	66
5.2.4	Power supply efficiency	67
5.3	Summary	69
6	Power Prediction on Multi-core system via Performance Counters.....	70
6.1	Approach	71
6.1.1	Measuring power and performance	71
6.1.2	Profiling power and performance	71
6.1.3	Selection of performance counters	72
6.1.4	Applying multivariate linear regression	72
6.2	Evaluation	74
6.3	Summary	80
7	PADD: Power Aware Domain Distribution.....	81
7.1	Model	83
7.1.1	Machine State and virtualization	83
7.1.2	Model description	84
7.2	Implementation	86
7.2.1	Buffering	87
7.2.2	Time cost of transitions	88
7.2.3	Additional parameters	89
7.2.4	Algorithm	90
7.3	Evaluation	91
7.3.1	Workload generation	91
7.3.2	System configuration	92
7.3.3	Analysis	93
7.4	Summary	98
8	Concluding remarks.....	100
8.1	A runtime power-performance directed adaptation	100
8.2	Dynamic power efficiency adaptation on multi-core architecture	101
8.3	Conclusion	103
	Bibliography	104

LIST OF TABLES

Table 3.1 Analysis of MPI call length and interval between MPI calls per benchmark. Benchmarks are sorted by 75th percentile in ascending order.	17
Table 3.2 P-state transition cost (microseconds) for a AMD Athlon-64 3000+ Processor [1]. The frequency in a row indicates a starting p-state and the frequency in a column is a target p-state.	24
Table 3.3 Benchmark attributes. Region information from <i>composite</i> with $\tau = 20$ ms and $\lambda = 20$ ms.....	25
Table 3.4 Overall performance of benchmark programs. Where appropriate, the value relative to the base case is also reported.	29
Table 3.5 Performance comparison between single p-state and automatic multiple p-state selection.....	33
Table 4.1 Analysis per region in LU benchmark	44
Table 4.2 The workload analysis of finding the best schedule	51
Table 4.3 Estimation verification.....	52
Table 5.1 Comparison of PowerMon and Power Egg access latencies. Power Egg requires at least 480 ms between requests.	60
Table 5.2 Measured hard disk power consumption	65
Table 5.3 Power supply efficiency by CPU loads on a quad-core AMD Phenom machine	67

Table 5.4 Power supply efficiency by CPU loads on a quad-core Intel Nehalem machine	68
Table 6.1 Correlation of selected performance counters with internal power consumption	75
Table 6.2 Three selected performance counter groups and their coefficients of determination of multivariate regression	75
Table 7.1 Parameters defined for PADD and its power and performance model	84
Table 7.2 Parameter values for the simulation	92

LIST OF FIGURES

Figure 1.1 Effect of DVFS in energy consumption	3
Figure 3.1 Micro-benchmarks showing time and energy performance of MPI calls with CPU scaling.	14
Figure 3.2 Micro-benchmarks showing time and energy performance of MPI calls with CPU scaling.	15
Figure 3.3 Cumulative distribution functions (CDF) of the duration of and interval between MPI calls for every MPI call for twelve benchmark programs we evaluate in this chapter.	16
Figure 3.4 This is an example trace of an MPI program. The line shows the type of code executed over time. There are 10 MPI calls, and calls 1–3 and 5–8 make up communication regions because they are close enough. Call 9 is considered a region because it is long enough. Even though they are MPI calls, calls 4 and 10 are not in a communication region because they are neither close enough nor long enough.	19
Figure 3.5 The mapping of OPS to P-state for a communication region.	21
Figure 3.6 State diagram for <i>composite</i> algorithm.	23
Figure 3.7 Overall EDP results for 4 methods relative to base case. S-3D and SMG are abbreviations for SWEEP3D and SMG2000, respectively. For readability, the y-axis starts at 0.70.	27
Figure 3.8 Comparison of single and multiple p-state transition. S-3D and SMG are abbreviations for SWEEP3D and SMG2000, respectively. For readability, the y-axis starts at 0.70.	31
Figure 3.9 These are p-state transition examples by different MPI call patterns. FN and FP mean false negative and false positive, respectively.	35
Figure 3.10 Breakdown of execution time for adaptive region-finding algorithms with $\tau = 20$ ms and $\lambda = 20$ ms.	36
Figure 3.11 Evaluating τ in <i>composite</i> . Show fraction of time mispredicted and number of regions (right-hand y-axis) as a function of τ . For readability, right-hand y-axis has different scale for CG and SP. (Continued in Figure 3.12.)	38

Figure 3.12 Evaluating τ in <i>composite</i> . Show fraction of time mispredicted and number of regions (right-hand y-axis) as a function of τ . For readability, right-hand y-axis has different scale for CG and SP. (Continuation of Figure 3.11.)	39
Figure 4.1 Execution time vs. energy consumption from single p-state execution	47
Figure 4.2 Execution time vs. energy consumption from single p-state execution	48
Figure 4.3 Execution time vs. energy consumption per region	50
Figure 4.4 Minimum energy over time	53
Figure 5.1 PowerMon prototype board	58
Figure 5.2 A reading from different sensors	59
Figure 5.3 Accuracy comparison of internal and external power meters.	61
Figure 5.4 PowerMon and <i>Power Egg</i> measurements with different active thread/core counts	63
Figure 5.5 PowerMon and <i>Power Egg</i> measurements running Iperf benchmark.	64
Figure 5.6 PowerMon and <i>Power Egg</i> measurements with different PCI cards installed.	65
Figure 5.7 PowerMon and <i>Power Egg</i> power measurements with/without a hard drive.	66
Figure 5.8 Power consumption over different monitoring intervals.	67
Figure 6.1 Distribution of prediction errors	76
Figure 6.2 Power Prediction Accuracy for NPB benchmark	77
Figure 6.3 Power Prediction Accuracy for SPECOMP benchmark	78
Figure 7.1 Node and Domain state diagrams. Each edge indicates the transition time delay.	83
Figure 7.2 Each node has its own Local Buffer. The Global Buffer is a pool of additional capacity spread across the active nodes.	87
Figure 7.3 One day access logs of the sporting event web site and synthetically generated CPU utilizations for a single domain.	90

Figure 7.4 Average power consumption and CPU utilizations for 11 different load levels. The power consumption is obtained from the SPECpower_ssj2008 benchmark. While running it, the CPU utilizations are measured in the same machine.	91
Figure 7.5 The overall comparison of the representative cases. (Util=14.5%, MW=60, AP=60, VS=1, MD=30)	93
Figure 7.6 The comparison of the adaptive GB/LB with different workload intensity. (MW=60, AP=60, VS=1, MD=30)	93
Figure 7.7 The comparison of adaptive buffering and fixed buffering. (Util=14.5%, MW=60, AP=60, VS=1, MD=30)	94
Figure 7.8 The comparison of domain migration and node transition cost. (Util=14.5%, MW=60, AP=60, VS=1, MD=30)	95
Figure 7.9 The comparison of the adaptive GB/LB with different number of domains. On X-axis, (x,y) indicates the number of domains and the maximum aggregate utilization, respectively. (MW=60, AP=60, VS=1, MD=30)	95

Chapter 1

Introduction

In the past decades, the “performance only” paradigm in high performance parallel and distributed computing systems tends to increase performance at all costs. The attention given to increasing computing performance has resulted in excessive power usage in two ways: drawing more power by CPU itself and dissipating more heat generated by CPU. As an example of the problem that is faced, several years ago it was observed that on their current trend, the power density of a microprocessor will reach that of a nuclear reactor by the year 2010 [2]. According to Eric Schmidt, CEO of Google, what matters most to Google is not speed but low power, because data centers can consume as much electricity as a small city. The 10 most powerful supercomputers on the TOP500 List (www.top500.org) each require up to 10 megawatts of peak power – enough to sustain a city of 40,000 [3]. As a result, power-aware computing has been recognized as one of critical research issues in parallel and distributed computing systems.

Our goal is a system that can understand power-performance tradeoffs and balance power consumption and performance. The system collects relevant data at runtime and evaluate the tradeoffs using empirical observations. Then, it dynamically and efficiently changes the execution environment of running an application in order to improve power and performance efficiency. This thesis shows our accomplishments to achieve our goal.

In this work, we develop and evaluate several systems that can understand different power and performance tradeoffs by characterizing application behaviors and take advantage of these characteristics. Our systems figure out the tradeoffs at runtime and make the best power saving decision on the fly among available power saving techniques. We show that the adaptive methods achieve improving power performance efficiency with minimal performance penalty.

1.1 Power and performance tradeoffs

Unfortunately, the “last drop” of performance tends to be the most expensive. One reason is power consumption, because dynamic power is proportional to the product of the frequency and the square of the voltage. A simple model of dynamic power consumption (P) of a CPU implemented in CMOS technology is approximated as follows:

$$P \approx A C V^2 f + L$$

where C is total capacitance, A is CPU activity, f is frequency, V^2 is the square of voltage, and L is leakage by static power. In recent computing systems, the static power becomes a significant component of total power dissipation [4]. It can be comparable to dynamic (switching) power in future technologies. Reducing the static power, however, is a hardware design issue.

Energy consumption (in Joules) is measured by integrating the production of instantaneous power ($P(t)$) by time;

$$E = \int_{t_1}^{t_2} P(t) dt$$

To dynamically balance the concerns of power and performance, new architectures have aggressive power controls. Because architectural advances for raw performance improvement are accompanied with more power usage, several power-aware computing techniques have been introduced to control both power and performance dynamically. For example, many microprocessors allow frequency and voltage scaling on the fly, which allows the application or operating system to provide *dynamic voltage and frequency scaling* (DVFS). DVFS is a popular and effective technique, performed in adjusting a power-performance tradeoff because CPU is the major power consumer in a single node, consuming 35-50% of the nodes’ total power [5]. We denote each possible combination of voltage and frequency a processor state, or *p-state*.¹ While changing p-states has broad utility, including extending battery life in small devices, the primary benefit of DVFS occurs when the p-state is reduced in code regions where the CPU is not on the critical path. In such a case, power consumption will be reduced with little or no decrease in end-user performance.

Figure 1.1 shows the energy-time (or, equivalently, power-performance) tradeoff graphically. The rectangular area in Figure 1.1(a) shows the energy consumption at the highest frequency.

¹The p-state is a processor performance state defined by AMD, which is a generic term defining possible combinations of voltage and frequency. The CPU scaling represents the transition of p-state. In this paper, lower p-state indicates higher CPU frequency, meaning faster speed.

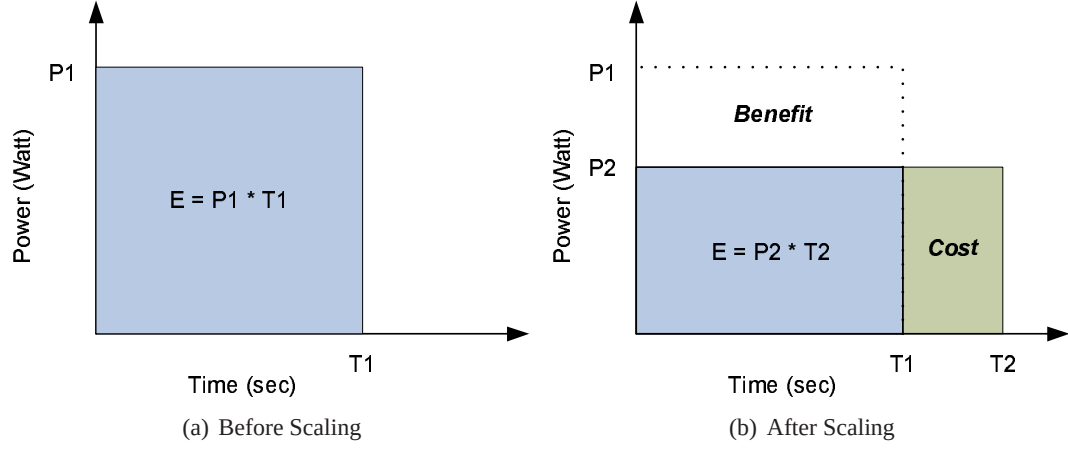


Figure 1.1: Effect of DVFS in energy consumption

On the other hand, Figure 1.1(b) shows the change of energy consumption to a lower frequency after p-state transition. Once we scale the CPU voltage and frequency, a cubic drop in power usage occurs. However, the performance decrease is at most linear with frequency reduction, which (usually) increases overall execution time. So, the overall energy savings can be realized only if cost is less than benefit. In high performance computing, Energy-Delay Product (EDP) is commonly used to evaluate the power-performance tradeoff. This metric is more biased toward execution time than the Energy metric. In this research, we use different metrics depending on the characteristics of applications.

1.2 Accomplishments

Our goal is to balance power savings and performance penalty in parallel or distributed applications by exploiting various power saving techniques. This dissertation consists of five related projects that we accomplished. Each project plays a vital role as a steppingstone to the goal. Chapter 3 describes a MPI runtime system that dynamically reduces CPU frequency and voltage during communication phases in MPI programs. It dynamically identifies such phases and, without profiling or training, selects the CPU frequency in order to balance power-performance tradeoffs (i.e., energy-delay product). All analysis and subsequent frequency and voltage scaling is performed via MPI profiling interface and so is entirely transparent to the application. This means that the large number of existing MPI programs, as well as new ones being developed, can use our system

without modification. Results show that the median reduction in energy-delay product for twelve benchmarks is 8%, the median energy reduction is 11%, and the median increase in execution time increase is only 2%.

In Chapter 4, we determine minimum energy consumption in voltage and frequency scaling systems for a given time delay. As we mentioned earlier, The benefit of DVFS varies by workload characteristics in code regions. Therefore, it is difficult to evaluate the effectiveness of a DVFS scaling algorithm. Our work establishes the optimal baseline of DVFS scheduling for any application. Given the baseline, one can better evaluate a specific DVFS technique. We assume we have a set of discrete places where scaling can occur. A brute-force solution is intractable even for a moderately sized set (although all programs presented can be solved with the brute-force approach). Our approach efficiently chooses the exact optimal schedule satisfying the given time constraint by estimation. We evaluate our time and energy estimates in NPB serial benchmark suite. The results show that the running time of the algorithm can be reduced significantly with our algorithm. Our time and energy estimations from the optimal schedule have high accuracy, within 1.48% of actual.

Chapter 5 describes an internal power meter that provides accurate, fine-grained measurements. The above projects use the entire system power consumption to quantify the effectiveness. However, the power consumed by each component within a system actually varies during program execution. External power measurements do not provide information about how the individual components utilize power. The fine-grained monitoring of the prototype is evaluated and compared with the accuracy and access latency of an external power meter. The results show that we can measure the power consumption more accurately and more frequently (about 50 measurements per second) with low power monitoring overhead (about 0.44 W). When combined with an external power meter, we can also derive the power supply efficiency and the hard disk power consumption.

Chapter 6 describe how to estimate the power consumption with performance data observed in a system. For this prediction model, we profile the power and performance data when running multi-threaded applications on multi-core architecture and then establish a power prediction model using a regression method. We evaluate and analyze the accuracy of this prediction model.

In Chapter 7, we develop a power-aware domain placement solution in virtualized system environments. This technique saves power by dynamically migrating virtual machines and packing them onto fewer physical machines when possible. We perform a simulation study on this technique to demonstrate the power and performance changes and evaluate that how much our approach

outperforms in terms of energy savings.

In Chapter 8, we conclude our research by summarizing our accomplishments. Also, we present future research directions in our research area. In addition to the previous accomplishments, we still strive for improving runtime adaptation of power-performance in parallel applications. The adaptation will be achieved through continuous power and performance monitoring by primarily exploiting DVFS. We also plan to explore various adaptation techniques, such as the number of active threads in OpenMP applications if useful.

1.3 Contributions

In this work, we make the following contributions. First, we develop a MPI runtime system that can dynamically balance power and performance tradeoffs in MPI applications by minimizing energy-delay product. Our system dynamically identifies power saving opportunities without profiling or training and then determines the best p-state to improve the power and performance efficiency. Our system is entirely transparent to MPI applications because it works within the MPI library.

Second, we develop a method for determining minimum energy consumption in voltage and frequency scaling systems for a given time delay. Our method efficiently chooses the exact optimal schedule satisfying a given time constraint by estimation. Our approach helps to better analyze the performance of a specific DVFS algorithm in terms of balancing power and performance.

Third, we develop a power estimation model that can correlate power and performance data on a chip multiprocessor machine. According to our evaluations, the prediction model is reasonably accurate to predict power consumption only with performance data in various execution configurations. While running an application, one can make a fast decision of power and performance adaptation by using performance data, because there is no delay for actual power measurements.

Last, we develop an algorithm to save power by dynamically migrating virtual machines and placing them onto fewer physical machines depending on their performance needs. Our scheme uses a two-level, adaptive buffering scheme which reserves processing capacity. It is designed to absorb workload surges to avoid significant performance violations, while reducing the amount of energy wasted on the buffers. We also develop a simulation framework for our study of the energy benefits and the performance effects of our algorithm along with studies of its sensitivity to various parameters.

Chapter 2

Background

Power and energy savings have been one of main issues in many computing research area. In this chapter, we describe the related work by making two parts in order to describe diverse ways of previous research. First, we summarize prior work for power savings in mobile / server systems. Then, the power savings research focusing on HPC systems are presented.

2.1 Energy savings in server systems

Originally, much work in energy savings has been aimed at the systems that operate on energy-constrained devices, such as mobile systems. The goal is to develop an efficient energy savings algorithm either through dynamic voltage and frequency scaling (DVFS) [6, 7, 8] or through disk / network control [9, 10] while, at the same time, satisfying system requirements. In this dissertation, we focus on energy savings in parallel and distributed server systems, instead of mobile systems. This research is complementary to ours and provides a worthwhile guidance.

Several projects have also examined energy savings in server systems. Energy savings has become one of the biggest concerns, especially those sites which have server farms, such as hosting centers managing sufficiently large numbers of machines. In these sites, energy management may become an issue; see [11, 12, 13] for examples of this using commercial workloads and web servers. Such work shows that power and energy management are critical for commercial workloads, especially web servers [14]. Additional approaches have been taken to include DVFS and request batching [15]. The work in [16] applies real-time techniques to web servers in order to conserve energy while maintaining quality of service. Recent work in commercial system area

evaluates the impacts of DVFS in terms of quality of service as well as power savings. One is to propose a DVFS algorithm meeting the end-to-end delay constraint in web servers [17]. The other [18] shows, given tolerable response time, the possible power savings in a Web server testbed. The paper [19] proposes a framework for improving the aggregate throughput by controlling the power allocation in multiple system under a certain power limitation. The papers [20, 21] show that efficient algorithms for server load balancing can be one promising solution. On the other hand, various methods have been proposed to achieve energy savings by applying power-scalable components to either CPU or disk, which are two major power consumers [22]. Most work focuses on leveraging power (energy) performance tradeoffs under given conditions such as maximum bound of power consumption and tolerable performance.

In server farms, disk energy consumption is also significant; several groups have studied reducing disk energy by reducing disk access speed or by varying its energy level (*e.g.*, [23, 24, 25]). The approach in the paper [26] infers disk access patterns of application with program counters, which can allow avoidance of unnecessary disk energy consumption. In our work, we do not consider disk energy savings as it is generally less than CPU energy, especially if scientific programs operate primarily in core.

Our work differs from prior research in that it focuses on HPC systems which execute scientific applications specifically, rather than mobile devices or commercial servers. Commercial workloads vary over time. Therefore, energy management is often performed by a conservative method which is used according to the service load. On the other hand, HPC applications often have regular patterns and they are more predictable than commercial workloads. Our work aims at efficiently applying DVFS with imposed time delay in a fine-grained way to maximize energy savings effect. Thus, the prior research in disk energy management is complementary to ours.

2.2 Energy savings in HPC systems

The most relevant related work to this dissertation is in high-performance, power-aware computing. For decades, performance improvement for both systems and applications has been the only major consideration in HPC research areas. Consequently, it resulted in the advent of large scale power consuming clusters, which causes substantial energy cost. Hence, much research for power awareness in HPC community has been introduced recently.

For power analysis, Feng *et al.* [27] performed a power profiling on a cluster system by

deploying digital multi meters. The profiling framework isolates 10 power points including CPU, memory, disk, and network and profiles power changes while running some benchmarks. Although its profiling approach is similar with ours, the approach is not flexible in that the monitoring framework is dedicated to its testing cluster machines. The paper [28] introduces a dynamic compilation technique of HPC applications to reduce power consumption. This work focuses only on I/O optimization in order to save both time and energy. Ge *et al.* [29] uses a variety of different DVFS scheduling strategies (for example, both with and without application-specific knowledge) to save energy without significantly increasing execution time. Follow-on work saved energy while limiting the execution time overhead to a user-specified amount [30]. A similar run-time effort is described by Hsu and Feng [31]. The prior work at NC State is fourfold: an evaluation-based study that focused on exploring the energy/time tradeoff in the NAS suite [32], development of an algorithm for switching p-states dynamically between phases [33], leveraging load imbalance by inter-node bottleneck to save energy [34], and minimizing execution time subject to a cluster energy constraint [35]. Another recent work [5] exploited inter-node slack time to reduce power consumption. The algorithm automatically identifies idle time at the end of each iteration and then steers the p-state for energy savings. This is an effective approach, especially in a load unbalanced job execution, such as N-Body simulation.

The above approaches strive to save energy for a broad class of scientific applications. Another approach is to save energy in an application-specific way; work by Chen *et al.* [36] used this approach dedicated to a parallel sparse matrix application, and Hsu and Kremer [37] leverage the memory bottleneck of HPC applications.

Recently, Curtis-Maury *et al.* [38] developed an adaptive concurrency control framework in OpenMP applications. This utilizes multiple thread-to-core mappings in multi-core architecture. This approach establishes a performance model for accurate power-performance prediction. Their following work [39] expands the approach by integrating DVFS. They achieved good power-performance adaptation on multi-core architecture. However, there exist several limitations. First, the approach only can be applied for OpenMP based applications. Second, the adaptation needs priori information that is used to train a system for accuracy. Different tuning is required on a different machine. Third, they do not provide in-depth analysis regarding their power-performance adaptation they achieved.

In addition to designing energy saving algorithms, determining the maximum bound of energy savings has been researched [40, 41]. Whichever metrics are used such as EDP, the achieve-

ment of energy savings itself does not explain the comparable value of the algorithms unless the possible savings at maximum is known. This research can establish the potential benefit of DVFS for a given application.

There are also a few high-performance computing clusters designed with energy in mind. One is BlueGene/L [42], which uses a “system on a chip” to reduce energy. While each node consumes little energy and has low performance, high performance is achieved by using up to 128,000 processors. An earlier energy-aware machine, Green Destiny [43], uses low-power Transmeta nodes.

Our work attempts to infer regions based on recognizing repeated execution. Others have had the same goal and carried it out using the program counter. Gniady and Hu used this technique to determine when to power down disks [44], along with buffer caching pattern classification [45] and kernel prefetching [46]. In addition, dynamic techniques have been used to find program phases [47, 48, 49], which is tangentially related to our work. We divide programs into one or more communication phases. There has been a large body of work in phase partitioning. Static techniques, such as [50, 51], appeared in the literature first.

2.3 Energy savings in server farms

Much research has been conducted on dynamic mapping of domains to physical machines. In some work, performance is the primary concern without power analysis. The work by Steinder, *et al.* [52], is both implementation and a simulation. In her work, there is a workload flow controller that manages the workloads being run and helps drive the application placement controller. In our work, there is no such workload controller: the workloads are run in the individual domains without any centralized control over them. The study by Bobroff, *et al.* [53], is a predictive placement scheme based on time series analysis. In contrast, our work is reactive, observing load and reacting appropriately.

The problem of saving power in a throughput-computing environment has also been studied previously. Most work centers on fluid workloads, i.e., workloads consisting of many short-lived independent requests that can be served by homogeneous software running on multiple physical machines and which have a centralized workload distributor that routes the requests to the machines which process them. Elnozahy, *et al.* [54] simulate the effects of dynamic voltage scaling coupled with vary-on/vary-off (“VOVO”, basically powering nodes on/off). They run their simulator against

two well-known production web traces and find that VOVO alone saves up to 42 % while the addition of voltage scaling saves an additional 18 %. Similarly, Chase, *et al.* [55] use an artificial resource economy to save energy by balancing the cost of resources against the benefit realized by employing them. This is also simulation-based work and focuses on satisfying the demand presented by web traffic traces. Both techniques save power by turning some nodes off and using a central workload distributor to route new requests to other nodes. The Power-Aware Request Distribution (PARD) system addresses data center power by using the individual request as the fundamental unit of work [56]. The system directs requests to a pool of compute resources which can be powered on or off depending on anticipated load. Because requests are independent and servers are stateless, powering down a server is simply a matter of removing it from the load balancer, waiting for pending requests to finish, and turning it off. In this environment, there is nothing to migrate or back up when shutting down a node, which leads to a very fine-grained power management: the system runs only the machines needed to satisfy the current demand. However, the PARD approach is only applicable to workloads similar to web-serving, and many workloads cannot be modeled in this manner. For example, processes may need to maintain state, or there may be multiple indivisible processes that each handle a separate incoming demand.

Recently, Nathuji and Schwan [57] proposed a framework to determine the power-performance tradeoffs of virtual machines and to efficiently manage physical machines within given power budgets. This work focuses on achieving good performance under a power cap, whereas our work seeks to satisfy an SLA while minimizing energy consumption. Verma, *et al.* [58, 59] designed and evaluated several domain placement algorithms on top of their existing platforms. Like our approach, they examined the tradeoff between power consumption and migration cost of running domains when determining their placement in virtualized systems. Although the problem they are addressing is similar, our contribution is distinguished in two ways. First, because our work is a simulation study, we can define and explore numerous configurable parameters that vary by hardware and software characteristics and evaluate their effects in theoretical environments, including very large scale computing systems. Second, we propose a two-level buffering scheme where the buffering sizes are adapted by given SLA requirements. Our algorithm finds power efficient placement of domains while minimizing the SLA violation.

There has been a significant amount of research into understanding and predicting workload patterns. Bradley, *et al.* [60] proposed two power management algorithms using short-term and long-term workload predictions. The algorithms identify temporal workload patterns and provide

enough available resources proactively by exploiting these patterns. More recently, Chen, *et al.* [61] studied incoming traffic patterns and used them to predict the amount of resources needed to process the load. Such prior work focuses on improving predictions of workloads and applying the information to power aware server provisioning. In contrast, PADD simply reacts to the work being processed or being generated within the computing environment. In addition, these studies assume that one server is dedicated to a service. Since the servers are not shared by different services, there is no domain migration between running servers. These approaches are complementary to our work because combining prediction with reaction to the workload can lead to better placement decisions, achieving more power savings with less performance impact.

In contrast to coarse-grained domain packing, CPU packing and node packing techniques [62, 63, 64] are available to the operating system as a fine-grained packing. This allows the packing of jobs onto a subset of the available processors and power down the rest.

There is a large body of work on Dynamic Voltage and Frequency Scaling (DVFS) to save energy [65, 66]. The work [67] proposes a online power management approach on a virtual machines supporting isolation between them and honoring the power management requests of the individual virtual machines. Using DVFS allows us to improve further the power efficiency of the running systems in the environment by reducing voltage and frequency when a server's processors are not a bottleneck. Since our work aims at domain distribution in virtualized server environments, any local DVFS algorithm in a system can be combined with our approach to get additional power savings.

Chapter 3

Adaptive, Transparent CPU Scaling Algorithms in MPI Communications

In this chapter, we present a transparent, adaptive system that reduces the p-state in communication phases. While changing p-states has broad utility, including extending battery life in small devices, the primary benefit for HPC occurs when the p-state is reduced in regions where the CPU is not on the critical path. In such a case, power consumption will be reduced with little or no reduction in end-user performance. Previously, p-state reduction has been studied in code regions where the bottleneck is in the memory system [68, 31, 29, 32, 33] or due to load imbalance between nodes [34]. In our approach, the communication phase is defined as a code region of applications calling Message Passing Interface (MPI).

Our system is built as two integrated components: training the system and making the actual shifting. We first designed several training algorithms that demarcate communication regions. In addition, we use a simple metric — operations per unit time — to determine the proper p-state for each region. To do so, it monitors MPI calls to determine communication regions on the fly. We then designed the shifting component, which includes the mechanics of reducing the p-state at the start of a region and increasing it at the end. Once regions are demarcated, our system uses performance counters to choose the appropriate p-state for each communication region; that is, at the beginning of the region it reduces the p-state to the region-specific target, and at the end of the region it increases the p-state back to its original setting.

The research in this chapter was funded in part by NSF grants CCF-0234285 and CCF-0429643, and an University Partnership award from IBM.

Because our system is built strictly with code executed within the PMPI runtime layer, there is no user involvement whatsoever. Thus, the large base of current MPI programs can utilize our technique with both no source code change and no recompilation of the MPI runtime library. While we aim our system at communication-intensive codes, the performance degradation for computation-intensive programs is negligible.

The difference between all the research presented in Section 2.2 and this chapter is the type of bottleneck we are attacking. Specifically, researchers have attacked both the memory bottleneck, where energy can be saved because the CPU is waiting for data from the memory system, and the node bottleneck, where energy can be saved because the CPU is waiting for data from another node. Here, we address the communication bottleneck, where energy can be saved because MPI communication operations are not CPU intensive. This is the first work we know of to address the communication bottleneck.

Results on the NAS benchmark suite and the ASCI Purple benchmark suite show that we achieve up to a 19.9% reduction in energy-delay product (EDP) compared to an energy-unaware scheme where nodes run as fast as possible. Furthermore, across the entire NAS suite, our algorithm that reduces the p-state in each communication region saved a median of 8% in EDP and 11% in energy. This was a significant improvement compared to simply reducing the p-state for each MPI communication call. Also, importantly, this reduction in EDP did not come at a large increase in execution time; the median increase across all benchmarks was 1.9%, and the worst case was only 5.8%.

The rest of the chapter is organized as follows. In Section 3.1, we provide motivation for reducing the p-state during communication regions. Then, Section 3.2 discusses our implementation, and Section 3.3 discusses the measured results on our power-scalable cluster. Finally, Section 3.4 summarizes and describes future work.

3.1 Motivation

To be effective, reducing the p-state of the CPU should result in a large energy savings but a small time delay. We implemented a microbenchmark that repeatedly calls the same MPI operation with a fixed message size. We executed this microbenchmark on 2 nodes for about 50 seconds using many common MPI operations using both small and large messages. Figures 3.1 and 3.2 show the execution time and the energy consumption of four MPI operations. The y -axis

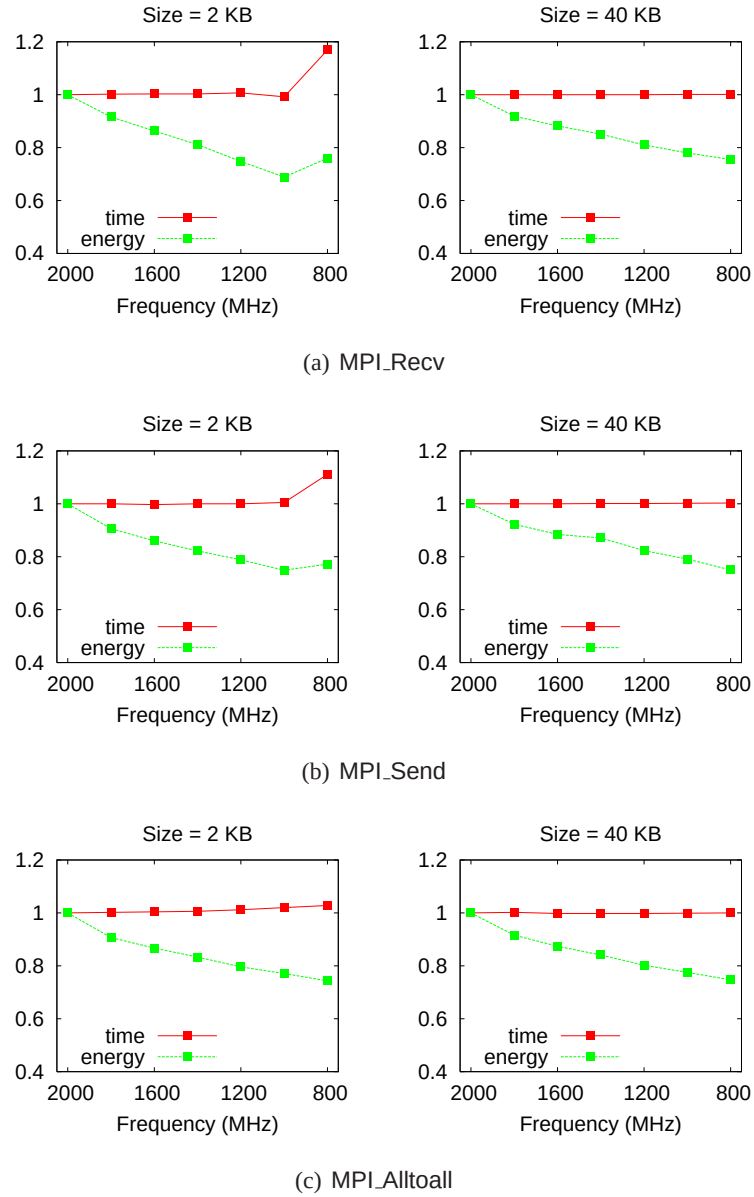


Figure 3.1: Micro-benchmarks showing time and energy performance of MPI calls with CPU scaling.

represents time and energy values relative to the fastest frequency (or p-state). For all the MPI operations (including those not shown), the p-state with frequency 1000 MHz provides an excellent tradeoff between time and energy: at least 20% energy is saved with a time increase of at most 2.6%. The greatest energy savings (31%) occurs when receiving 2 KB at 1000 MHz, with no time penalty.

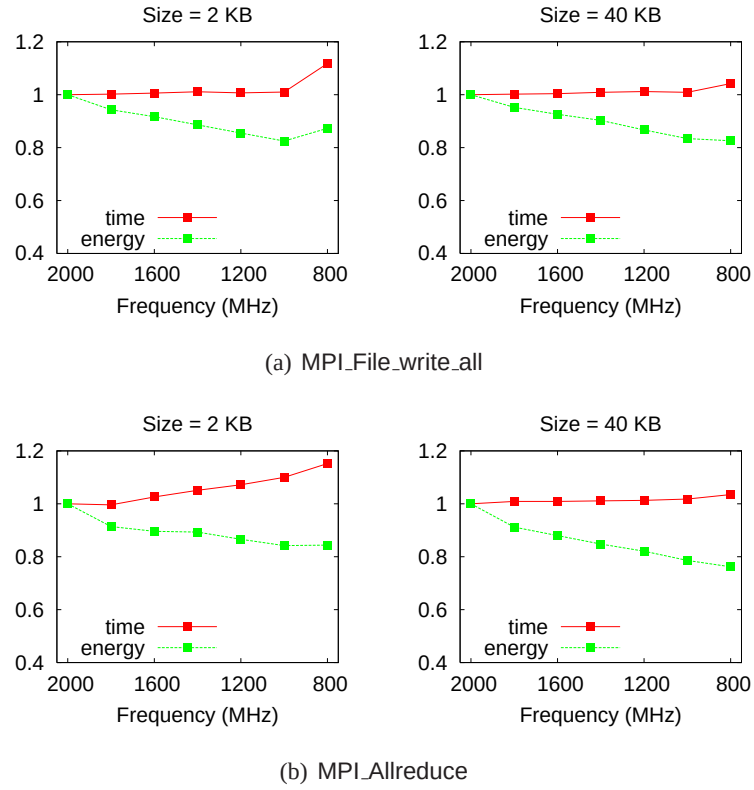


Figure 3.2: Micro-benchmarks showing time and energy performance of MPI calls with CPU scaling.

On the other hand, there often is a large time penalty using the slowest frequency (800 MHz) for sending small messages, up to 17%. For large messages, 800 MHz has a time increase as large as 5%. Overall, these data show that MPI calls provide an excellent opportunity to save energy with little or no time penalty.

However, in practice, an MPI operation may be too short to make reducing the p-state before and increasing the p-state after every call effective. The problem exists because p-state transitions are not instantaneous. Therefore, a short MPI operation does not have sufficient benefit to overcome the transition cost. Figure 3.3(a) shows a cumulative distribution function (CDF) of elapsed times of all MPI calls in all 12 benchmarks evaluated in this chapter (the entire NAS benchmark suite and four ASCI Purple benchmarks). These data are obtained from executions on 8 or 9 nodes on AMD Athlon-64 cluster connected by a 100Mbps network (see Section 3.3 for a complete description of the cluster). Every message from every node is included. (There is no normalization,

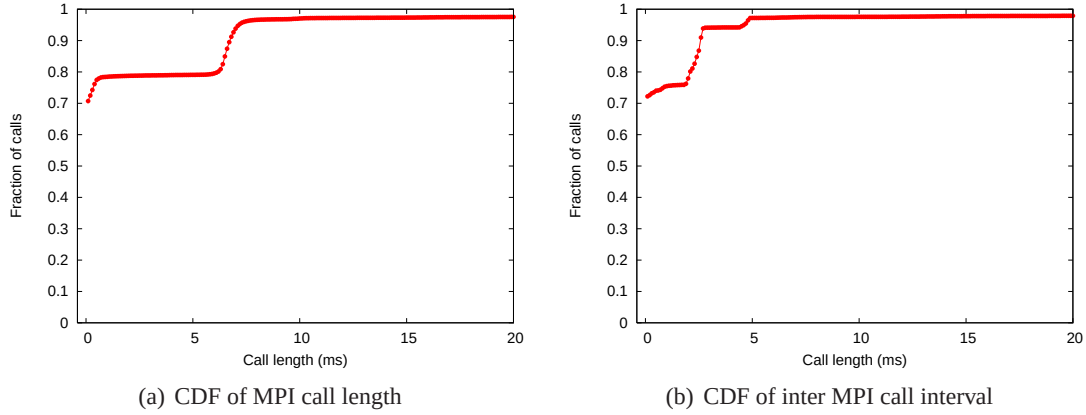


Figure 3.3: Cumulative distribution functions (CDF) of the duration of and interval between MPI calls for every MPI call for twelve benchmark programs we evaluate in this chapter.

so a communication-intensive benchmark has greater influence on this figure than a benchmark with few messages.) This figure shows that over 97% of MPI calls take less than 10 ms and 78% take less than 1 ms.

Considering that changing the p-state averages 1200 microseconds and can take up to 5600 microseconds (See Section 3.2.2 for details), time (and energy) increases when one reduces the p-state for each MPI operation and only for the duration of the operation. Figure 3.3(b) plots the CDF for the interval between MPI calls from the same experiments above. It shows that three-fourths of all intervals are less than 1 millisecond, indicating that MPI calls are clustered in time. Therefore, we can group “adjacent” MPI calls together as a region, avoiding frequent p-state transitions. Shifting only at region boundaries amortizes the overhead of p-state transitions, but executes the non-MPI code between calls in a reduced p-state, which may not be ideal.

The data presented in Figure 3.3 is an aggregate of all benchmarks. However, benchmarks have different communication patterns. Table 3.1(a) shows MPI call length per benchmark. (EP, which has only 5 MPI calls, is not shown.) The table shows the first, second, and third quartiles, sorted by the third quartile (column 3). The table shows that for 7 of 11 benchmarks, more than three-fourths of their MPI operations take less than 1 millisecond. Table 3.1(b) shows the detailed analysis of the inter-MPI intervals per benchmark. These data are again sorted by the third quartile; however, the order is different. It shows that for 5 benchmarks over three-fourths of the intervals take less than 1 millisecond. Also, the median interval is less than 1 millisecond for 8 benchmarks. The two benchmarks with the longest third-quartile inter-MPI interval, IS and FT, have very long

Table 3.1: Analysis of MPI call length and interval between MPI calls per benchmark. Benchmarks are sorted by 75th percentile in ascending order.

Benchmark	25%	50%	75%
Aztec	0.01	0.01	0.03
SMG2000	0.01	0.01	0.03
MG	0.04	0.07	0.11
LU	0.04	0.08	0.14
SPPM	0.01	0.06	0.39
CG	0.02	0.05	0.52
SWEEP3D	0.15	0.30	0.56
BT	6.52	6.71	6.97
SP	0.02	0.10	17.9
IS	0.47	41.8	10124
FT	1.47	1698	38541

(a) Analysis of MPI call length (ms).

Benchmark	25%	50%	75%
Aztec	0.01	0.02	0.03
SMG2000	0.01	0.02	0.03
BT	0.03	0.03	0.03
CG	0.01	0.03	0.48
MG	0.03	0.28	0.91
LU	0.07	2.37	2.66
SPPM	0.01	0.11	10.3
SWEEP3D	0.13	0.20	33.2
SP	0.03	6.21	34.2
IS	0.03	0.06	624
FT	2403	2937	3743

(b) Analysis of time interval (ms) between MPI calls.

MPI operations, so there is no need to amortize the shifting overhead over multiple operations. The tables foretell a problem in SWEEP3D, which has many short operations but a rather long third-quartile interval.

Because there is a cost to p-state shifting, it is necessary to avoid shifting too frequently. The data in these table show that the vast majority of MPI operations are too short to justify p-state shifting. However, the data also show that operations are clustered very close together. This presents an opportunity to be amortize the overhead of shifting across multiple MPI operations. Because it is unlikely that there is significant computation in the short code sections between MPI, the penalty for reducing the p-state in non-communication code is not expected to be great. Therefore, this project considers closely clustered MPI operation and the code between them to be communication regions, which are candidates for executing in a reduced p-state and saving energy.

Finally, our solution only considers shifting at MPI operations. However, there are other places shifting can be done. For example, one could shift at program (or computational) phase boundaries or even at arbitrary intervals. We rejected the latter because such shifting is not *context-sensitive*, meaning that we could not learn from previous iterations. The former is context-sensitive, and with elaborate phase analysis it will likely find excellent shifting locations. However, shifting only at MPI operations can easily be implemented transparently and in our tests was effective.

3.2 Design and implementation

The overall aim of our system is to save significant energy with at most incurring a small time delay. Broadly speaking, this system has three goals. First, it will identify program regions with a high concentration of MPI calls. Second, it determines the “best” (reduced) p-state to use during such MPI communication regions, called *reducible regions*. Both the first and second goals are accomplished with adaptive *training*. Third, it must satisfy the first two goals with *no involvement by the MPI programmer*, i.e., finding regions as well as determining and shifting p-states should be transparent. In addition to the training component, the system has a *shifting* component. This component will effect the p-state shifts at region boundaries.

While we discuss the training and shifting components separately, it is important to understand that our system does not transition between these components. Instead, program monitoring is continuous, and training information is constantly updated. Thus our system is always shifting using the most recent information. Keeping these components activated is necessary because we may encounter a region that is variable in terms of time length and MPI calls. So, our system can adaptively work for region finding.

To achieve transparency—i.e., to implement the training and shifting components without any user code modifications—we use our MPI-jack tool. This tool exploits PMPI [69], the profiling layer of MPI. MPI-jack transparently intercepts (hijacks) any MPI call. A user can execute arbitrary code before and/or after an intercepted call using *pre* and *post* hooks. Our implementation uses the same pre and post hook for all MPI calls.

An MPI routine (such as `MPI_Send`) can occur in many different contexts. As a result, the routine itself is not a sufficient identifier. But it is easy to identify the dynamic parent of a call by examining the return program counter in the current activation record. However, because an MPI call can be called from a general utility or wrapper procedure, one dynamic ancestor is not sufficient to distinguish the start or end of a region. Fortunately, it is simple and inexpensive to examine all dynamic parents. We do this by hashing together all the return PCs from all activation records on the stack. Thus, a call is uniquely identified by a hash of all its dynamic ancestors.

3.2.1 The training component

The training component has two parts. The first is distinguishing regions. The second is determining the proper p-state for each region. Starting from an MPI program with C calls of MPI

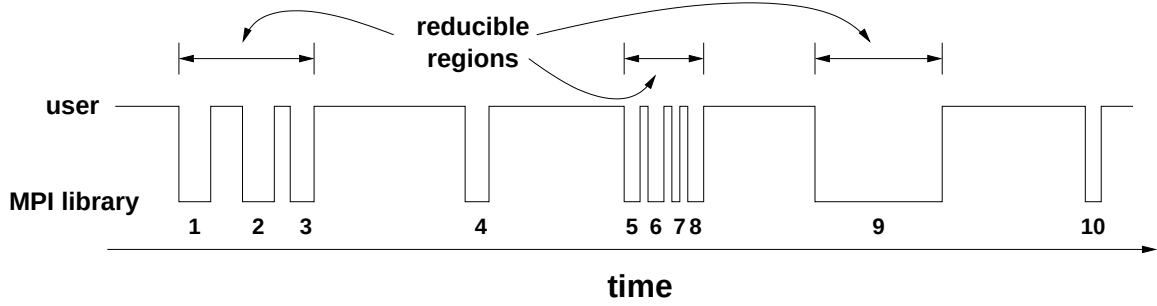


Figure 3.4: This is an example trace of an MPI program. The line shows the type of code executed over time. There are 10 MPI calls, and calls 1–3 and 5–8 make up communication regions because they are close enough. Call 9 is considered a region because it is long enough. Even though they are MPI calls, calls 4 and 10 are not in a communication region because they are neither close enough nor long enough.

routines, the goal is to group the C calls into $R \leq C$ regions based on the distance in time between adjacent calls, the time for a call itself, as well as the particular pattern of calls.

Figure 3.4 presents an example trace of an MPI program, where we focus solely on whether the program is in user code or MPI library code. The program begins in user code and invokes 10 MPI calls. In Figure 3.4 there are two composite reducible regions—(1, 2, 3) and (5, 6, 7, 8). It also has a region consisting of only call 9 that is by itself long enough to be executed in a reduce p-state. Thus, Figure 3.4 has three reducible regions, as well as two single MPI calls that will never execute in a reduce p-state. The MPI calls 4 and 10 are not long enough or close enough to be in a communication region. We leave these MPI calls without changing p-state.

First, we must have some notion that MPI calls are close together in time. We denote τ as the value that determines whether two adjacent MPI calls are “close”—anything less than τ indicates they are. Clearly, if τ is ∞ , then all calls are deemed to be close together, whereas if it is zero, none are. Next, it is possible that an MPI call itself takes a significant amount of time, such as an `MPI_Alltoall` using a large array or blocking `MPI_Send/MPI_Receive`. Therefore, we denote λ as the value that determines whether an MPI call is “long enough”—any single MPI call that executes longer than λ warrants reducing. Our tests use $\tau = 20$ ms and $\lambda = 20$ ms. Section 3.3.2 discusses the rationale for these thresholds.

Finally, we must choose a training paradigm; we consider three. The first is the degenerate case, *none*, which has no training. It always executes in the top p-state. The next paradigm, *static*, uses only knowledge known prior to execution. This knowledge can be extracted from com-

piller analysis or execution traces of past runs. The third paradigm is *adaptive*, which is what we focus on in this chapter. In this case, the system has no specific knowledge about the particular program, so knowledge is learned and updated as the program executes. For *adaptive*, there are many ways to learn. Next, we discuss two baseline region-finding algorithms, followed by two adaptive region-finding algorithms. Following that, we combine the best region-finding algorithm with automatic detection of the reduced p-state—this serves as the overall training algorithm that we have developed.

First, there is the *base* region-finding algorithm, in which there are no regions—the program always executes in the top p-state. It provides a baseline for comparison. Next, we denote *by-call* as the region-finding algorithm that treats every MPI call as its own region. It can be modeled as $\tau = 0$ and $\lambda = 0$: no calls are close enough and all are long enough. Section 3.1 explains that *by-call* is not a good *general* solution because the short median length of MPI calls can cause the shifting overhead to dominate. However, in some applications the *by-call* algorithm is very effective, saving energy without having to do any dynamic training.

We consider two adaptive region-finding algorithms. The first, called *simple*, predicts each call’s behavior based on what it did the last time it was encountered. For example, if a call ended a region last time, it is predicted to end a region the next time it appears. The *simple* method is effective for some benchmark programs. For example consider the pattern below.

... AB ... AB ... AB ...

Each letter indicates a particular MPI call invoked from the same call site in the user program; time proceeds from left to right. Variable τ is shown visually through the use of ellipses, which indicates that the calls are not close. The pattern shows that the distances between A and B are deemed close enough, but the gaps from B to A are not. Thus, the reducible regions in a program with the above pattern always begin with A and end after B. Because every A begins a region and every B ends a region, once trained, the *simple* mechanism accurately predicts the reducible region every time thereafter.

However, there exist patterns for which *simple* is quite poor. Consider the following pattern.

... AA ... AA ... AA ...

The gap between MPI calls alternates between less than and greater than τ . However, in this case both gaps are associated with the same call. Because of this alternation, the *simple* algorithm always

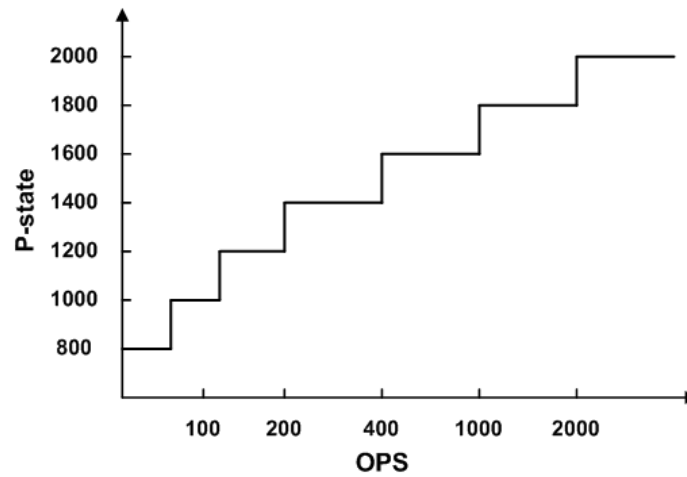


Figure 3.5: The mapping of OPS to P-state for a communication region.

mispredicts whether *A* is in a reducible region—that is, in each group *simple* predicts the first *A* will terminate a region because the second *A* in the previous group did. The reverse happens when the second *A* is encountered. This is not a question of insufficient training; *simple* is not capable of distinguishing between the two positions that *A* can occupy.

Our second region-finding algorithm, *composite*, addresses this problem. At the cost of a slightly longer training phase, *composite* collates information on a *per-region* basis. It then shifts the p-state based on what it learned the last time this region was encountered. It differs from *simple* in that it associates information with regions, not individual calls. The *composite* algorithm matches the pattern of these calls and saves it; when encountered again, it will be able to determine that a region is terminated by the same final call.

The second part of training is determining the p-state in which to execute each region. As mentioned previously, extensive testing indicates that the MPI calls themselves can execute in very low p-states (1000 or 800 MHz) with almost no time penalty. However, a reducible region also contains user code between MPI calls. Consequently, we found that the “best” p-state is application dependent.

The overall adaptive training algorithms we advocate are *simple* and *composite* using performance counters to dynamically determine the best p-state on a per-region basis. Specifically, in order to judge the dependence of the application on the CPU in these reducible re-

regions, *automatic* measures the micro-operations retired during the region. It uses the rate micro-operations/microsecond (or OPS) as an indicator of CPU load. Thus a region with a low OPS value is shifted into a low p-state. Our system continually updates the OPS for each region via hardware performance counters. (This means the p-state selection for a region can change over time.) Using large amounts of empirical test data, we developed by hand a single table that maps OPS to the desired p-state for a reducible region. The table is shown graphically in Figure 3.5. The mapping is used for all benchmark programs, so it is not optimized for any particular benchmark.

Our algorithm considers any MPI calls when training a communication region. The MPI library has asynchronous MPI calls, such as `MPI_Isend` and `MPI_Irecv`. These asynchronous calls usually takes short time and they are often overlapped with other computation, causing high OPS value. Therefore, it is less likely that these calls make a communication region or the p-state is significantly reduced in such a region.

3.2.2 The shifting component

There are two parts to the shifting component. The first is determining when a program enters and leaves a reducible region. The second part is effecting a p-state shift. The *simple* region-finding algorithm maintains *begin* and *end* flags for each MPI call (or hash). The pre hook of all MPI calls checks the *begin* flag. If set, this call begins a region, so the p-state is reduced. Similarly, the post hook checks the *end* flag and conditionally resets the top p-state. This flag is updated every time the call is executed: it is set if the region was close enough or long enough and unset otherwise.

Figure 3.6 shows a state diagram for the *composite* algorithm. There are three states: *OUT*, *IN*, and *RECORDING*. The processor executes in a reduced p-state only in the *IN* state; it always executes in the top p-state in *OUT* and *RECORDING*. The initial state is *OUT*, meaning the program is not in a reducible region. At the beginning of a reducible region, the system enters the *IN* state and shifts to a reduced p-state. The other state, *RECORDING*, is the training state. In this state, the system records a new region pattern.

The system transitions from *OUT* to *IN* when it encounters a call that was previously identified as beginning a reducible region. If the call does not begin such a region but it was close enough to the previous call, the system transitions from *OUT* to *RECORDING*. It begins recording the region from the previous call. It continues recording while calls are close enough.

The AMD executes a RISC engine internally. It translates x86 (CISC) instructions into RISC micro-operations. Therefore, micro-operations are a better indicator of CPU performance than instructions.

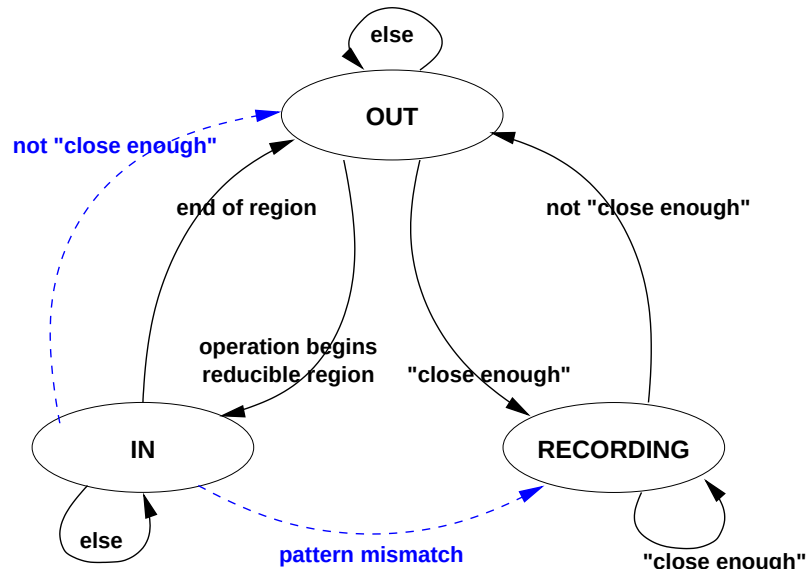


Figure 3.6: State diagram for *composite* algorithm.

The system ordinarily transitions from *IN* to *OUT* when it encounters the last call in the region. However, there are two exceptional cases—shown with dashed lines in Figure 3.6. If the current call is no longer close enough, then the region is truncated and the state transitions to *OUT*. If the pattern has changed—this current call was not the expected call—the system transitions to *RECORDING*. A new region is created by appending the current call to the prefix of the current region that has been executed. None of the applications examined in Section 3.3 caused these exceptional transitions.

Our implementation labels each region with the hash of the first MPI call in the region. When in the *OUT* state at the beginning of each MPI call, we look for a region with this hash as a label. If the region exists and is marked reducible, then we reduce the p-state and enter the *IN* state.

There are limitations to this approach if a single MPI call begins more than one region. The first problem occurs when some of these regions are marked reducible and some are marked not reducible. Our implementation cannot disambiguate between these regions. The second problem occurs when one reducible region is a prefix of another. Because we do not know which region we are in (long or short), we cannot know whether to end the region after the prefix or not. A more sophisticated algorithm can be implemented that addresses both of these limitations; however, this was not done because it was not necessary for the applications examined. Rather, our system elects

Table 3.2: P-state transition cost (microseconds) for a AMD Athlon-64 3000+ Processor [1]. The frequency in a row indicates a starting p-state and the frequency in a column is a target p-state.

	2.0Ghz	1.8Ghz	1.6Ghz	1.4Ghz	1.2Ghz	1.0Ghz	0.8Ghz
2.0Ghz	-	355.55	554.66	850.63	1138.78	1954.74	568.62
1.8Ghz	785.46	-	278.68	556.72	853.07	1522.51	292.44
1.6Ghz	1436.75	566.70	-	279.47	555.84	1122.03	289.90
1.4Ghz	2204.94	1218.49	279.32	-	279.61	746.02	289.22
1.2Ghz	3415.02	2259.24	556.16	280.20	-	362.66	575.44
1.0Ghz	5015.28	3597.30	853.51	564.01	280.72	-	297.68
0.8Ghz	5647.90	3892.10	289.13	289.70	1134.16	2141.23	-

to be conservative in time, so our implementation executes in top p-state during all ambiguities. However, this was not evaluated because no benchmark has such ambiguities.

The second part of the execution phase is changing the p-state. In our system using AMD microprocessor, the p-state is defined by a frequency identifier (FID) and voltage identifier (VID) pair. The p-state is changed by storing the appropriate FID-VID to the *model specific register* (MSR) in three phases. During phase 1 the processor driver performs a series of VID-only transitions each changing the voltage by the *maximum voltage step* (MVS). During phase 2 the processor core frequency is changed to the FID associated with the OS-requested P-state. During phase 3 the processor voltage is transitioned to the VID associated with the OS-requested P-state. Such a store is privileged, so it must be performed by the kernel. The *cpufreq* and *powernow* modules provide the basic support for this. Additional tool support and significant “tweaking” of the FID-VID pairs were needed to refine the implementation. The overhead of changing the p-state is dominated by the time to scale—not the overhead of the system call.

For the processor used in these tests, the upper bound on a p-state transition (including system call overhead) is about 5600 microseconds (from lowest frequency to highest frequency), but the average is less than 1200 microseconds, and the median is about 570 microseconds (Table 3.2).

3.3 Results

This section presents our results in two parts. The first part discusses results on several benchmark programs on a power-scalable cluster. These results were obtained by applying the different algorithms described in the previous section. The second part gives a detailed analysis of several aspects of our system and particular applications. It explains how the benchmarks behave

Table 3.3: Benchmark attributes. Region information from *composite* with $\tau = 20$ ms and $\lambda = 20$ ms.

	MPI calls	Reducible regions	Calls per region	Average time (ms)		Time fraction	
				Per MPI	Per region	MPI	region
EP	5	1	4	68.7	337.0	0.005	0.005
SWEEP3D	4,072	65	3.2	2.82	144	0.141	0.116
FT	46	42	1.0	18,399	20,151	0.859	0.859
IS	37	11	3.0	3,117	10,487	0.871	0.871
BT	108,706	475	226.1	8.51	1,951	0.886	0.887
Aztec	20,763	279	74.3	2.05	156	0.822	0.842
SMG2000	232,202	4	58050.2	0.08	8,171	0.581	0.987
CG	41,954	1976	21.2	6.56	143	0.727	0.751
MG	10,001	134	74.5	4.59	391	0.539	0.615
SP	19,671	4013	3.7	20.7	112	0.442	0.486
LU	162,193	766	211.6	0.92	709	0.228	0.827
SPPM	2,110	199	5.3	43.1	465	0.196	0.200

and how our technique works.

Benchmark attributes Our twelve benchmark applications come from the NAS parallel benchmark [70] suite and the ASCI Purple benchmark suite [71]. The NAS suite consists of scientific benchmarks including application areas such as sorting, spectral transforms, and fluid dynamics. We test class C of these benchmarks. The benchmarks are unmodified with the exception of BT, which took far longer than all the other benchmarks; to reduce the time of BT without affecting the results, we executed it for 60 iterations instead of the original 200. Four more benchmarks, *Aztec*, *SPPM*, *SMG2000*, and *SWEEP3D*, are from the ASCI Purple benchmark suite.

Table 3.3 presents overall statistics of the benchmarks running on 8 or 9 nodes (9 for BT and SP), which are distilled from traces that were collected using MPI-jack. The second column shows the number of MPI calls invoked dynamically. The number of reducible regions (column 3) is determined by executing the *composite* algorithm on the traces with both the close enough threshold (τ) and the long enough threshold (λ) set to 20 ms. The table shows the average number of MPI calls per region in the fourth column. Because some MPI calls are not in communication regions, the product of the third and fourth columns is not necessarily equal to the second column. Next, the table shows the average time per MPI call and per communication region. The seventh and eighth columns show the fraction of the overall time spent in MPI calls and regions, respectively.

According to our analysis, the applications have diverse characteristics. In particular, in

terms of the key region parameters—the number of regions, the number of calls per region, and the duration of the regions—there are at least two orders of magnitude difference between the greatest and least values. Noticeably, a lot of MPI calls are grouped to a communication region. As a result, the region length becomes at least 5 times longer, compared with MPI length. For instance, communication regions have on average 211 MPI calls in the LU benchmark. More than 80% of the execution time consists of the communication regions. Finally, with the exception of EP, the communication fraction is broadly distributed between 11.6% and 98.7%.

As we will show below, while several of the particular algorithms we developed may be best for *some* benchmark, the wide variety of characteristics in the entire suite means that they may be quite poor on other benchmarks. Therefore, our goal was to find an algorithm that works *well* for *all* benchmarks. Below we will show that *composite* is such an algorithm.

Experimental methodology For all experiments, we used a 10-node AMD Athlon-64 cluster connected by a 100 Mbps network. Each node has 1 GB of main memory. The Athlon-64 CPU supports 7 p-states (from 2000 to 800 MHz). Each node runs the Fedora Core 3 OS and Linux kernel 2.6.8, and frequency shifting was done through the *sysfs* interface. All applications were compiled with either *gcc* or the Intel Fortran compiler, using the O2 optimization flag. We controlled the entire cluster, so all experiments were run when the only other processes on the machines were daemons.

Elapsed time measurements are from the system clock using the *gettimeofday* system call. A power meter sits in between the system power supply of each node and the wall. We read this meter to get the power consumption. These readings are integrated over time to yield energy consumption. Thus the energy consumption measurements are empirically gathered for the entire system of 8 or 9 nodes.

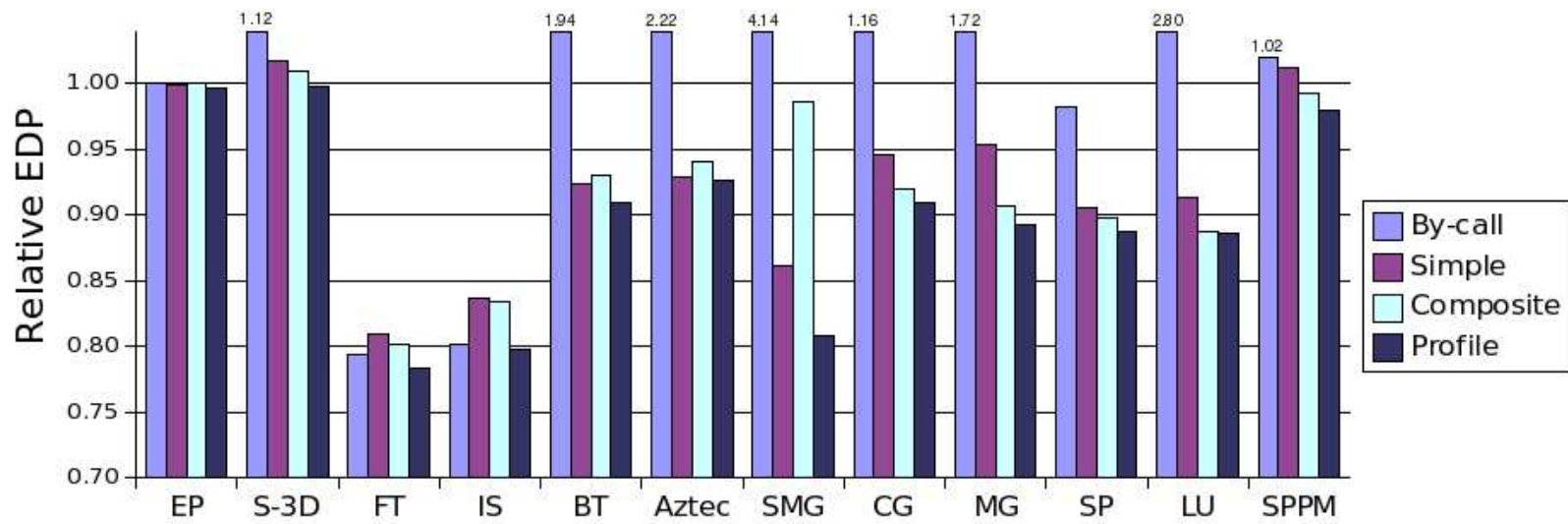


Figure 3.7: Overall EDP results for 4 methods relative to base case. S-3D and SMG are abbreviations for SWEEP3D and SMG2000, respectively. For readability, the y-axis starts at 0.70.

3.3.1 Overall results

Overall performance comparison Our work trades time for energy. This is a 2-dimensional tradeoff. The importance of each dimension will vary between users. Energy-delay product (EDP) is one canonical evaluation metric that essentially gives equal weight to the dimensions.

$$EDP = Energy \times Time = Power \times Time^2$$

It converts the two-dimensional energy-time tradeoff into a single scalar metric. We conducted five tests for each of the benchmarks. For each test, the measured elapsed time and consumed energy are presented. EDP is computed from these empirical measurements. The value relative to the base case is also presented. All tests were conducted a minimum of five times, with the median value reported. The variance between any two runs was very small, so it is not reported.

We tested three algorithms: *by-call*, *simple*, and *composite*. Additionally, we present two other algorithms for reference. One uses no power management, establishing the *base* case. The second uses *profile* data, representing the “best one can do.” Figure 3.7 shows the EDP of each method relative to the base case, which executes the program in the top frequency (2000 MHz) at all times. Linux puts the processor into a low-power halt state when the run queue is empty, so we note that the base test is already power-efficient to some degree. Table 3.4 on page 29 presents complete data from these tests.

The *composite* test shows the results of the algorithm that uses *composite* region finding and selects the reduced p-state for each region using OPS. The last column in Table 3.4 shows the results of *profile*; it executes a schedule of p-state changes that were produced from profiling traces. Essentially, *profile* is *composite* without mispredictions. The first algorithm (*by-call*) use a single reduced p-state throughout, while the last three may use several reduced p-states depending on the OPS.

The rightmost bar in Figure 3.7 (and rightmost column in Table 3.4) shows the performance of *profile*. In all cases, *profile* has the best EDP; this is expected because it uses prior application knowledge (a separate profiling execution). The usefulness of *profile* is that it serves as a rough lower bound on EDP. It is not a precise lower bound because of several reasons, including that the reduced p-state in a given region cannot be proven to be best for EDP.

The applications fall into several categories based on the performance of the algorithms. The first is no benefit. EP executes less than 0.5% of the total execution time in MPI calls, so no

Table 3.4: Overall performance of benchmark programs. Where appropriate, the value relative to the base case is also reported.

		Base	By-call			Simple		Composite		Profile	
EP	Time (s)	75.53	75.55	1.000	75.52	1.000	75.45	0.999	75.49	0.999	
	Energy (KJ)	59.876	59.878	1.000	59.857	1.000	59.946	1.001	59.669	0.997	
	EDP (MJ·s)	4.522	4.524	1.000	4.520	1.000	4.523	1.000	4.504	0.996	
SWEEP3D	Time (s)	81.21	91.78	1.130	81.81	1.007	81.65	1.005	81.12	0.999	
	Energy (KJ)	69.136	68.638	0.993	69.842	1.010	69.445	1.004	69.043	0.999	
	EDP (MJ·s)	5.615	6.300	1.122	5.714	1.018	5.670	1.010	5.601	0.998	
FT	Time (s)	984.40	987.25	1.003	985.77	1.001	984.23	1.000	984.64	1.000	
	Energy (KJ)	654.804	518.196	0.791	529.235	0.808	524.720	0.801	513.028	0.783	
	EDP (MJ·s)	644.588	511.588	0.794	521.704	0.809	516.445	0.801	505.147	0.784	
IS	Time (s)	128.16	129.02	1.007	128.55	1.003	128.32	1.001	128.78	1.005	
	Energy (KJ)	82.708	65.852	0.796	69.020	0.835	68.940	0.834	65.616	0.793	
	EDP (MJ·s)	10.600	8.496	0.802	8.872	0.837	8.847	0.835	8.450	0.797	
BT	Time (s)	929.97	1361.36	1.464	963.83	1.036	949.95	1.021	945.11	1.016	
	Energy (KJ)	610.476	807.352	1.322	543.956	0.891	555.549	0.910	546.529	0.895	
	EDP (MJ·s)	567.721	1099.096	1.936	524.282	0.923	527.743	0.930	516.529	0.910	
Aztec	Time (s)	51.59	83.85	1.625	54.41	1.055	53.73	1.042	53.69	1.041	
	Energy (KJ)	33.238	45.323	1.364	29.291	0.881	29.989	0.902	29.598	0.890	
	EDP (MJ·s)	1.715	3.800	2.216	1.594	0.929	1.613	0.941	1.589	0.927	
SMG2000	Time (s)	27.19	56.28	2.070	28.78	1.058	27.32	1.005	27.37	1.006	
	Energy (KJ)	19.488	39.024	2.002	15.864	0.814	19.132	0.982	15.642	0.803	
	EDP (MJ·s)	0.530	2.196	4.145	0.457	0.862	0.523	0.986	0.428	0.808	
CG	Time (s)	373.31	442.55	1.185	408.13	1.093	394.97	1.058	395.87	1.060	
	Energy (KJ)	254.364	249.464	0.981	220.164	0.866	221.020	0.869	218.052	0.857	
	EDP (MJ·s)	94.958	110.399	1.163	89.855	0.946	87.297	0.919	86.318	0.909	
MG	Time (s)	78.08	114.16	1.462	82.42	1.056	79.63	1.020	79.05	1.012	
	Energy (KJ)	59.208	69.632	1.176	53.501	0.904	52.621	0.889	52.208	0.882	
	EDP (MJ·s)	4.623	7.949	1.720	4.410	0.954	4.191	0.907	4.127	0.893	
SP	Time (s)	910.51	965.14	1.060	945.63	1.039	933.10	1.025	936.50	1.029	
	Energy (KJ)	778.150	721.161	0.927	679.124	0.873	681.844	0.876	671.022	0.862	
	EDP (MJ·s)	708.515	696.023	0.982	642.201	0.906	636.226	0.898	628.415	0.887	
LU	Time (s)	627.79	1225.83	1.953	648.98	1.034	641.43	1.022	642.98	1.024	
	Energy (KJ)	520.428	746.972	1.435	460.134	0.884	451.973	0.868	450.304	0.865	
	EDP (MJ·s)	326.721	915.663	2.803	298.619	0.914	289.936	0.887	289.530	0.886	
SPPM	Time (s)	460.33	475.27	1.032	475.53	1.033	468.09	1.017	463.92	1.008	
	Energy (KJ)	374.392	369.988	0.988	367.020	0.980	365.563	0.976	363.820	0.972	
	EDP (MJ·s)	172.344	175.844	1.020	174.529	1.013	171.115	0.993	168.785	0.979	

energy savings is possible with our mechanisms. Thus we do not evaluate EP any further. Communication accounts for 14% of SWEEP3D overall execution time, which is one-sixth that other benchmarks. This limits the maximum potential savings, for example even using profiling data the EDP is reduced by only 0.1%.

The second category contains FT and IS—applications that perform well with the trivial *by-call* method. These are the only two applications that have singleton regions—an individual MPI call that is longer than λ , thus allowing a reduction in the p-state. Moreover, in FT, no calls are within τ . The average length of MPI calls is 18.4 and 3.1 seconds for FT and IS, respectively. Consequently, the reducible regions are almost entirely MPI code and the reduced p-state is 800 MHz — the lowest p-state. As a result, these two benchmarks have the greatest energy saving. Although these programs are in the reduced p-state more than 85% of the time, the increase in execution time is less than 1%. Therefore, the EDP is very low. This is our best case in terms of EDP, but most applications do not fall into this category. In all the other benchmarks, the *by-call* shows the worst EDP values among the proposed algorithms.

The next class of programs are ones for which *by-call* performs poorly due to excessive p-state switching, but *simple* is better than *composite*. In our benchmark suite, this includes BT, Aztec, and SMG2000. The pattern of MPI calls in BT is such that *simple* predicts well so there are few false positives for *composite* to remove. Moreover, the longer training required by *composite* creates false negatives—which are lost opportunities to save energy—that do not occur in *simple*. Therefore, *composite* actually spends more time in the wrong p-state than *simple*. As a result, *simple* is 0.7% lower in EDP than *composite*; this is because *simple* results in consumption of 1.9% less energy but 1.5% more time. SMG is the only application where *composite* is significantly worse ($> 2\%$) than *simple* in EDP, which occurs because there are many false negatives due to training.

The fourth category, which consists of five programs, are those that both *by-call* and *simple* do not perform well. CG, MG, SP, LU, and SPPM belong to this category. In these benchmarks, the EDP of *composite* is lower than *simple* because of improved region finding. The *composite* algorithm eliminates both excessive p-state switching of *by-call* as well as a non-trivial number of false predictions of *simple*. We will show the detailed analysis of prediction accuracy in the next section.

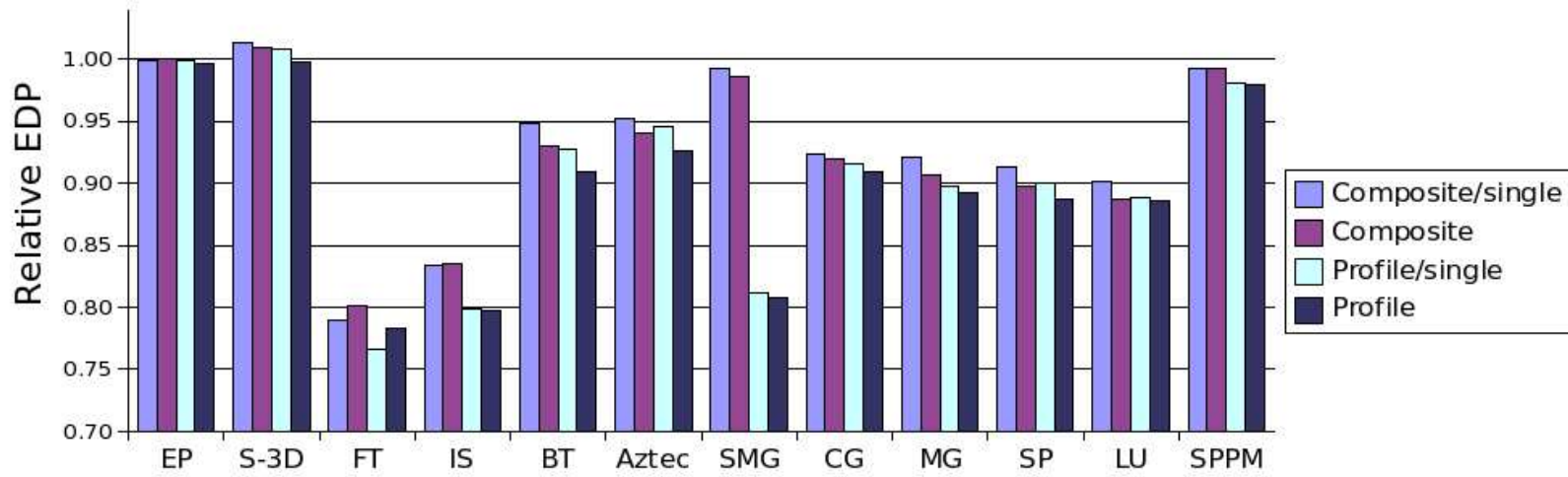


Figure 3.8: Comparison of single and multiple p-state transition. S-3D and SMG are abbreviations for SWEEP3D and SMG2000, respectively. For readability, the y-axis starts at 0.70.

Justification of automatic multiple p-state selection Our algorithms determine the reduced p-state dynamically for each region. This section evaluates the benefits of using multiple reduced p-states. We compare the online, multiple p-state performance to the best single-state performance. The best single p-state performance was determined by exhaustive search of all p-states. Figure 3.8 presents results in terms of EDP. There are four tests for each benchmark: *composite* and *profile* are the same as previous. The two additional tests are annotated with */single*. Table 3.5 on page 33 present the complete data from which the data in the figure is extracted.

The automatic solution selects the p-state for a region based on the measured performance (OPS) of the region during its previous execution. When the preferred p-state changes over time, the automatic will adjust, but always after the change has occurred. Therefore, if the preferred p-state changed every iteration, automatic would never execute a region in its preferred p-state. In which case, a single p-state test may perform better than automatic. This is the case in FT where the EDP of single is 1.5%. The preferred p-state changes in FT and there are very few iterations, so automatic is unable to amortize the overhead of learning. However, as the single-state method is right more often than automatic, it has lower EDP. In IS the single p-state EDP is 0.1% lower and in SPPM it is the same. In all other benchmarks, automatic is better. The automatic achieved its savings with no prior knowledge, whereas the single tests presented are the best overall—requiring as many as seven tests runs to determine.

In summary, *simple* is generally better than *by-call* due to less p-state switching overhead. However, *composite* is generally better than *simple* because it has, for several benchmarks, far fewer false positives at the cost of a small number of additional false negatives. There are three programs for which *simple* is better than *composite*. In terms of EDP, BT and Aztec are 0.7% and 1.2% better, but SMG2000 is 10.9%. In all three cases, *composite* suffers from false negative due to a long training period. The next section explains this in detail. Furthermore, automatic p-state selection is very effective.

3.3.2 Detailed analysis

This section focuses on the details of how we achieved the performance reported in the previous section. In particular, it explains overhead and then discusses false negatives and positives. Finally, it investigates how we selected our threshold values.

Table 3.5: Performance comparison between single p-state and automatic multiple p-state selection.

		Base	Composite/single		Composite		Profile/single		Profile	
EP	Time (s)	75.53	75.47	0.999	75.45	0.999	75.51	1.000	75.49	0.999
	Energy (KJ)	59.876	59.878	1.000	59.946	1.001	59.865	1.000	59.669	0.997
	EDP (MJ·s)	4.522	4.519	0.999	4.523	1.000	4.520	1.000	4.504	0.996
SWEEP3D	Time (s)	81.21	81.91	1.009	81.65	1.005	81.39	1.002	81.12	0.999
	Energy (KJ)	69.136	69.500	1.005	69.445	1.004	69.520	1.006	69.043	0.999
	EDP (MJ·s)	5.615	5.693	1.014	5.670	1.010	5.658	1.008	5.601	0.998
FT	Time (s)	984.40	986.03	1.002	984.23	1.000	986.87	1.003	984.64	1.000
	Energy (KJ)	654.804	516.068	0.788	524.720	0.801	500.274	0.764	513.028	0.783
	EDP (MJ·s)	644.588	508.856	0.789	516.445	0.801	493.703	0.766	505.147	0.784
IS	Time (s)	128.16	128.24	1.001	128.32	1.001	128.83	1.005	128.78	1.005
	Energy (KJ)	82.708	68.936	0.833	68.940	0.834	65.672	0.794	65.616	0.793
	EDP (MJ·s)	10.600	8.840	0.834	8.847	0.835	8.461	0.798	8.450	0.797
BT	Time (s)	929.97	955.45	1.027	949.95	1.021	951.79	1.023	945.11	1.016
	Energy (KJ)	610.476	563.178	0.923	555.549	0.910	552.973	0.906	546.529	0.895
	EDP (MJ·s)	567.721	538.089	0.948	527.743	0.930	526.315	0.927	516.529	0.910
AZTEC	Time (s)	51.59	55.22	1.070	53.73	1.042	54.61	1.059	53.69	1.041
	Energy (KJ)	33.238	29.595	0.890	29.989	0.902	29.718	0.894	29.598	0.890
	EDP (MJ·s)	1.715	1.634	0.953	1.613	0.941	1.623	0.946	1.589	0.927
SMG2000	Time (s)	27.19	27.282	1.003	27.321	1.005	27.30	1.004	27.37	1.006
	Energy (KJ)	19.488	19.282	0.989	19.132	0.982	15.766	0.809	15.642	0.803
	EDP (MJ·s)	0.530	0.526	0.993	0.523	0.986	0.430	0.812	0.428	0.808
CG	Time (s)	373.31	385.44	1.032	394.97	1.058	388.07	1.040	395.87	1.060
	Energy (KJ)	254.364	227.672	0.895	221.020	0.869	223.994	0.881	218.052	0.857
	EDP (MJ·s)	94.958	87.753	0.924	87.297	0.919	86.925	0.915	86.318	0.909
MG	Time (s)	78.08	80.13	1.026	79.63	1.020	79.14	1.014	79.05	1.012
	Energy (KJ)	59.208	53.132	0.897	52.621	0.889	52.466	0.886	52.208	0.882
	EDP (MJ·s)	4.623	4.258	0.921	4.191	0.907	4.152	0.898	4.127	0.893
SP	Time (s)	910.51	936.99	1.029	933.10	1.025	938.78	1.031	936.50	1.029
	Energy (KJ)	778.150	690.560	0.887	681.844	0.876	678.988	0.873	671.022	0.862
	EDP (MJ·s)	708.515	647.046	0.913	636.226	0.898	637.422	0.900	628.415	0.887
LU	Time (s)	627.79	645.02	1.027	641.43	1.022	643.28	1.025	642.98	1.024
	Energy (KJ)	520.428	456.520	0.877	451.973	0.868	451.117	0.867	450.304	0.865
	EDP (MJ·s)	326.721	294.464	0.901	289.936	0.887	290.196	0.888	289.530	0.886
SPPM	Time (s)	460.33	465.35	1.011	468.09	1.017	463.65	1.007	463.92	1.008
	Energy (KJ)	374.392	367.624	0.982	365.563	0.976	364.537	0.974	363.820	0.972
	EDP (MJ·s)	172.344	171.073	0.993	171.115	0.993	169.017	0.981	168.785	0.979

System Overhead This paragraph discusses the overhead of the system in two parts. The first part is the overhead of changing p-states. As stated above, on this AMD-64 microprocessor the cost to shift between p-states is between 270 and 5600 microseconds, with the median about 570 microseconds [1]. The actual cost of this overhead depends on the number of shifts between p-states, which is twice the number of reducible regions. Table 3.3 shows that the shortest average region length is approximately 112 milliseconds, which is more than 50 times longer than the average shift cost. Thus the overhead due to shifting is nominal.

The second part of the overhead is due to data collection, which happens at the beginning and end of each MPI call. The overhead consists of three parts: hijacking the call, reading system time, and reading the performance counters. The first two parts occur every MPI call, while the latter occurs only at the beginning and end of each region. The cost of all three parts is approximately 1.3 microseconds. Because shifting p-states changes elapsed time, we cannot directly compare the overhead of our technique to the baseline, which does not shift. To measure the overhead of the algorithm only, we modified the system to perform all steps except shifting. In this case, we can compare elapsed time to the baseline. The relative cost of this overhead is less than 1% on all the NAS programs (the greatest overhead is 0.38% for LU).

False negatives and positives Figure 3.9 graphically compares *simple* and *composite* with two p-state transition examples. In the first case (Figure 3.9(a)), both perform the same with one false negative during training. If no MPI calls ever repeat, as occurs in EP, then *simple* and *composite* are the same. The second case (Figure 3.9(b)) shows the advantage of *composite*, or rather the shortcoming of *simple*. Because call A alternates between beginning and ending a region, *simple* always mispredicts. This results in both false negatives and false positives as the first call in the pair is a false negative, and the user code following this region is a false positive.

Figure 3.10 shows the time analysis of *simple* and *composite*. The algorithms are applied postmortem to the collected trace data. In practice, reducing the p-state can increase time. Furthermore, there is overhead in shifting p-states and executing each algorithm. This analysis does not consider either of these factors because adding such costs analytically is approximate at best. Rather, this analysis shows the *potential* impact by showing how much of the base code is affected by our scaling.

This analysis divides the unscaled, base time into four categories along two dimensions: (a) *IN* or *OUT* of a reducible region and (b) *true* or *false* prediction. Correct predictions are

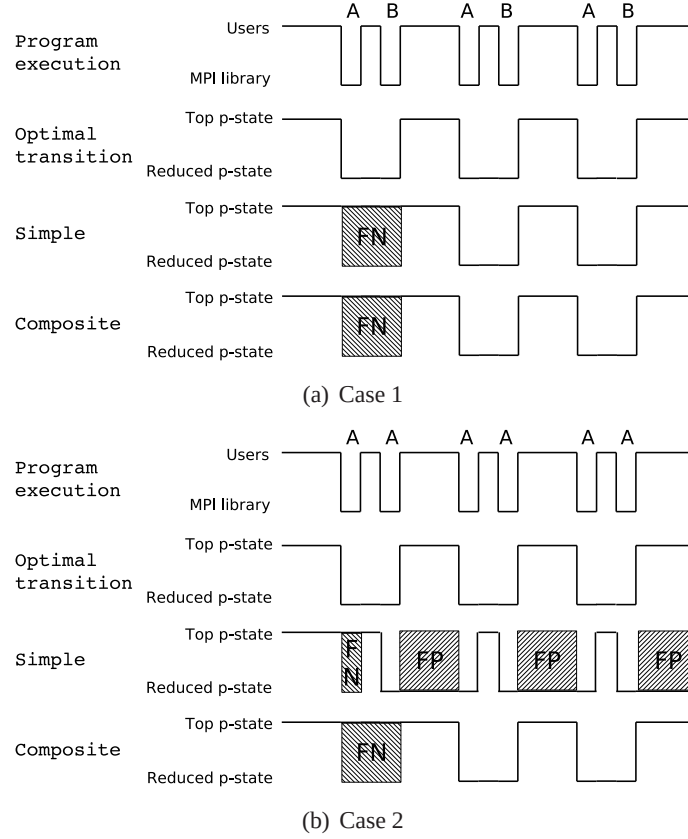


Figure 3.9: These are p-state transition examples by different MPI call patterns. FN and FP mean false negative and false positive, respectively.

determined by examining trace data, as is done in *profile*. In Figure 3.10, bars labeled “IN (true)” and “OUT (true)” are correctly predicted. The bar labeled “IN (false)” is a false positive; the application mistakenly executes in a reduced p-state instead of the top p-state. The bar labeled “OUT (false)” is a false negative; the application mistakenly executes in the top p-state instead of a reduce one. Generally, a false positive is considered worse than a false negative because it can result in a time penalty. In contrast, a false negative represents a missed opportunity to save energy.

For SWEEP3D, 11.6% of the total execution time is considered as reducible regions in the *composite* algorithm. However, 72% among them turns out to be misprediction among them. *simple* also suffers from many false negatives. So, our algorithms could not reduce EDP. For FT and IS, *simple* and *composite* have false negatives, which explains why they are worse in EDP than *by-call*. Thus, *simple* and *composite* are slightly faster but consume more energy. Furthermore, with singleton regions there is no difference in the predictive power of *simple* and *composite*. These plots

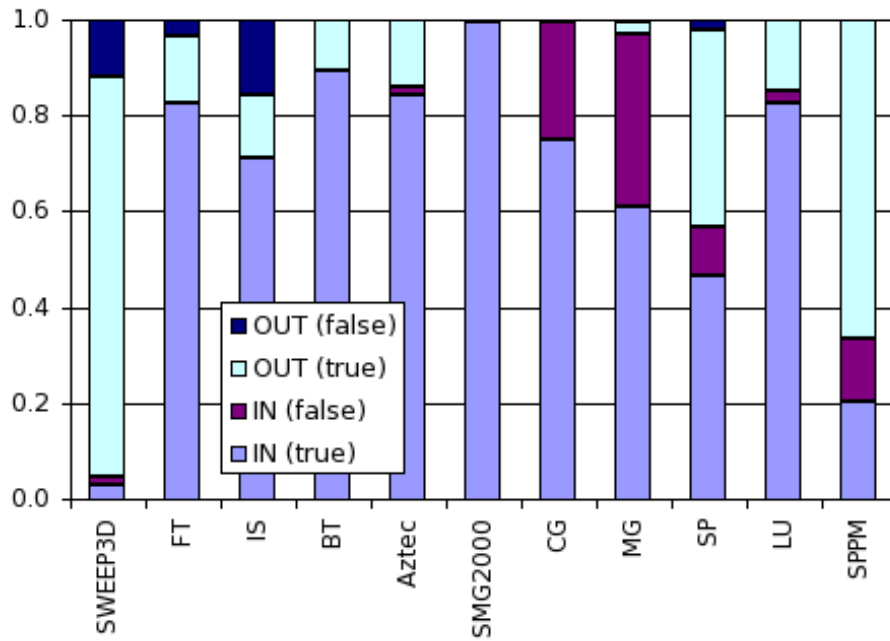
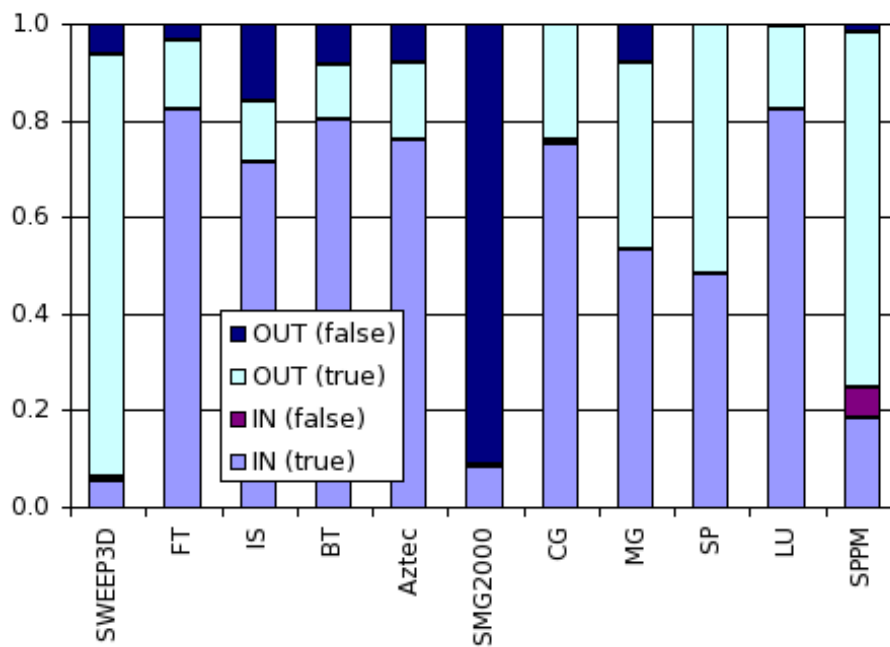
(a) *simple*(b) *composite*

Figure 3.10: Breakdown of execution time for adaptive region-finding algorithms with $\tau = 20$ ms and $\lambda = 20$ ms.

show that the longer training period in *composite* adds false negatives to BT, Aztec, and SMG2000. They have 8.2, 7.8, and 90.9% more false negatives than *simple*, respectively. In these benchmarks *composite* has fewer false positives than *simple*, but adds false negatives due to longer training periods. For the remaining five benchmarks, *composite* eliminates significant mispredictions compared to *simple*. For CG, SP, and LU, there are in fact no mispredictions in *composite*. For MG, all false positive cases are eliminated as well. In SPPM, false positive cases are significantly reduced, from 13.3% to 6.2%.

For these benchmarks, the problem patterns for *simple* are ... BABA ... and ... A ... BAC Those explain the improved performance of *composite* over *simple*. Because call A alternates between beginning and ending a region, *simple* always mispredicts. This results in both false negatives and false positives: the first call in the pair is a false negative, and the user code following this region is a false positive. However, the time of the false positive is much longer than the false negative. Additionally, there are many more regions found when using *simple* than with *composite*, which adds transition overhead.

Choosing the close-enough threshold (τ) Figure 3.11 shows data for evaluating τ , the threshold that determines whether two MPI routines are “close enough.” This evaluation is done with *composite*. Each figure is horizontally divided into two parts. The top-half plots three metrics derived from profile data: the fraction of time for false negatives, the fraction of time for false positives, and the number of regions. The bottom-half plots three empirical measures: the execution time, the energy, and the EDP.

Clearly, there is a tension in the choice of τ . If τ is too large, then a few, large regions will result. This means that when using *composite*, a large number of false negatives would result, which in turn would result in a significant fraction of time due to them. For example, if $\tau = \infty$, the training component will be run during the whole program, and the executing component will never run—essentially the entire program (between first and last MPI call) becomes a false negative. On the other hand, if τ is too small, then *composite* would approach *by-call*, which we know from Section 3.3.1 is often an ineffective algorithm due to excessive time due to p-state shifting.

Indeed, the figure shows that the value of τ has a significant impact on benchmarks that have several short MPI calls (IS and FT do not show noticeable change in their figures). The goal is, essentially, to choose for τ the smallest value possible such that the region overhead (which is proportional to the number of regions) is small and false predictions are low. All things considered,

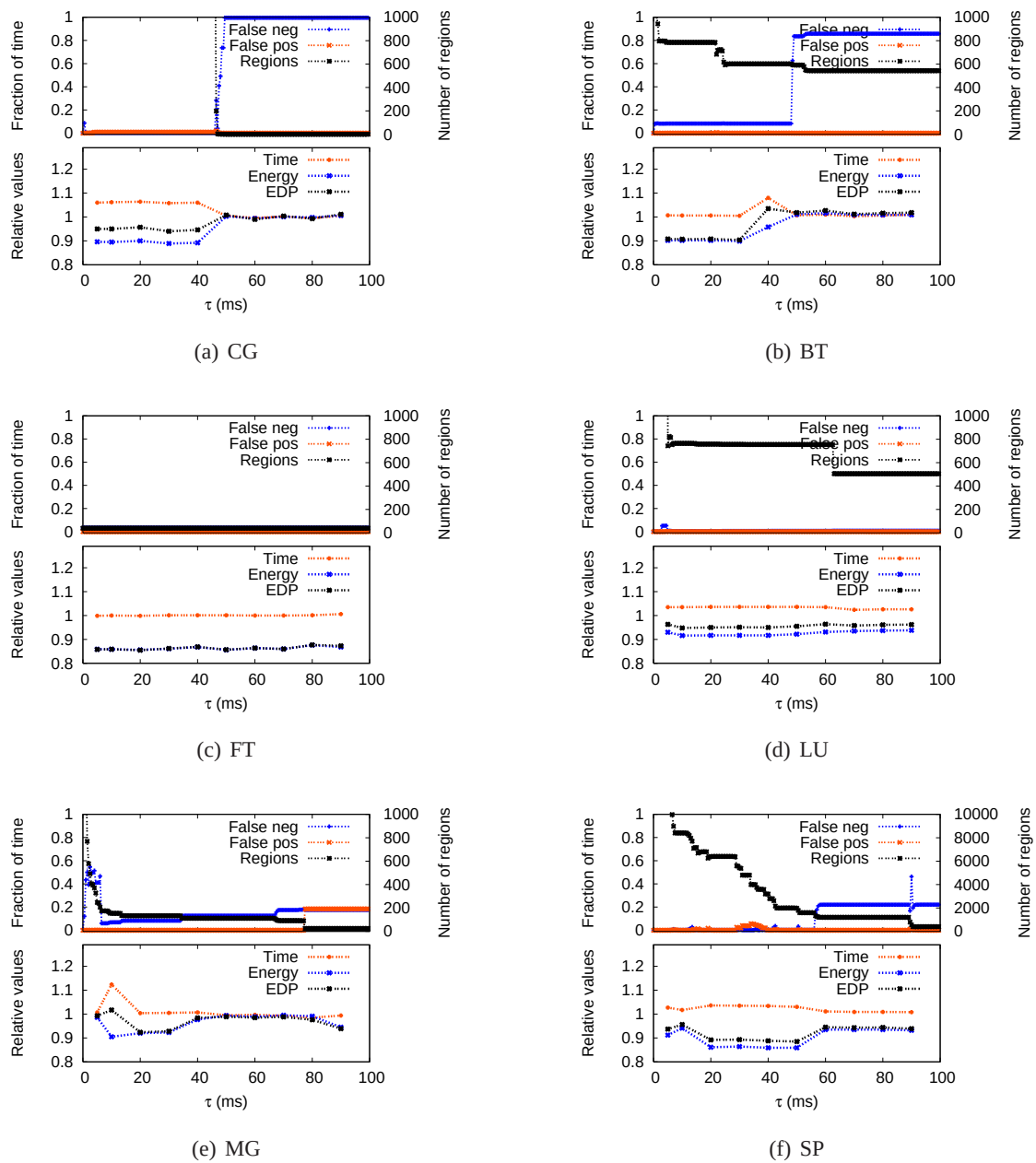


Figure 3.11: Evaluating τ in *composite*. Show fraction of time mispredicted and number of regions (right-hand y-axis) as a function of τ . For readability, right-hand y-axis has different scale for CG and SP. (Continued in Figure 3.12.)

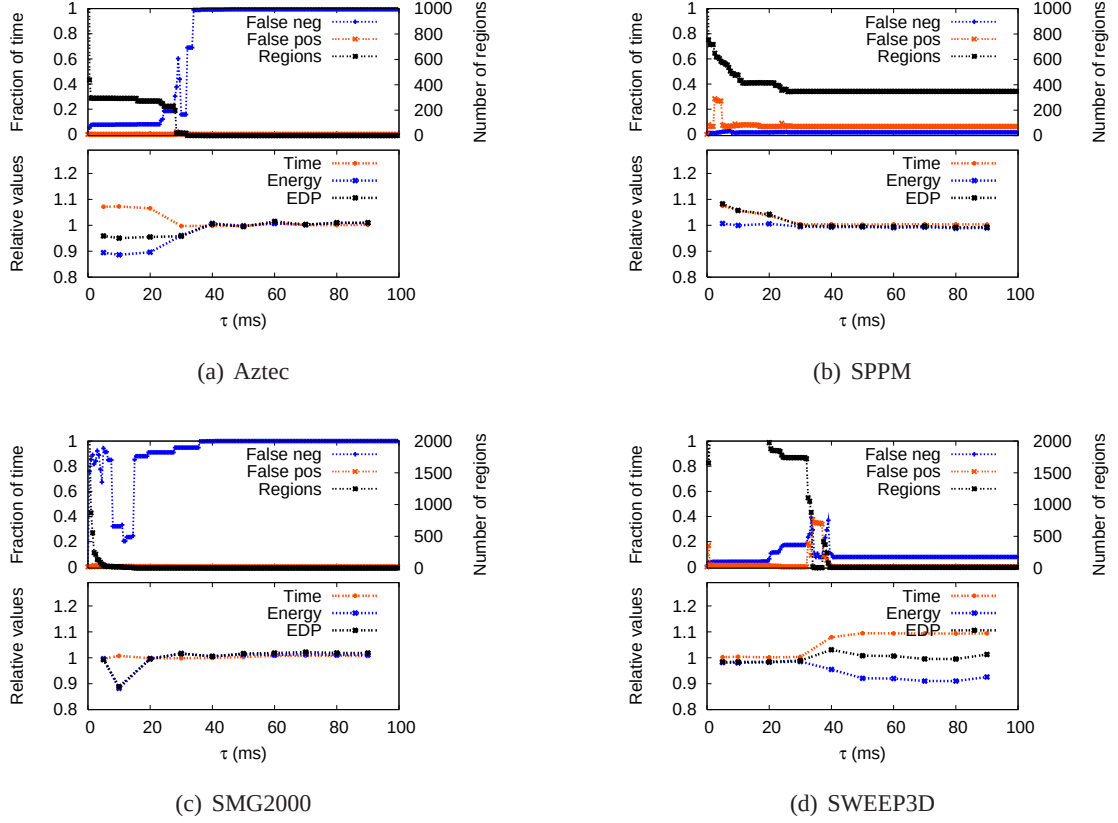


Figure 3.12: Evaluating τ in *composite*. Show fraction of time mispredicted and number of regions (right-hand y-axis) as a function of τ . For readability, right-hand y-axis has different scale for CG and SP. (Continuation of Figure 3.11.)

we can achieve the best EDP between 10 and 20 ms, though there is not a choice that works for all benchmarks. Again, we chose 20 ms for the value of τ for all benchmarks. The one problematic application is SMG2000, which has a lot of false negatives—32% of total execution time in 10 ms and 91% in 20 ms. In this case, it is likely that our approach does not determine right communication pattern due to nondeterministic behaviors at runtime. We plan to investigate this problem as future work. For SP, the choice of a longer τ can reduce EDP by more than 0.6%.

Choosing long-enough threshold (λ) The duration of MPI calls over all benchmarks is generally small—Section 3.1 showed that 95% of the calls take less than 10 ms. Basically, there is an extreme bimodal distribution of times for MPI calls over the programs, because the calls in IS and FT take 3 and 18 *seconds*, respectively. We apply the long enough threshold λ to the composite algorithm

to amortize the system scaling overhead. As it increases, the overhead becomes less significant but we lose more chances for p-state transition. In our experiments, we empirically choose 10 or 20 ms, the same as τ . However, this is not optimal for all benchmarks. For example, we achieve 2% improvement in terms of EDP with 10 ms in the SWEEP3D benchmark. We do point out that for different benchmarks, where the calls may be more evenly distributed, a different approach might be needed to choose λ .

3.4 Summary

This chapter has presented a transparent, adaptive system for reducing the p-state in MPI communication phases. While communicating among nodes, CPU is not on the critical path, which allows us to achieve significant energy savings with little performance degradation. Frequent p-state transitions by short MPI operations, however, may result in serious performance penalty without benefit in energy savings. We define communication regions by clustering close MPI calls together.

Our algorithms are designed to find such regions on the fly by monitoring MPI calls and keeping a state machine that recognizes the regions. Then, we use performance counters to guide our system to choose the best p-state in terms of energy-delay product (EDP). User programs can use our runtime system with zero user involvement. Results on the NAS benchmark suite and ASCI Purple benchmark showed 7.5% reduction in EDP and 1.92% in time on average.

Our future plans include integrating the idea of reducing the p-state during communication regions with our past work on reducing the p-state during computation regions. Additionally, we will conduct these experiments on a newer cluster, which has a gigabit network and multi-core, multiprocessor nodes. For computation, we leverage the memory or node bottleneck to save energy, sometimes with no increase in execution time. The overall goal is to design and implement one MPI runtime system that simultaneously exploits all three bottlenecks.

Chapter 4

Determining the Minimum Energy Consumption

As we mentioned earlier, changing p-state is a popular and effective technique for increasing energy efficiency. However, its effectiveness is highly dependent on the characteristics of applications running on a system. Simply speaking, if the CPU is on a critical path, the execution time will be increased in proportion to frequency decrease. On the other hand, if the bottleneck is caused not by CPU but by other factors, such as I/O or network communication, then execution time will not increase as much. Therefore, applying CPU scaling judiciously can result in energy savings with little or no time delay.

The problem we address in this chapter is to establish an optimal baseline for DVFS scaling for any application. In other words, we determine the minimum energy consumption for a given time delay. Given the baseline, one can evaluate a specific DVFS technique. We assume that we are given a finite number of program locations where shifting may occur. The locations define a set of regions in programs. Then, we assume we know the time and energy of each region for each p-state. Further, we suppose that we have the time and energy to transition between any two p-states. A schedule is a list of p-states—one for each region. Our algorithm finds the schedule with the minimum energy from those under the given time limit. There are a potentially large number of schedules: $\mathcal{O}(F^R)$ where F is the number of available p-states and R is the number of regions. However, an efficient scheduling of the p-state transitions may lead to find the optimal solution in which the minimum energy consumption occurs within a given time delay.

The research in this chapter was supported in part by IBM UPP and SUR awards.

Our problem is equivalent to the problem of finding a path minimizing cost from the root node to the leaf node in a tree. Each node has F (available p-states) children. There exist R (number of regions) levels in the tree. Given this tree, finding the optimal path satisfying problem requirements requires the same computational complexity ($\mathcal{O}(F^R)$). A brute-force solution is intractable even for a moderately sized set (although all programs presented in this chapter can be solved with the brute-force). Branch and bound [72] is a general algorithmic method for finding exact optimal solutions for various optimization problems. There are several branching ways, such as depth-first-search and breadth-first-search. The bounding is a fast way of finding bounds (upper or lower) for the optimal solution within a feasible subregion. The efficiency of the method depends critically on the effectiveness of the branching and bounding algorithms used. The paper [41] proposed a scheduling algorithm using it, which is evaluated on architectural simulators. However, we use a branch and bound method with three branching orders after determining the scaling regions on real benchmark applications. We develop our algorithm with a unique branching technique and bounding function to reduce the processing time significantly.

In our approach, we build time and energy models and use empirically obtained data. Then, we design a scheduling algorithm in which the optimal schedule with minimum energy consumption within given time delay is determined with precise estimation. For evaluation of our approach, we use programs from the NAS serial benchmark suite. Our results verify that our time and energy estimations are highly accurate. In addition, we show that our algorithm is efficient—obtaining huge reduction in running time over the brute-force method.

Section 4.1 and Section 4.2 describe our approach and implementation respectively. It includes our time and energy estimation model and scheduling algorithm. The section 4.3 provides the evaluation of our approach and the verification of our scheduling estimates. Then, we close with the summary, point out limitations of our approach.

4.1 Approach

The objective of this chapter is to choose a DVFS schedule that minimizes energy consumption while satisfying the allowable time delay in uni-processor programs. The idea is to explore all possible schedules for p-state transitions in program execution. As we mentioned before, we may get different time/energy tradeoffs depending on the characteristics of applications. In this work, we assume that we are given a maximum allowable time, T_{Limit} ($T_{Limit} \geq T_0$, T_0 is the execution

time at the highest frequency). We next present our time and energy model. Then, we describe our scheduling algorithm to find the optimal schedule.

4.1.1 Modeling

In this section, we present our model of execution time and energy consumption. We denote $\mathcal{F}$ is a set of available p-states and $\mathcal{R}$ is a set of regions. Moreover, F is $|\mathcal{F}|$ and R is $|\mathcal{R}|$. First, we are given the time and energy for each region for each p-state, i.e.,

$$T_f^r \text{ and } E_f^r, \forall r \in \mathcal{R} \text{ and } \forall f \in \mathcal{F}$$

Second, we are given the time and energy to transition between p-states, i.e.,

$$T_T(f, g) \text{ and } E_T(f, g), \forall f, g \in \mathcal{F}$$

where $T_T(f, g)$ and $E_T(f, g)$ are the time and energy to transition from p-state f to p-state g , respectively (Section 4.2 explains how we collect the above data.). Then, the problem is to find a schedule (S) which is a list of p-states:

$$S = (f_1, f_2, f_3, \dots, f_R)$$

that satisfies the following conditions for a given delay, T_{Limit} :

$$\hat{T}(S) = \sum_{i=1}^R T_{f_i}^i + \sum_{i=2}^R T_T(f_{i-1}, f_i) \leq T_{Limit}, \text{ and}$$

$$\hat{E}(S) = \sum_{i=1}^R E_{f_i}^i + \sum_{i=2}^R E_T(f_{i-1}, f_i) \text{ is minimized.}$$

$\hat{T}(S)$ and $\hat{E}(S)$ are estimated values computed from the given time and energy data and the optimal schedule. In Section 4.3, we execute the program with the optimal schedule and measure the actual execution time, $T(S)$, and energy consumption, $E(S)$.

Region	Q1	Q2	Q3	Q4	P-State Search Order with 5% Time Delay
R1	[]	[4, 5]	[0, 1, 2, 3]	[6]	[5, 4, 3, 2, 1, 0, 6]
R2	[]	[3, 4, 5]	[0, 1, 2]	[6]	[4, 5, 2, 3, 1, 0, 6]
R3	[]	[2, 3, 4, 5]	[0, 1]	[6]	[4, 5, 2, 3, 1, 0, 6]
R4	[]	[5]	[0, 1, 2, 3, 4]	[6]	[5, 4, 3, 2, 1, 0, 6]
R5	[]	[2, 3, 4]	[0, 1]	[5, 6]	[2, 3, 4, 1, 0, 5, 6]

Table 4.1: Analysis per region in LU benchmark

4.1.2 Scheduling algorithm

In brute-force approach, F^R schedules must be evaluated to determine the schedule with minimum energy consumption and at most a given time delay. In this section, we first describe our algorithm in detail. Then, we introduce p-state search ordering to improve the efficiency of our scheduling algorithm.

Finding the optimal schedule

Our algorithm requires the following inputs: an allowable time delay (T_{Limit}), time and energy per region per p-state (T_f^r, E_f^r), and time and energy for all possible pairs of p-state transitions (T_T, E_T). The scheduling algorithm needs to explore a state space tree to determine the optimal schedule. The state space tree is a tree with $\sum_{r=0}^R F^r$ nodes to express F^R schedules in leaf nodes. Each internal node has F children. An exhaustive scheduling algorithm will visit all the nodes in the tree. Therefore, finding the optimal schedule through the exhaustive algorithm demands a huge computational complexity.

In our approach, we have designed a scheduling algorithm that is based on the branch and bound algorithm [72]. Thus, our algorithm finds the exact optimal schedule with the same worst case complexity as the exhaustive method. In order to achieve the significant reduction in running time, however, we define a branching way and two bounding functions. The branching way is a best-first-search method [73] in which we first look at the p-state with a lower index in the p-state search list. That reduces the chances of visiting unnecessary nodes because it is highly probable that the remaining nodes will be pruned by the bounding functions later. Our bounding functions are specified from observations in time and energy estimations, respectively.

First, we estimate the time at the k -th level in the tree indicated by the partial schedule,

$S_k = (f_1, f_2, \dots, f_k)$:

$$\hat{T}(S_k) = \sum_{i=1}^k T_{f_i}^i + \sum_{i=2}^k T_T(f_{i-1}, f_i)$$

Then, we check if this partial schedule is valid in time as defined below:

If $\hat{T}(S_k) + MIN_T(k) \leq T_{Limit}$ then **valid**

$$\text{where } MIN_T(k) = \sum_{i=k+1}^R T_0^i$$

T_{Limit} is a given maximum execution time and $MIN_T(k)$ is the minimum execution time for the $R - k$ remaining regions. This time bounding function implies we do not need to follow the current path in the tree because the minimal value of our estimation already exceeds the maximum time limit.

Next, we estimate the energy for the same schedule:

$$\hat{E}(S_k) = \sum_{i=1}^k E_{f_i} + \sum_{i=2}^k E_T(f_{i-1}, f_i)$$

Then, we check if this partial schedule is valid in energy as defined below:

If $\hat{E}(S_k) + MIN_E(k) \leq E_{Best}$ then **valid**

$$\text{where } MIN_E(k) = \sum_{i=k+1}^R \min(E_{f_j}^i \mid f_j \in \mathcal{F})$$

E_{Best} is a minimum energy consumption found until this point in the algorithm, and $MIN_E(k)$ is the lowest possible energy consumption for the $R - k$ remaining regions. This energy bounding function implies that there is no need to look at the path in case that our estimated value is greater than the current minimum energy.

P-state search ordering

At each node there are F children to be evaluated. The order in which those children are evaluated can have a significant effect on the running time. In the above algorithm, we always explore the child node in the same p-states order, such as increasing p-state order. This may cause

unnecessary node visits in the scheduling. Therefore, we identify a p-state search order per region to reduce its searching space so that we reach the optimal schedule as fast as we can. The p-state search order contains all available p-states in the order where most likely selected p-state in the optimal schedule is considered first.

There are several ways to specify the ordering. In this chapter, we use three ways to order p-states: ordering by time, energy, and quadrant. By time, the p-state list is sorted from faster to slower execution time. By energy, the p-state with lower energy consumption is taken first. The ordering by quadrant divides the p-state list into 4 groups by the baselines of allowable time delay and energy. The groups are ranked by probability in the following order: the group with less time and less energy (Q1), the group with more time but less energy (Q2), the group with less time but more energy (Q3), the group with more time and more energy (Q4). The p-states in each group is also ordered by EDP because the p-state with lower EDP has high probability to be chosen in the optimal schedule. Table 4.1 gives the p-state search order per region after ordering by quadrant in the NAS LU benchmark.

4.2 Implementation

For any application, we first collect the time and energy data empirically. Next, we find the optimal schedule by applying our algorithm. Then, we verify our estimation to validate the schedule in the actual algorithm. In this section, we describe several technical issues in the procedure.

4.2.1 Time and energy data collection

We need to collect data for each program. To do so, the target program is repeatedly executed for all p-states without transitions. On each run, we measure time (T_f^r) and energy (E_f^r) for each region r and each p-state f .

We also need the time (T_T) and energy (E_T) during p-state transitions. Our research was carried out on AMD Athlon64 processor that is capable of DVFS. In the AMD, the p-state is defined by a frequency identifier (FID) and voltage identifier (VID) pair. The p-state is changed by storing the appropriate FID-VID to the *model specific register* (MSR). The p-state transition has three phases. In the first phase, the processor voltage is transitioned to the level required to support frequency transitions. Next, the processor frequency is transitioned to the frequency associated with OS-requested p-state. In the last phase, the processor voltage is transitioned to the voltage associated

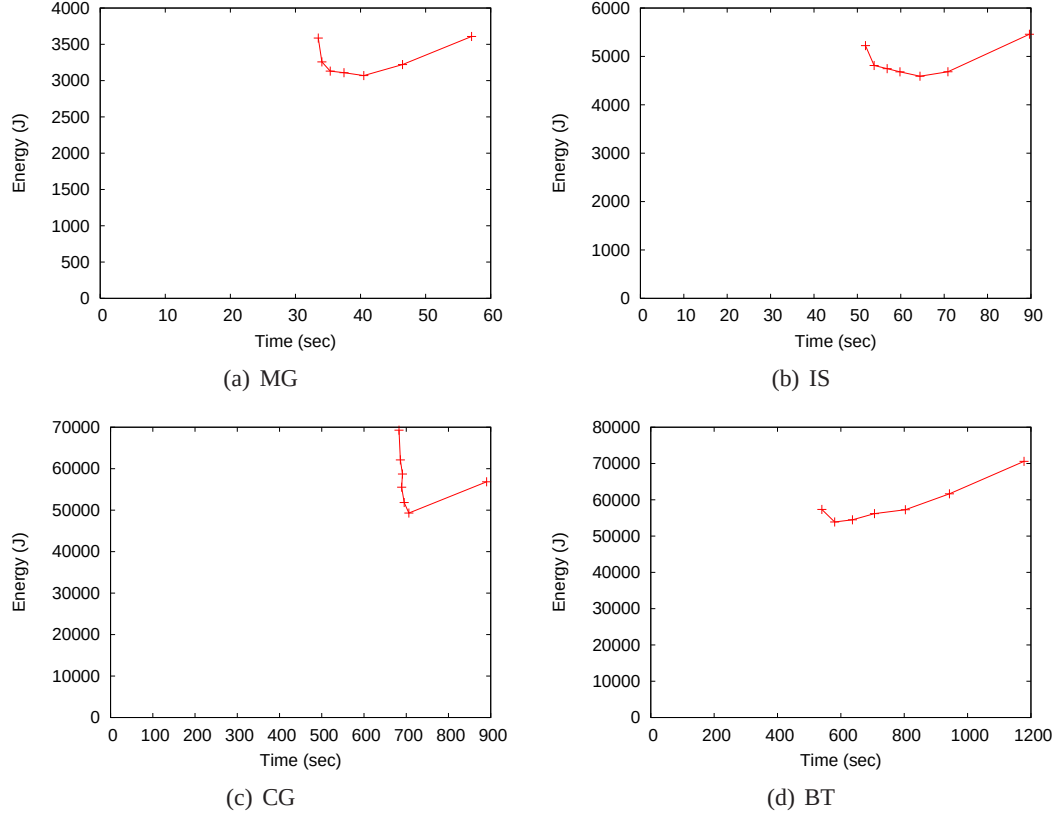


Figure 4.1: Execution time vs. energy consumption from single p-state execution

with the OS-requested p-state. Such a store is privileged, so it must be performed by the kernel. The *cpufreq* and *powernow* modules provide the basic support for this. Some additional tool support and significant “tweaking” of the FID-VID pairs was needed to refine the implementation. As a result, the p-state is changed with a system call. The overhead of changing the p-state is dominated by the time to scale—not the overhead of the system call. For the processor used in our implementation, the time overhead on a p-state transition (including system call overhead) is in the range between 280 microseconds and 5647 microseconds, measured in Table 3.2. It is noted that time increases as the gap between p-states increases. This is because the maximum voltage step is prespecified, and we have more steps to the requested voltage level. We calculate the energy consumption by multiplying the time and the idle power of target p-state

The time and energy per region are context-insensitive. These data are dependent on the current p-state for a region exclusively, not the p-state before or after the region. Therefore, we can

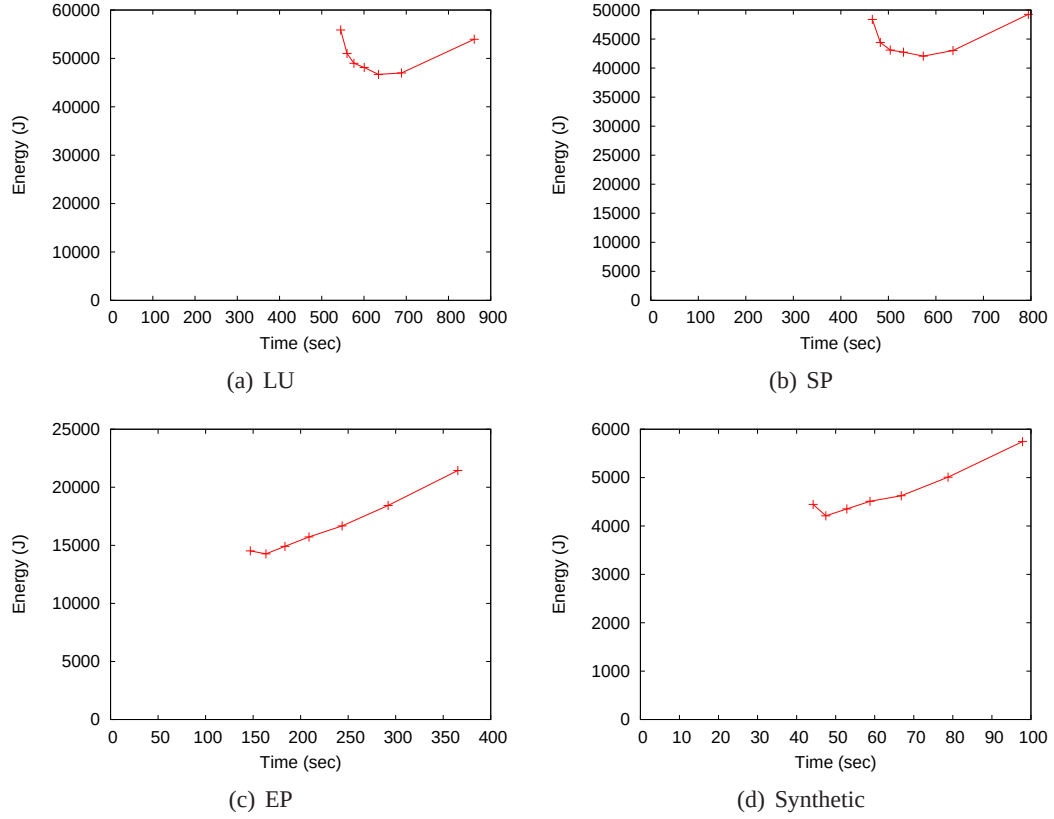


Figure 4.2: Execution time vs. energy consumption from single p-state execution

compose p-state transition schedule.

4.2.2 Power measurements

In order to measure the energy consumption of the whole system, we use a WattsUp Pro power meter. The meter is directly attached between the wall power and the system's power connector. serial interface is used to allow the system to monitor its own power consumption. Our meter samples the waveform 4,000 times per second (once per 250 microseconds) at maximum electrically. However, the meter only reports its measurements at one second intervals. It causes a precision problem of power meter because the current power usage is always emitted after 1 second at maximum. This makes direct energy measurements of regions infeasible. Therefore, we store timestamps of all regions during program executions and also generate the log for power usages

separately. From the power logs, we know the average of power usage every a second. For each timestamp, we can take the latest power usage before the time by comparing with the logs. Then, we estimate energy consumption per region by multiplying the elapsed time by the power matched in the log.

4.2.3 Finding regions

In our implementation, the regions in target programs are specified statically by hand for each application. To do this, we edit the source code of target programs. We recognize this could be improved by automatic insertion. Furthermore, it would be selected from strong analysis for ideal region findings.

We define a data collecting operation as an entry point of region. The region is specified as the code section between two consecutive operations. The operations have two execution modes: *profiling* and *scaling*. In the *profiling* mode, it emits the current timestamps (by *gettimeofday* call) and its program location. The *location identifier* is defined as a pair of hash values of program counters in call history and its counting value to uniquely identify an execution point. In the *scaling* mode, it executes the assignment of schedule by shifting appropriate p-state.

After inserting operations, regions are defined statically in program context. Grouping several operations into one region may be possible depending on the size of regions. Because of the transition time between p-states, we do not consider the region with the time less than 200 milliseconds. NAS serial benchmarks we evaluated have very small code sequence. This limits the number of static regions. For example, only three regions are found in CG benchmark because there is a nested loop and most of time is spent in inner loop.

We also collapse all regions in iterative sections into one region. We can do this for the NAS benchmark because the time and energy for a region do not change over time. However, this is not true for all applications. But our data is sufficient to tell when collapsing is appropriate.

The effectiveness of our approach relies on correct partitioning of regions. Coarse-grained (only a few regions are defined) or biased (one region takes most of the execution time) partitionings make our approach invalid. In this case, our resulting schedule can never be close to real optimal energy savings. We regard the general phase partitioning as orthogonal and complementary research to our work.

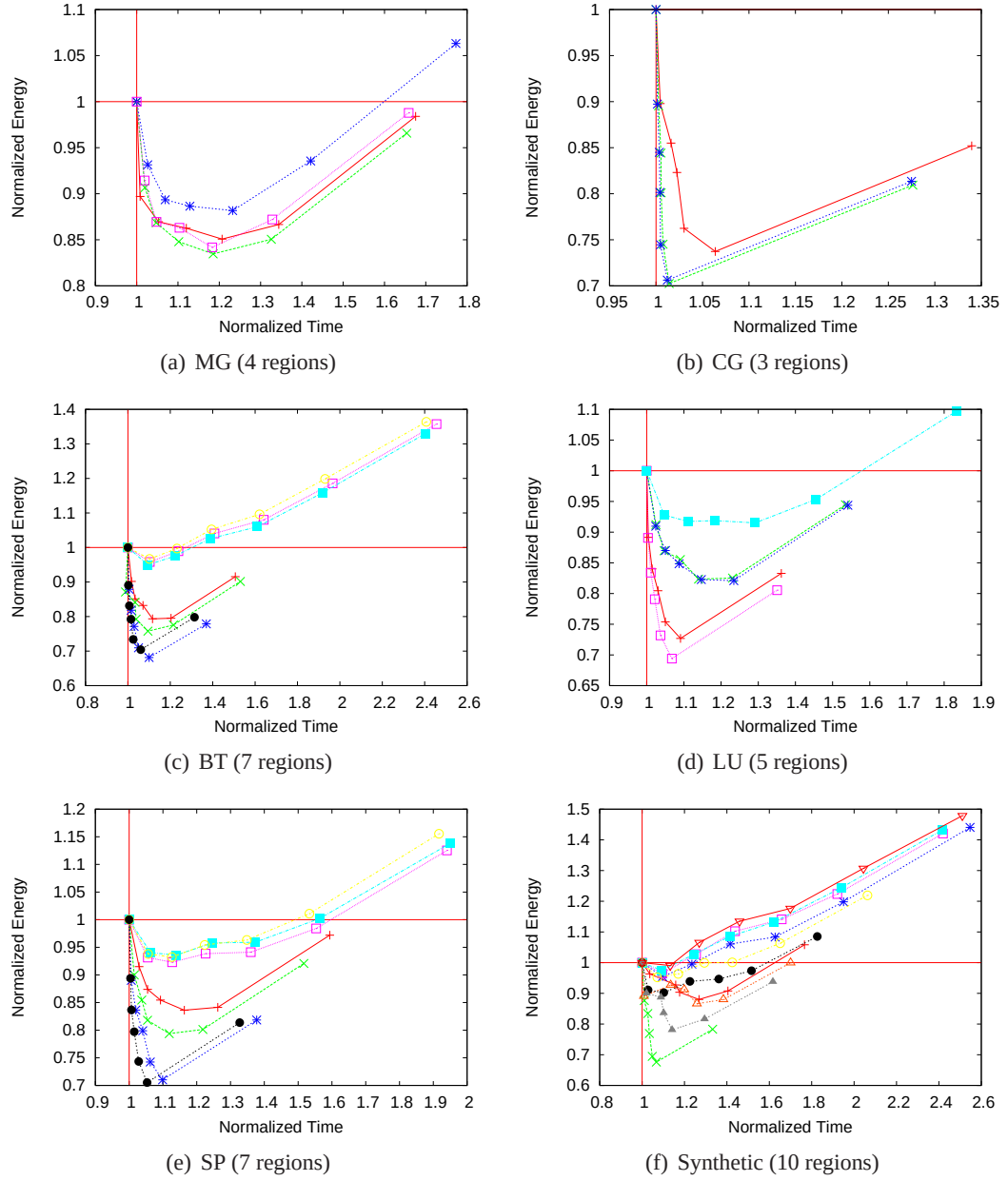


Figure 4.3: Execution time vs. energy consumption per region

4.3 Results

For all experiments, we used a single-node AMD Athlon-64 system with 1 GB of main memory. The Athlon-64 CPU supports 7 p-states (from 2000 to 800 Mhz). The system runs the

Table 4.2: The workload analysis of finding the best schedule

		Brute-force (F^R)	Ordering by time			Ordering by energy			Ordering by quadrant		
			5%	10%	15%	5%	10%	15%	5%	10%	15%
MG (R=4)	Schedules	$2.401 * 10^3$	23	37	5	5	3	3	5	4	5
	Nodes	2801	71	110	119	30	40	23	26	41	25
	Time (sec)	0.059	0.005	0.008	0.008	0.002	0.003	0.002	0.002	0.003	0.002
CG (R=3)	Schedules	$3.430 * 10^2$	40	40	40	1	1	1	1	1	1
	Nodes	400	58	58	58	4	4	4	4	4	4
	Time (sec)	0.005	0.003	0.003	0.003	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003
BT (R=7)	Schedules	$8.235 * 10^5$	94	129	132	3	2	1	3	2	1
	Nodes	960800	782	274	273	343	19	8	343	10	8
	Time (sec)	14.398	0.093	0.027	0.027	0.102	0.002	0.001	0.091	0.002	0.001
LU (R=5)	Schedules	$1.680 * 10^4$	64	120	166	4	4	1	3	3	5
	Nodes	19608	183	319	296	65	121	11	64	120	19
	Time (sec)	0.256	0.015	0.026	0.022	0.007	0.020	0.001	0.006	0.035	0.001
SP (R=7)	Schedules	$8.235 * 10^5$	96	161	167	7	3	1	4	3	1
	Nodes	960800	510	520	512	223	24	8	165	24	8
	Time (sec)	14.828	0.068	0.061	0.058	0.071	0.003	0.001	0.068	0.003	0.001
Synth etic (R=10)	Schedules	$2.825 * 10^8$	73	54	55	12	7	1	4	7	2
	Nodes	$3.296 * 10^8$	1485	290	254	632	169	11	269	95	14
	Time (sec)	6007.005	0.224	0.038	0.033	0.095	0.024	0.001	0.04	0.013	0.001

Fedora Core 3 OS and Linux kernel 2.6.8. All applications were compiled with *gcc* compiler 4.0.2 or the Intel Fortran compiler 8.1 using the O2 optimization.

We use applications from the NAS serial benchmark suite for evaluation purpose. We test class B of these benchmarks. We do not use FT because it takes more than 10 hours for each execution.

4.3.1 Detailed analysis

First, we describe the energy-time tradeoff for six NAS benchmarks using a single p-state. Figures 4.1 and 4.2 show the results of executing them on a single Athlon-64 processor. For each figure, the total energy consumption at each p-state is plotted on the y-axis and the total execution time is plotted on the x-axis. There are 7 points. Each of them indicates the p-state used. The higher of two points uses more energy, and the further right of two points takes more time. Therefore, a near-vertical slope indicates an large energy savings with little time delay between adjacent p-states, whereas a horizontal slope indicates a time penalty and no energy savings.

EP is a CPU bound benchmark, so there is little benefit through p-state transitions. On the other hand, CG is a memory bound benchmark which results in large energy savings and little time delay. All the others are more typical applications which are in the middle of the two extreme benchmarks. EP is not considered further.

Table 4.3: Estimation verification

		Time Delay (sec)	Estimated Time (sec)	Estimated Energy (J)	Measured Time (sec)	Measured Energy (J)	Time Error (%)	Energy Error (%)
MG	5%	34.97	34.75	3218.56	34.60	3257	-0.43	1.19
	10%	36.43	36.11	3168.91	35.81	3158	-0.83	-0.34
	15%	37.88	37.63	3142.34	37.20	3189	-1.14	1.48
CG	5%	715.98	697.16	49409.77	697.01	49165	-0.00	-0.50
	10%	749.16						
	15%	782.34						
BT	5%	566.68	566.20	53203.87	565.87	53521	-0.06	0.60
	10%	593.37	592.78	52132.53	594.42	52388	0.28	0.49
	15%	620.07	594.38	52070.00	594.24	52180	-0.02	0.21
LU	5%	571.60	571.34	49035.22	572.06	49094	0.13	0.12
	10%	598.50	598.33	47807.41	598.25	47891	-0.01	0.17
	15%	625.40	618.70	46755.73	618.52	46782	-0.00	+0.00
SP	5%	489.24	489.07	42807.32	489.12	42878	+0.00	0.17
	10%	512.33	510.99	41519.96	513.80	41994	0.55	1.14
	15%	535.43	525.59	41326.46	523.21	41158	-0.45	-0.41
Synthetic	5%	46.43	46.41	4163.49	46.11	4141	-0.65	-0.54
	10%	48.64	48.50	4130.29	48.18	4082	-0.66	-1.17
	15%	50.85	49.18	4121.60	49.70	4114	1.06	-0.18

In the Figure 4.3, we plot the execution time and the energy consumption per region by normalizing the time and energy values. For our evaluation, we figured out 1-10 regions by the region finding method described in Section 4.2.3. We use the average of time and energy values per region in case that there is one region shown repeatedly over iteration. The time and energy values per region have 0.68% and 2.78% of standard deviations on average.

The figures show that time and energy tradeoffs per region can be different each other. In other words, we get different energy savings and time penalty depending on the characteristics of code regions. In our experiments, we use the NAS benchmarks which are iterative with several timesteps during execution. So, we see that many regions have different patterns of tradeoffs because of the static partitioning in our implementation identifies different code sections as a separate region.

After figuring out the time and energy per region, we applied the scheduling algorithm to each benchmark. Table 4.2 presents overall statistics of the algorithms on each benchmarks to reach the optimal solution. Each benchmark has three values. The first row indicates the number of evaluated schedules which is equal to the number of leaf nodes we visited in the state space tree. The second row corresponds to the number of internal nodes visited while running the algorithm. The last row shows the running time to get the optimal schedule. There is a strong correlation between nodes and running time across all benchmarks. We compared the workloads of the scheduling with three different p-state search ordering by varying the given time delay from 5% to 15%. The re-

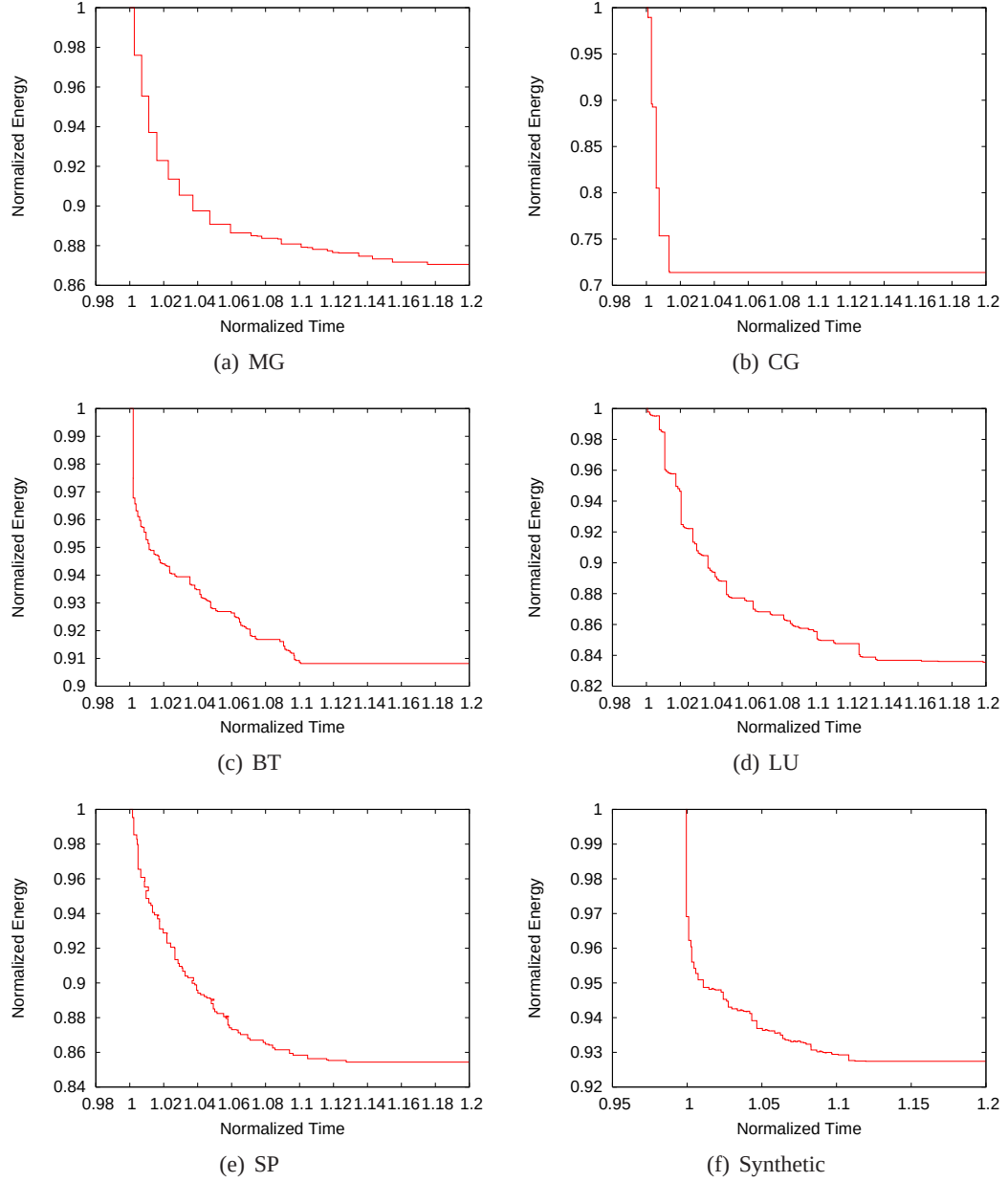


Figure 4.4: Minimum energy over time

sults reported our scheduling algorithm can reduce the scheduling workload significantly, compared with the number of possible schedules (F^R). Furthermore, ordering the p-state list provides more opportunities to get rid of node visits in the tree in most cases.

We verified our estimations of time and energy from the scheduling algorithm. The results are shown in Table 4.3. We first show the allowable time delay varying from 5% to 15%. Next, the estimated execution time and energy consumption are presented. These are generated by the optimal schedule of our scheduling algorithm. Then, we measured the real execution time and energy consumption by using the optimal schedule. In the last two columns, we calculated the differences between our estimation and real measurements in time and energy. For the accuracy of real measurements, we take the median values of time and energy after running the executions 5 times. In cases of IS and CG, the schedules are same for all given time delay because the minimum energy is achieved with less than 5% delay (IS is not shown in this chapter due to having only one region). Our results show that the time and energy estimation from our scheduling algorithm are accurate with 1.48% error at maximum in overall benchmarks. In our experiments, we populated the time and energy values per region by only one program execution before performing the scheduling.

In Figure 4.4, we finally show the estimated minimum energy over time for all the benchmarks. We get these plots by applying our scheduling algorithm repeatedly with a halved time delay until the same schedule is generated. The results show that the minimum bound of energy consumption we can achieve, given regions.

4.3.2 Discussion

In our approach, the region partitioning is a critical concern to achieve near-optimal scheduling. This is not straightforward but we have guidelines to figure out it effectively. First of all, we know the lower bound of execution time in each region. The time should be much greater than the maximum value of p-state transition overheads. Otherwise, our time estimation will be dominated by the transition overheads, which prevents our scheduling algorithm from estimating minimal energy consumption. Second, we also have the upper bound in the number of regions. In reality, we do not need to define large number of regions for the scheduling. Again, note that more regions may cause more p-state transition overheads.

In this chapter, we assume that target programs are single processor applications for now. As future work, we will address this problem on multiple processor programs with mutual communications, such as MPI programs. Extending our model and estimation to deal with parallel programs requires more complicated analysis because the programs in parallel systems often use collective operations to synchronize themselves at some points. The p-state transition in one node may cause collateral delay in other nodes, which do not guarantee to finish the job within given

allowable time delay. Thus, the state space tree has much more nodes to be considered.

4.4 Summary

In this chapter, we proposed a way to achieve the minimum energy consumption by applying voltage and frequency scaling on the fly. The basic idea is to build the time and energy estimation models, given regions in programs. Then, we designed an efficient scheduling algorithm to compose the best p-state transitions for all regions while satisfying a given time delay. Results on the NAS serial benchmark suite showed that the running time can be reduced significantly with our algorithm. Our time and energy estimations are highly accurate with 1.48% of differences at maximum, compared by real measurements.

Our future plans include designing an automatic region finding method so that we determine the minimum energy consumption efficiently. Additionally, we will extend this research on parallel programs. In this case, we have much more complexity not only because of the number of nodes, but also because of the communications each other. The ultimate goal is to design and implement a system which efficiently figure out the energy saving potential by DVFS for all programs.

Chapter 5

Fine-grained and Accurate Power Measurements

Energy costs in a computer system have dramatically increased in the last decade. Most of this cost is power. To increase the performance of microprocessors, vendors have produced systems that require more power to run and cool [29, 74]. To reduce the cost of these systems, the power (or energy) consumption has become one of the major design concerns for modern hardware and software systems.

One important component in any detailed analysis and decision process to improve the power efficiency or lower consumption is the ability to monitor actual power consumption. Previous work with both the hardware and software has attempted to reduce power consumption without decreasing performance [65, 66, 30] by measuring the power consumption of the whole system. With this approach, an external power meter is used, such as *Power Egg* [75] or one of the Watts up? products [76]. A single value that aggregates all power for the system, this includes processors, memories, disk drives, networking, power loss in the power supply, etc. is obtained at some maximum rate defined by the external monitor.

An external power meter has limitations, which impact the design and the evaluation of various power reduction methods. The precision of external power meters is limited and reading the values entails considerable latency. They also do not provide separate power values for each component in a system. The latency to access the power values tends to be large in most external

The research in this chapter was supported in part by PERI - Performance Engineering Research Center (PERC-3) DE-FC02-06ER25764 and RENCI award.

meters. This limits the maximum number of measurements made in a given time. The values need to be averaged over the time between the accesses. This averaging smooths out the measurements so small duration peaks and valleys are not visible in the data. Each component has its individual power consumption, which changes depending on application characteristics. With external power monitors, there is no way to measure power at the component level. When making hardware and software changes to the individual components, using external meters prevents the direct measurement of any change.

The RENCi PowerMon was built by RENCi staff to provide accurate, fine-grained power measurements for any commodity computing system. It is designed to provide fine-grained monitoring by device and to characterize static power, e.g. power supply overhead. It is expected to assist projects that attempt to reduce power and energy consumption by providing accurate measurements of their effectiveness and to help projects that have a need to observe power efficiency.

The PowerMon connects inside a system to a standard 24-pin ATX power supply. It uses a standard USB communication interface and a simple API to communicate with the outside world. Drivers for Linux, Windows, and MacOS are available to collect the readings from the device. Our prototype separates the power cable into six different power rails and can read the values detected by different sensors. We refer to a power rail as a sensor.

In this chapter, we focus on the evaluation of our first prototype. First, we compare the access latencies of the PowerMon vs. an external power meter, called *Power Egg*. The PowerMon can measure power consumption at better than 5 times the rate of the external meter (2 Hertz) when reading all the sensors. When only 1 power sensor is read, it can be read at about 50 Hertz. This increased measurement rate allows a more accurate view of the power utilization to be gathered. Using several simple test programs, the fine-grained power measurement allows the individual component's power usage to be distinguished. Once identified, we show how power usage varies over time even within simple benchmarks. When attached the power overhead of monitoring is relatively small, on average about 0.44 W. Using both use the PowerMon and an external monitoring device simultaneously, the power supply efficiency and hard disk power usage can be computed.

In Section 5.1 we describe the RENCi PowerMon, our prototype power monitor, including its specification and supporting software tools. The evaluation section, 5.2, explores some of the possible uses of the PowerMon on a desktop system. Finally, we summarize our work and describe revisions for the next version of PowerMon in Section 5.3.

5.1 Design

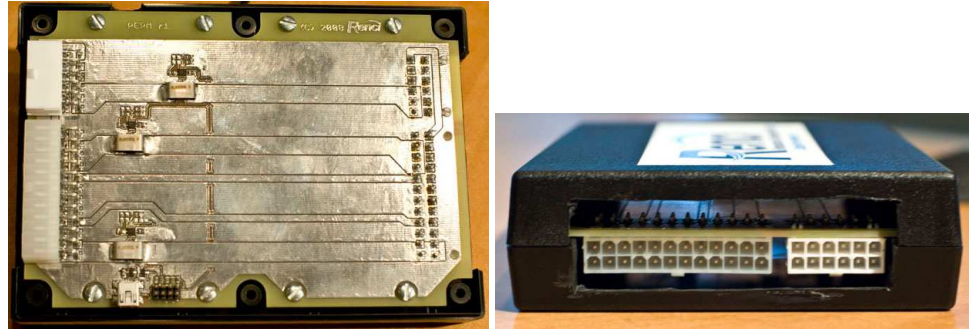


Figure 5.1: PowerMon prototype board

The RENCI PowerMon is an inexpensive measurement device designed to monitor individual DC power rails provided to a PC motherboard by a standard AC/DC power supply. It supplies power consumption values to the outside world through a USB connection.

The device consists of six Analog Devices ADM1191 [77] digital power monitor ICs and an Atmel ATtiny45 [78] micro-controller. The power monitor ICs amplify a voltage differential across a precision 5 milliohm resistor. The amplified voltage is digitized on-chip using a 12-bit ADC, and the digitized values are accessed via an integrated I^2C transceiver. The micro controller is programmed with a modified version of Till Harbaum's i2c-tiny-usb [79] firmware. The i2c-tiny-usb implements the appropriate timing and logic to control the board's I^2C bus via USB. The data from all individual sensors is accessed through USB protocol. An upgraded prototype with a more capable micro-controller with a hardware USB interface is capable of collecting 500 complete sets per second, or over a thousand sets if only the currents are collected.

The PowerMon basically intercepts all the power cables between power supply and motherboard. The cables are first connected to PowerMon internal meter and then the jumper cables from the PowerMon are linked to the motherboard. It does not measure the power consumption of any component that is directly connected to power supply, such as hard disk. For some systems, it would not catch the power consumption for cooling fans drawing power directly from the power supply. The systems examined to date drive their fans from the motherboard.

Driver

A driver for the i2c-tiny-usb interface is built into the Linux kernel in versions 2.6.22 and above. A kernel module is available for earlier versions. With the driver installed, the operating system can interface directly to the I^2C bus and each of the ADM1191 ICs.

```
> sudo ./pepm_read
address, voltage (V), current (A), power (W)
0x30, 3.276152, 1.793285, 5.875075
0x31, 5.089043, 0.909563, 4.628803
0x32, 3.315000, 4.428949, 14.681967
0x33, 5.121416, 1.235145, 6.325689
0x34, 12.139893, 0.180879, 2.195850
> █
```

Figure 5.2: A reading from different sensors

A sample utility was created to demonstrate the process of collecting sensor readings (Figure 5.2). The application iterates through available I^2C buses until the i2c-tiny-usb device is found. The application then queries each of the sensor ICs for a combined voltage and current reading (three bytes total). The received values from the ICs' analog-to-digital converters are converted to voltage and current using the formula available from the manufacturer's datasheet. Besides the kernel driver, PowerMon program requires the I^2C Tools for Linux from the lm-sensors project [80].

A monitoring daemon

We develop a monitoring server application running on a system connected to the PowerMon by USB cable. It runs as a daemon process accessing to the meter continuously and serving a client request. Each time the PowerMon is accessed, the voltage and current are returned for the specified power sensor. To compute the power usage and the energy consumption over time, the monitoring process continuously collects the power usage at the maximum sampling rate. The process also serves the latest measurement locally or remotely over TCP protocol concurrently. Energy consumption is computed by numerical integration. The client can ask the power or energy values with different command option.

5.2 Evaluation

To understand the performance of the prototype RENCi PowerMon, it is attached to an HP Pavilion with a 2.3 GHz AMD Phenom processor and 4GBs of DDR2 memory. Functionality, performance and overheads are compared against a standard external monitor, the *Power Egg* by WatchDog.com. As with most recent commodity machines, our system has two power cables between the power supply and the motherboard: a standard 24-pin connector and an additional 4-pin connector dedicated to CPU. Many systems use additional 8-pin connector instead of the 4-pin. By changing the set of jumpers as needed, the PowerMon also supports both setups.

The prototype PowerMon is tested to determine its maximum response rate. This rate is important for any measurement or tool that attempts to understand fine-grained performance and power fluctuations. With this knowledge, a series of micro-benchmarks is used to determine which components use which power sensors. After identifying the components of interest, several of NAS NPB benchmarks are run on the system and the results are reported.

Table 5.1: Comparison of PowerMon and Power Egg access latencies. Power Egg requires at least 480 ms between requests.

	Time (ms)	Required delay between measurements (ms)	Maximum number of measurement per second
One sensor	17.004		
All sensors(6)	102.007	0	9.804
Power Egg	173.942	480	1.529

5.2.1 Access latency and accuracy

Average response time measured for both power meters, as shown in Table 5.1. The external power meter, *Power Egg* is connected via RS-232 serial connector and the PowerMon via USB connection. Since the prototype PowerMon only allows access to one sensor at a time, the access time for both cases is shown. To compute total power consumed by the motherboard all sensors need to be measured, but the measurement of individual components can be performed at higher rate. The time to access a sensor does not depend on which sensor is being checked. The PowerMon takes about 17ms for each request or about 102ms for all the sensors, or about 9.8Hz. Although a *Power Egg* request takes about 174ms to return, it will not compute new values for 480ms after the request limiting performance to about 1.5Hz. or 1/6th of the PowerMon prototype.

The difference of power consumptions of the PowerMon is compared with the *Power Egg*

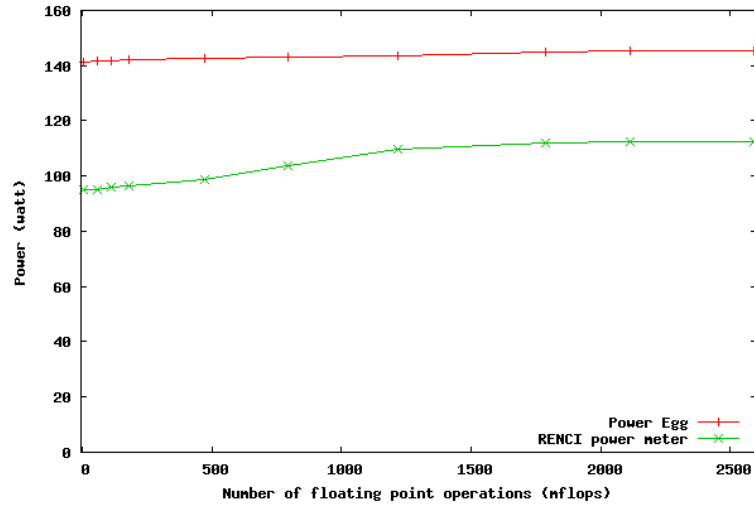


Figure 5.3: Accuracy comparison of internal and external power meters.

in Figure 5.3. Using a micro-benchmark with an adjustable floating point operation rate, power utilization is measured with both monitors independently. The two monitors were evaluated independently to prevent any interactions between the meters. Figure 5.3 shows the power values collected by each monitor at the monitor’s maximum rate. *perfmon2* performance module using the hardware performance counters is used to calculate the average MFlops of the runs. The two lines show the power as measured by the PowerMon and the *Power Egg* in Watts. The benchmark was run 13 times and the median value reported. To minimize the measurement overhead, the benchmark ran for 2 minutes. PowerMon shows a much larger range between the floating point unit being nearly off and being saturated. The PowerMon measures difference is 17.52 Watts, as the Mflop rate rose from 5 the 2592. By contrast, the *Power Egg* only records a difference of 4.24 Watts over the same range. This is because the power consumption from the *Power Egg* includes the power consumptions of power supply and hard disk, which does not necessarily increase by the number of floating point operations.

Since the PowerMon does not monitor the power used by the power supply or hard drives, the *Power Egg* line is higher. The range of entire system power usages is smaller as floating point activity is increased. We believe that much of the difference is a result of unmodeled changes

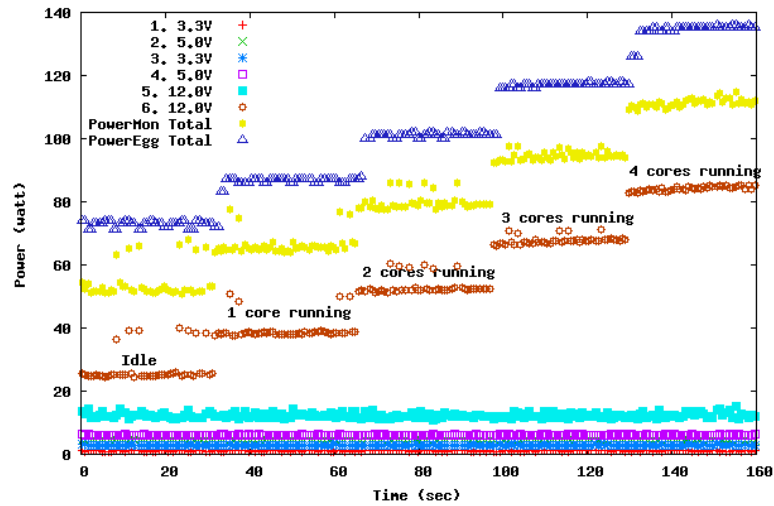
in power consumption of hard disk and power supply, as well as the decreased precision of the *Power Egg*. The rise in power supply efficiency hides the increase in power actually consumed by the motherboard from the external meter. This is a result of where the measurement is occurring, external meters must sit outside of the power supply and are vulnerable to any changes in power utilization by the power supply itself.

5.2.2 Fine-grained power measurement

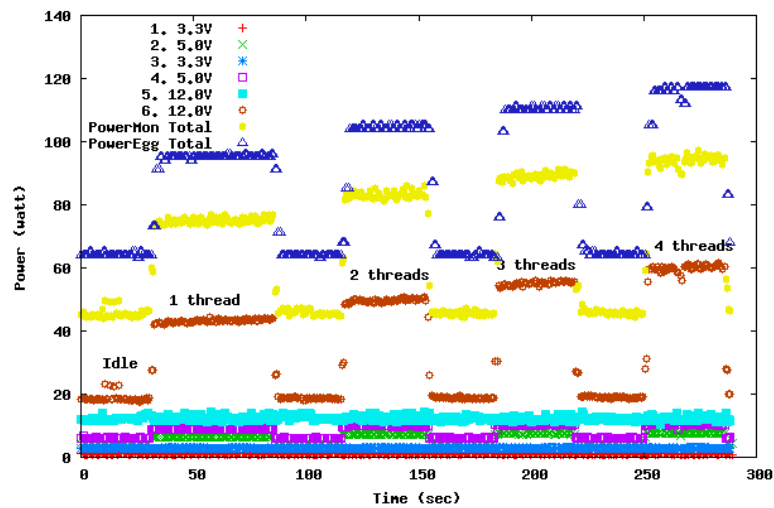
The RENCi PowerMon has 6 individual sensors, each responsible for measuring a different power sensor from the power supply. Which hardware component uses which power sensor must be determined for each system. This can be done by checking the motherboard documentation or by empirical tests, i.e. running benchmarks and examining what changes. Figures 5.4, 5.5, and 5.6, show various power measurements from the *Power Egg*, all 6 PowerMon sensors and the calculated total using the PowerMon, while running two simple benchmarks (infinite loop (`while(1){}`) and STREAM [81]).

Figure 5.4(a) shows the transitions of power consumption, as more cores are active running CPU intensive (but no floating point operation or off-chip memory accesses) threads. Every 30 seconds an additional core starts to run the program on an idle core. As we can see there are, as expected 5 power level steps one for each core count. Only one (red - sensor 6) of the sensors increases value as the core count increases. For this test, the other sensors saw almost no change. Sensor 6 reacts to the power utilized by the AMD Phenom processor. With the PowerMon, several one time anomalies in the power utilized are observed. We believe these to be caused by various Linux daemon processes running on other cores for short periods of time. These fluctuations are lost looking at the *Power Egg* results (blue triangles), because of its low monitoring rate and smoothing effects.

Figure 5.4(b) shows the power monitoring results of running STREAM-OMP benchmark [81] which measures effective memory bandwidth. As the number of threads increases from 1 to 4, each thread runs on a different core and the program used more power. The largest power changes occur on sensor 6 ('CPU'), but we also see sensors 2 and 4 reacting to the increase memory utilization. As we reduce the number of physical memory DIMMS, the power utilized along sensors 2 and 4 fell. For this system, these 2 sensors are shared by the memory modules. The CPU power consumption is not as high as measured in Figure 5.4(a). This is expected, since the STREAM is often waiting for memory requests to complete.



(a) Running infinite loops



(b) Running STREAM benchmark

Figure 5.4: PowerMon and *Power Egg* measurements with different active thread/core counts

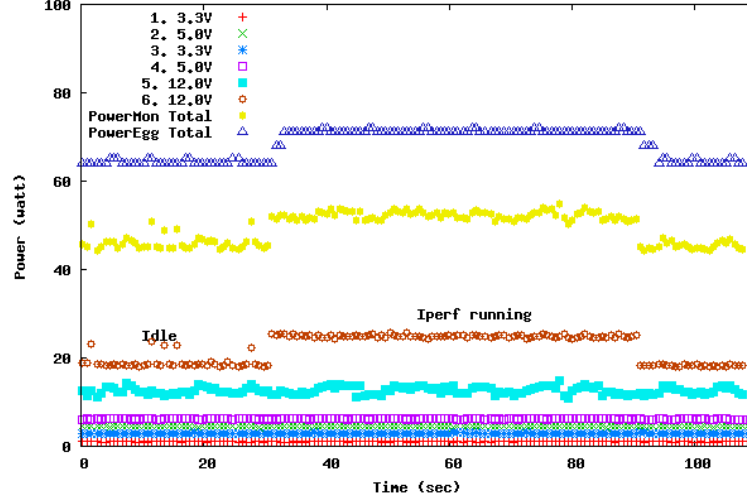


Figure 5.5: PowerMon and *Power Egg* measurements running Iperf benchmark.

We attempt to isolate the Network Interface Chip power, in Figure 5.5, by executing the TCP bandwidth measurement benchmark, called Iperf [82]. There is no significant variation in any sensor other than the 3 previously identified, for the CPU and memory modules. This confirms that the power required by a NIC does not tend to depend on the number of bits transmitted, just maintaining the connection requires the NIC to be active which is the majority of the power used by the NIC. When it is active, it consumes consistent power by continually sending and receiving bits. This is consistent with Feng, et al. [27].

Next, we examine the power utilized by various PCI cards installed on the system. The HP Pavilion had two installed PCI cards, one for VGA output and one for CATV. Four configurations were tested with them: neither installed, only VGA installed, only CATV installed and both installed.

Figure 5.6 shows that the CPU and memory power usage is consistent (sensors 2, 4, and 6 over all configurations. The Power consumptions from sensors 3 and 5 change depending on the different PCI configurations. These two sensors clearly supply power for the PCI bus, with the 12.0V bus (sensor 5) dominating the 3.3V bus (sensor 3). Together, in idle state, the two PCI buses account for 34.2% of motherboard power usage and 24.2% for entire system power usage. Clearly,

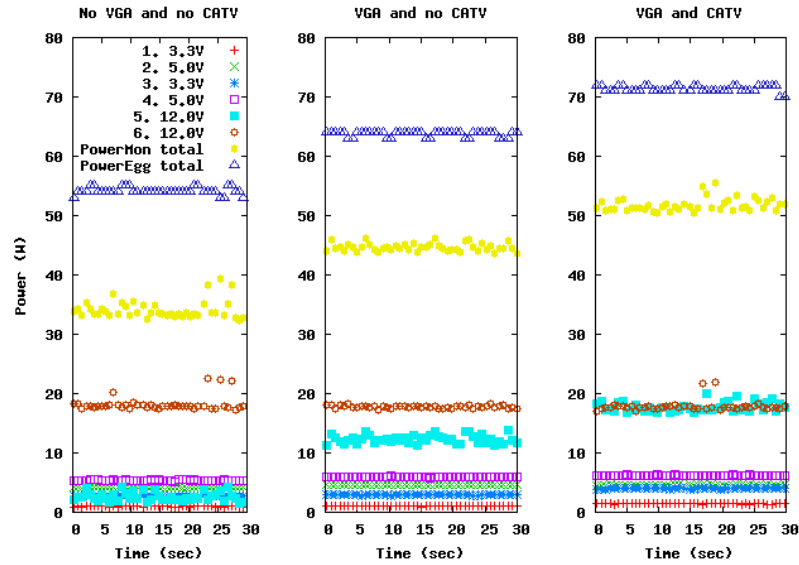


Figure 5.6: PowerMon and *Power Egg* measurements with different PCI cards installed.

controlling power utilized by the PCI bus when the system is in idle state is significant.

Figure 5.7 compares hard drive energy usage under changing system loads. The no-disk drive configuration is accomplished by booting an Ubuntu Linux kernel from a USB flash drive and running CPU intensive while-loop program to increase system load. The PowerMon, as expected, shows no difference in power depending on whether the disk drive was installed. The *Power Egg* sees an increase in the power used by the system between the two configurations. Table 5.2 presents the measurements for both cases. A hard disk installed increased the external power usage between 5.212W and 7.325W depending on system load. This is consistent with what has been observed by

Table 5.2: Measured hard disk power consumption

	<i>Power Egg</i> & PowerMon diff with disk (W)	<i>Power Egg</i> & PowerMon diff with no disk (W)	Derived disk power (W)
Idle CPU	19.369	14.157	5.212
1 core running infinite loop	21.438	13.553	7.885
2 cores running infinite loop	20.924	14.322	6.603
3 cores running infinite loop	22.248	14.898	7.350
4 cores running infinite loop	23.392	16.067	7.325

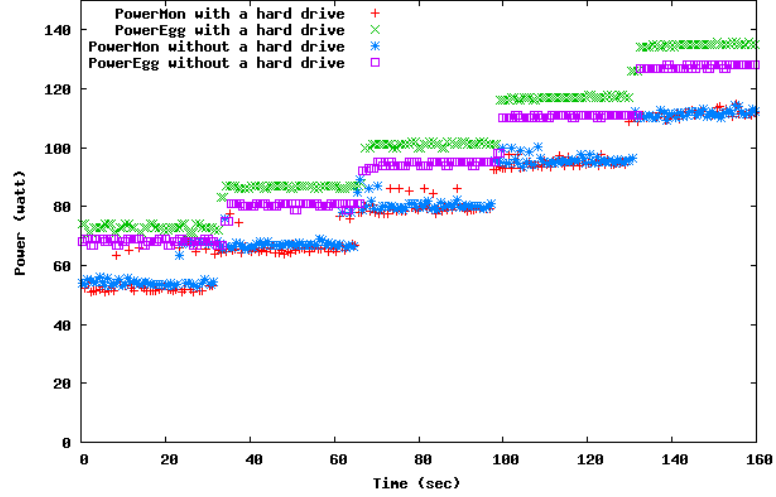


Figure 5.7: PowerMon and *Power Egg* power measurements with/without a hard drive.

others [27, 83].

5.2.3 Power monitoring overhead

One important aspect of using the PowerMon understands its overheads. By using a separate system for monitoring, the performance overheads have been eliminated on all of these tests. The power overheads are not eliminated. Figure 5.8 shows the increase in power usage (externally measured with the *Power Egg*) from the presence of and frequency of monitoring with the PowerMon. The smaller time intervals indicate more frequent accesses to the PowerMon by the monitoring system. When the PowerMon is not connected, an idling system consumes 78.02W. The power consumption increases only about 0.06W when the PowerMon is connected but not used. Monitoring 5 times a second, results in a total power increase of only .41W. If the monitoring is continuous ($\tilde{10}\text{HZ}$), the power overhead rises to 1.79W (about 2% of total system power). At the maximum sampling rate, the CPU time takes up to 1.5 % in one core.

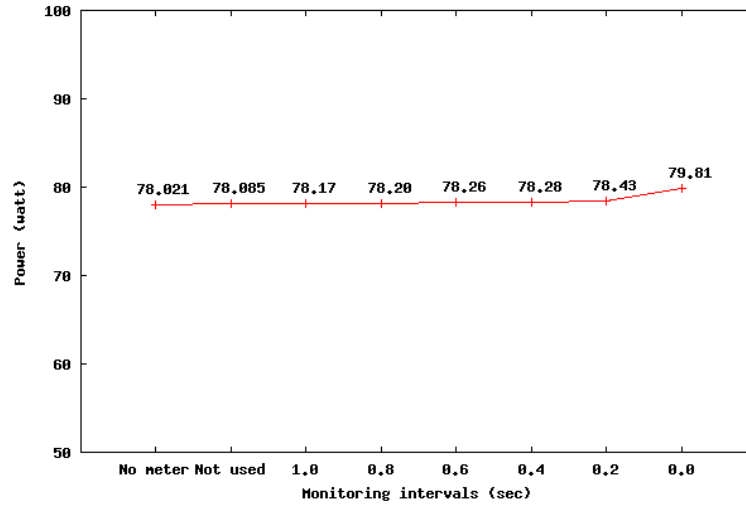


Figure 5.8: Power consumption over different monitoring intervals.

Table 5.3: Power supply efficiency by CPU loads on a quad-core AMD Phenom machine

	Input power (W) (external power)	Output power (W) (internal power)	Power supply overhead (W)	Power supply efficiency (%)
Idle CPU	105.71	72.61	33.10	68.69
1 infinite loop	131.62	91.63	39.99	69.62
2 infinite loops	150.30	105.41	44.89	70.14
3 infinite loops	176.02	121.76	54.26	69.17
4 infinite loops	204.69	140.67	64.02	68.72

5.2.4 Power supply efficiency

One of the driving questions motivating PowerMon is to understand the effect of power supply efficiency. The efficiency is defined by dividing the output power of power supply by its input power. The information from external power meters is limited by the fact that they must measure the inputs to power supply. We use PowerMon to obtain the outputs of power supply.

It is known that the efficiency varies by power usage from a power supply [84]. Also, different power supplies may have different characteristics in efficiency. The efficiency of power supply can hide other changes in the system. In CPU power utilization, the difference between two chips can be masked by differences in the power supplies for the systems. Therefore, characterizing

Table 5.4: Power supply efficiency by CPU loads on a quad-core Intel Nehalem machine

	Input power (W) (external power)	Output power (W) (internal power)	Power supply overhead (W)	Power supply efficiency (%)
Idle CPU	142.79	97.96	44.83	68.6
1 infinite loop	173.99	125.19	48.80	71.95
2 infinite loops	189.54	138.85	50.69	73.26
3 infinite loops	206.49	154.93	51.56	75.03
4 infinite loops	224.16	168.20	55.96	75.04
5 infinite loops	229.80	172.44	57.36	75.04
6 infinite loops	234.97	176.87	58.10	75.27
7 infinite loops	240.17	183.12	57.05	76.25
8 infinite loops	245.12	185.91	59.21	75.84

power supply efficiency is essential to understand overall power behaviors in a system.

In this section, we evaluate the power supply efficiencies from two different machines: a quad-core AMD Phenom machine and a quad-core Intel Nehalem machine with Hyperthreading technology (8 effective cores). The machines have 350W and 700W power supplies, respectively. In order to obtain the input power of power supply, we use the WattsUp PRO power meter. The output power of power supply is measured by PowerMon. We measure the efficiency over varying CPU loads. Adding a CPU intensive while-loop to each CPU core makes a significant change in power consumption because the core running the loop goes into an active mode from a low-power mode.

In Table 5.3, we show the power supply efficiencies measured in different CPU loads. The power consumption of power supply significantly increases from 33W to 64W as the number of the loops increases. Due to the increase of internal power consumption, however, the power supply efficiency changes only about 2%. The same evaluation is made on a quad-core Intel Nehalem machine as shown in Table 5.4. In this machine, each physical core is shared to run two threads simultaneously due to Hyperthreading technology. The internal power difference up to 4 loops is more than 12W but there is only about 4W difference from 5 to 8 loops. This is because we run the first 4 loops on different CPU cores. So, the CPU power consumption of 2 cores is relatively bigger than the power when running 2 loops on one core. The power consumption of power supply increases from 44W to 59W as the CPU loads increases. The power supply efficiency increases about 7%, which is bigger than the efficiency change of the Phenom machine. The efficiency difference is more noticeable when we add up to 3 infinite loops at the beginning. Then, the efficiency remains

around 76%. The study [84] shows that the power supply efficiency increases and then decreases as the power load grows. Our results show the similar curve in that the efficiency increases at low power loads and decreases at high power loads. But there is no significant difference at high loads although the power consumption of power supply continue to increase. In our evaluations, however, we could not test a full range of power usages in power supply because the power consumption of each component in a system is limited.

5.3 Summary

In this chapter, the RECNI PowerMon has been described, including both the hardware design and the software tools to allow internal power measurements on standard desktop systems available today. We show that the power of individual components can be separately measured from six power sensors multiple times a second. The accuracy and overhead were evaluated and compared with an external power meter. By reading the external meter at the same time, the power supply efficiency is derived and the hard disk power consumption computed.

Our plan is to use the PowerMon to profile the power consumption in parallel and distributed systems and to analyze the power usage patterns of different components in a system by workloads. In Chapter 6, we model the system power consumption using hardware performance counters. The accurate and fine-grained monitoring features in PowerMon supports the accuracy of our prediction model.

Chapter 6

Power Prediction on Multi-core system via Performance Counters

In recent computing systems, the analysis and/or the optimization of power/energy consumption becomes one of the critical concerns [85, 86, 87]. Many computing systems including supercomputing and clustering machines, support real time power measurements via either internal or external power monitoring devices. Although these dedicated power monitoring devices support fine-grained, accurate power measurements, there is a significant delay required to communicate with them at runtime. This communication delay may cause unnecessary power/performance penalty when dynamic decisions are made. In addition, measuring the power consumption separately on hundreds or thousands compute nodes is troublesome for a large cluster. Therefore, we figure out how to estimate the power consumption with performance data observed in a system. Since the power is proportional to the utilization of system resources and the utilization can be measured by performance monitoring, the power consumption can be derived by the performance data with reasonable accuracy.

This chapter presents how to establish our power prediction mechanism using performance data on multi-core architectures and evaluates how accurate it is.

The research in this chapter was supported in part by PERI - Performance Engineering Research Center (PERC-3) DE-FC02-06ER25764 and RENCI award.

6.1 Approach

In this section, we describe our detailed approach regarding how to build a prediction model. Our approach mainly consists of the 4 steps: profiling power and performance data, selecting representative performance counters, establishing a prediction model, and validating the accuracy of our prediction model. We describe each step in the following subsections.

6.1.1 Measuring power and performance

For hardware performance data, we collect the performance counters from Linux system with the **Perfmon2** [88] package. The application programming interface of Perfmon2, called *libpfm*, is used to gather the performance data per code region at application runtime.

For power profiling, we use an internal power monitoring device, **PowerMon** described in Chapter 5. The PowerMon is an internal power meter that measures power consumption of six different power rails between power supply and motherboard. The voltage and current, and therefore the power consumption, are measured for each power rail and can be used to calculate power for specific components. It provides fine-grained and accurate power measurements for any commodity computing system.

The PowerMon measures total power consumption of all the components in a system but the power supply. Since the power consumption of power supply is directly correlated with the internal power consumption, we can easily incorporate it into our model.

6.1.2 Profiling power and performance

For a better accuracy in power prediction, it is important to profile the power and performance data with fine granularity in execution time. In many applications, the power and performance characteristics vary by code regions at runtime. For example, there are often many distinct parallel regions in HPC applications, such as computation and I/O. Therefore, we divide the applications into several code regions and profile the data per region. Identifying code regions may be different by applications but we target for parallel applications to evaluate the scalability of our model. In OpenMP [89] applications, determining parallel regions is straightforward by find the OpenMP parallel directive, such as *OMP PARALLEL*.

6.1.3 Selection of performance counters

First of all, we determine the representative performance counters among available counters. The number of representative counters is limited by underlying system architecture. For example, AMD Athlon or Opteron architectures support monitoring up to 4 hardware performance counters at given time. Pentium 4 architecture supports monitoring up to 18 counters at the same time. In general, more performance data help to build a better power prediction model because the power is consumed by many components in a system. In this chapter, we only use 4 performance metrics because our machine with a quad-core Phenom processor supports monitoring at maximum 4 counters concurrently. According to our evaluation, however, we could achieve a better goodness of fit of our prediction model by adding more metrics in our regression because more information can provide a better prediction model in general.

In order to determine the best performance counters for our model, we use Spearman's rank correlation method [90], which describes the relationship between two variables without making any assumption about the relationship. After correlating each performance counter with power consumption, we can determine the representative performance counters by ordering them. Given a set of n data set denoted by $\{(x_i, y_i)\}_{i=1}^n$, the Spearman's rank correlation coefficient (ρ) is calculated in Equation (6.1).

$$\rho = 1 - \frac{6 \sum (x_i - y_i)^2}{n(n^2 - 1)} \quad (6.1)$$

In Table 6.1, the Spearman's rank correlations of performance counters with internal power consumption are shown in order. Note that the correlation orders are noticeably different depending on different correlated power consumptions. For example, when the power rail is directly associated with memory, the performance counters related to memory performance, such as MEMORY_CONTROLLER_REQUESTS:ALL or CPU_IO_REQUESTS_TO_MEMORY_IO:ALL are on top in the list. However, the memory power consumption (approximately, one tenth of CPU power at maximum) is relatively small compared with CPU power consumption in our system. We focus on correlation information with CPU or internal power consumption.

6.1.4 Applying multivariate linear regression

In this chapter, we use a multivariate linear regression method to build a power prediction model. Suppose we have a set of n observations denoted by $\{y_i, x_{i1}, \dots, x_{ip}\}_{i=1}^n$, where y_i is the dependent variable and x_i is p -vector of predictor variables. Let $\beta = \{\beta_1, \beta_2, \dots, \beta_p\}$ denote the

regression coefficients corresponding to the predictor variable. In our model, we assume that the relationship between the dependent variable and the predictor variable is approximately linear. This approximated linear relationship is modeled through a random error, ϵ_i , which is an independent and unobserved random variable. Our multivariate linear regression is formulated in Equation (6.2).

$$y_i = \beta_1 x_{i1} + \beta_2 x_{i2} + \dots + \beta_p x_{ip} + \epsilon_i, i = 1, \dots, n \quad (6.2)$$

Finding the regression model fitted to the observed data set indicates the determination of the regression coefficients, β_i . The *least square fitting* method is a commonly used estimator to analyze the given data set. It basically identifies the best-fitting model by minimizing the sum of squared deviations of predicted and observed values as formulated in Equation (6.3):

$$S(\beta) = \sum_{i=1}^n \left(y_i - \sum_{j=1}^p \beta_j x_{ij} \right)^2 \quad (6.3)$$

It is not known that there is a general and invariable relationship between power and performance counter numbers. The relationship may also depend on which performance counters are chosen as a predictor variable. In prior work, Lee *et al.* [91] mixed linear and non-linear models but modeled a linear relationship for performance counters as used in our approach and Singh *et al.* [92] empirically established a piece-wise prediction model for power prediction. According to our evaluations, however, our model can achieve reasonable prediction accuracy. The linear model can be easily generated and applied when on-line prediction and adaptation is required. For non-linear prediction models, we will investigate more research on determining a better concrete relationship of power and performance counters in near future.

With our linear regression model, we make an analysis of the goodness of fit using the coefficient of determination to determine the best model [93]. It is denoted as R^2 :

$$\begin{aligned} SST &= \sum_i (y_i - \bar{y})^2 \\ SSM &= \sum_i (f(x_i) - \bar{y})^2 \\ R^2 &= \frac{SSM}{SST} \end{aligned}$$

where SST represents the prediction error when the average value $\bar{y}$ is used as an estimate for the response variable. On the other hand, SSM represents the amount of error in the regression model. Its value ranges between 0 and 1. The value is close to 1 if most of the variability observed in the response variable can be explained by the regression model. In our evaluations, we make three different groups of performance counters and show their coefficient of determinations. We pick the best performance group close to 1 and use it to generate a prediction model.

For the validation of our prediction model, we divided the power and performance profiling data into two sets: generation and test. The generation set is solely used to generate a power prediction model. The test set is used to verify the prediction accuracy of our model. Our test sets include different profiling data in three dimensions: core frequency, number of active cores, and application problem size. We report how well our model can predict the power consumption in different configurations.

6.2 Evaluation

To evaluate the performance of the power prediction model, we collect the performance data on a HP Pavilion with a 2.3 GHz AMD Phenom processor and 4GBs of DDR2 memory by running NAS NPB OMP and SPEComp benchmarks. The PowerMon is used to measure the internal power consumption every 100 millisecond.

For the statistical analysis, we use the package R, a free software environment for statistical computing. All the statistical analysis described in Section 6.1 were automatically performed by this tool.

While running the benchmark, we first profiled the performance data and the power consumption. Note that we collect the performance and power data per code region to derive a better prediction model with fine-grained measurements. Then, we figure out their correlation for all code regions. The Table 6.1 shows the performance counters highly correlated with power consumption. All the performance counters in the Table have reasonably high correlation because they are directly related to rely on the CPU (including Cache) or Memory resource utilization.

Given the promising performance counters representing the correlation with power consumption well, we now determine a group of performance counters that establishes a power prediction model. As we mentioned, the number of performance counters in a group depends on an underlying architecture. Since we use a AMD Phenom processor for evaluation, only 4 performance

Table 6.1: Correlation of selected performance counters with internal power consumption

Correlation	Performance Counter
0.200	RETIRED_UOPS
0.165	DISPATCH_STALLS
0.091	MEMORY_CONTROLLER_REQUESTS:ALL
0.084	INSTRUCTION_CACHE_FETCHES
0.084	RETIRED_MMX_AND_FP_INSTRUCTIONS:ALL
0.074	DRAM_ACCESSES_PAGE:ALL
0.069	L3_CACHE_MISSES:ALL
0.060	RETIRED_INSTRUCTIONS
0.052	DATA_CACHE_ACCESSES
0.050	CPU_IO_REQUESTS_TO_MEMORY_IO:ALL

Table 6.2: Three selected performance counter groups and their coefficients of determination of multivariate regression

Group (R^2)	Performance Counter
1 (0.681)	RETIRED_UOPS
	DISPATCH_STALLS
	L2_CACHE_MISS:ALL
	RETIRED_MMX_AND_FP_INSTRUCTIONS:ALL
2 (0.619)	RETIRED_INSTRUCTIONS
	CPU_CLK_UNHALTED
	L2_CACHE_MISS:ALL
	MEMORY_CONTROLLER_REQUESTS:ALL
3 (0.800)	RETIRED_UOPS
	DISPATCH_STALLS
	DATA_CACHE_ACCESSES
	RETIRED_MMX_AND_FP_INSTRUCTIONS:ALL

counters can be measured at the same time. We may choose any combination of 4 performance counters among them to make a power prediction model. In Table 6.2, we show some of performance groups. These groups are selected by choosing performance counters measuring different types of resources in a system. The table shows their coefficients of determination of multi-variable regression to compare the goodness of the fit by the regression. As shown, the last group indicates its goodness of the fit outperforms the others. In other words, the prediction model using the last group is likely to predict the future power consumption well, compared to the others. It may be possible to construct a better prediction model either by different combination of performance counters or by increasing the number of counters.

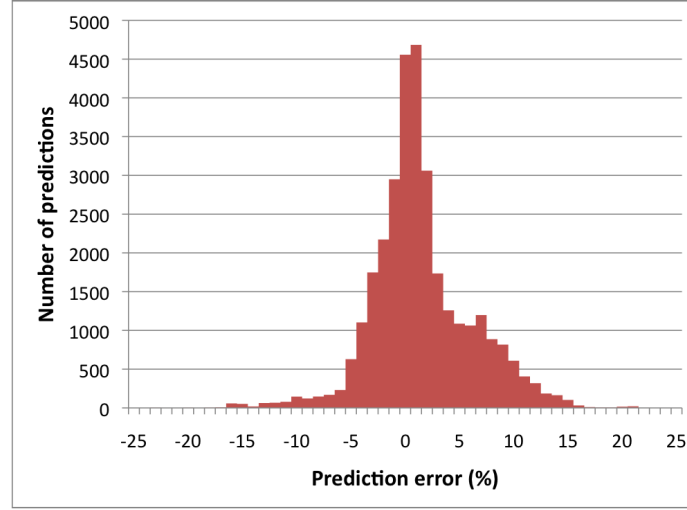


Figure 6.1: Distribution of prediction errors

For our evaluations, we use the group with the highest coefficient of determination in the table. In order to build a power prediction model, we used the performance and power data profiled while running the NAS NPB benchmark. The benchmark is executed on 4 threads, where a different thread runs on a different CPU core. In our profiled data, we also added the performance and power data of the system at idle state.

Then, we used multivariate linear regression with the profiled performance and power data to generate a power prediction model. In this process, more complicated regression method can be applied to derive more accurate prediction model, such as nonlinear regression or piecewise regression. We conclude that our prediction model is overall good enough to predict the power consumption with reduced complexity for a fast decision.

Given the prediction model by regression, we evaluate how accurate the prediction model is. The Figure 6.1 shows the distribution of prediction errors. From all the profiling data, we calculate the error every 500 millisecond, regardless of code properties. The distribution is obtained from about 34000 prediction evaluations. As shown in the figure, most of prediction errors are in the range of -5% and 5%. The positive error in the figure indicates under-prediction, that is, the predicted value is lower than the measured value. The mean value is about 3%, which means that our prediction model is accurate.

We then evaluate the prediction model for OpenMP applications. In this evaluation, we

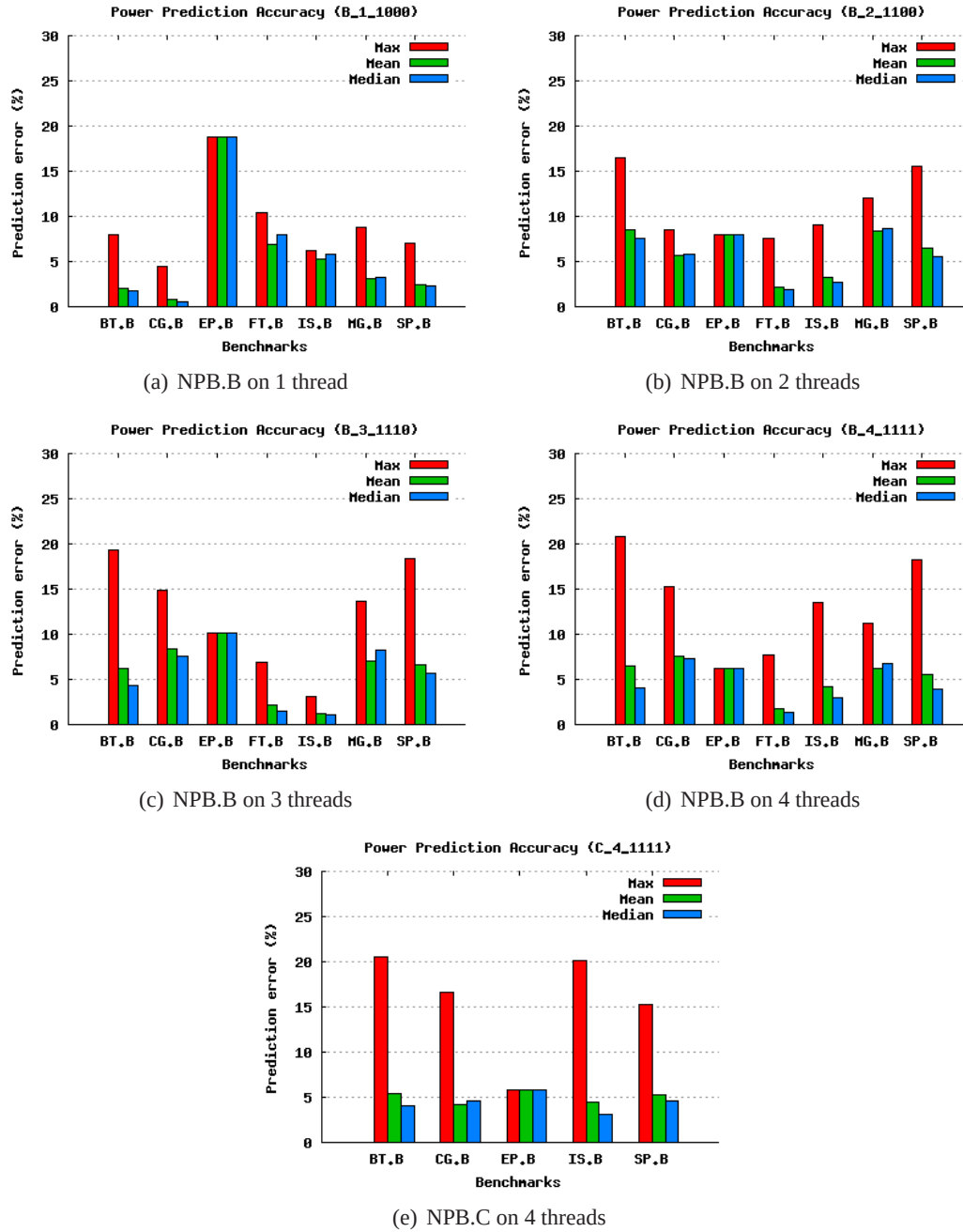


Figure 6.2: Power Prediction Accuracy for NPB benchmark

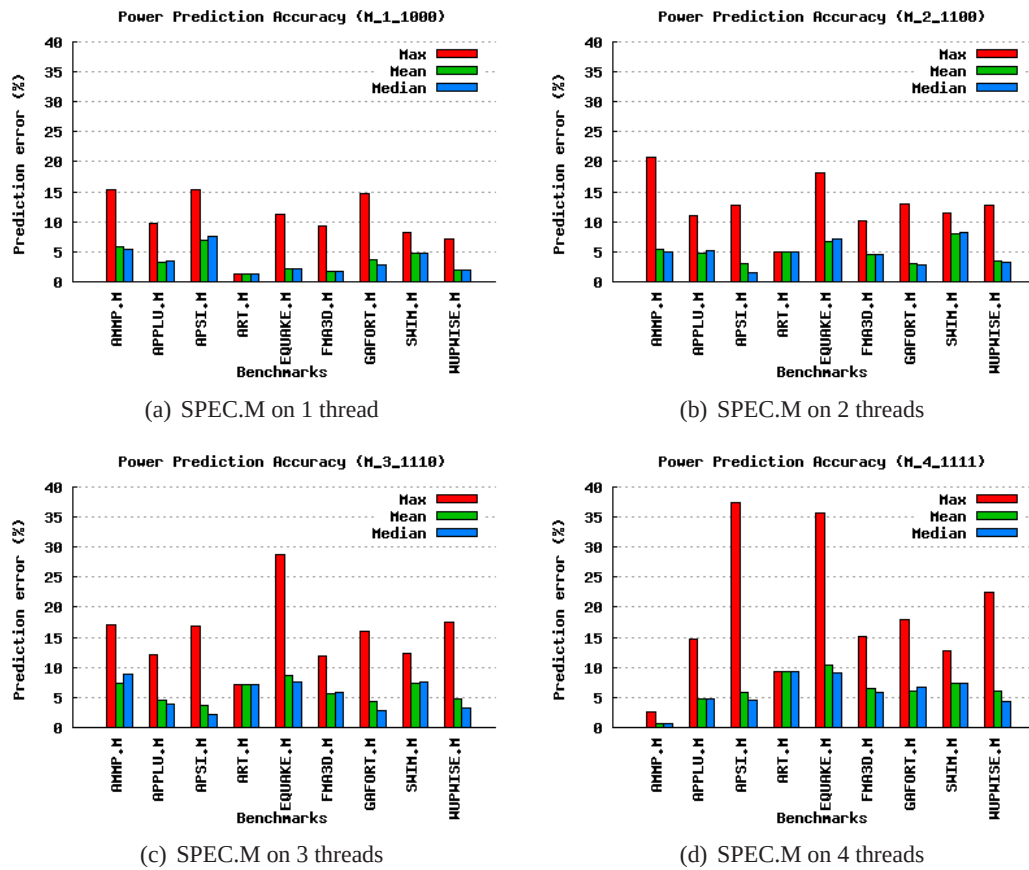


Figure 6.3: Power Prediction Accuracy for SPECOMP benchmark

show how accurately the power can be estimated in code regions. This test is conducted on different applications and execution configurations. The Figure 6.2 shows the prediction accuracy of the same NAS NPB benchmark with 5 different execution configurations. We run the benchmark by increasing the number of threads up to 4. On single thread execution, the p-state of the remaining 3 cores are scaled down to the lowest CPU frequency (1150 Mhz). On the other hand, each thread run on different CPU core on 4 thread execution. We also run it with the larger problem size (class C). We could not test some benchmarks in class C due to a compilation error by limited computing resources.

With the profiled data, we test the prediction accuracy as shown in each subfigure. The accuracy of prediction is measured by mean, median, and maximum values. Note that the mean and max values are same in EP benchmark because there is only one code region that takes long time in a single run. In all the other benchmarks, several hundreds or thousands of predictions are evaluated during single execution. As seen in the figures, the overall prediction accuracy have 5.6% error on average. In most of benchmarks, the mean error is less than 10%, except EP.B on 1 thread (18%) although some of predictions are as high as 20% that makes the average prediction worse. While the most of benchmarks achieve good power prediction, EP.B on 1 thread execution has the worst accuracy. According to our analysis, this misprediction is caused by over-prediction in which the prediction value is higher than the actual measurement. This is because the power consumption increases fast on low utilization and then become slow on high utilization. Therefore, our regression line stay higher than real measurement on 1 thread execution. As future work, we will further investigate this issue to derive more accurate prediction model.

In Figure 6.3, we evaluate our prediction model on different benchmark: SPEComp benchmark. We also profiled the power and performance data on 4 possible thread configurations. Although our prediction model is derived by the data from NAS NPB benchmark, the average prediction errors remain below 10%. Our analysis show that the worst predictions indicated by the maximum are most from the first or the end of code regions in our profiled data. We think that while executing these code regions, the system states are not stabilized, such as filling the caches or irregular resource usage patterns.

6.3 Summary

In this chapter, we developed a power-performance model that can correlate power and performance data on a chip multiprocessor machine. To establish an accurate prediction model, we profiled power and performance data of parallel applications on multiple execution configurations and figured out their correlation with linear regression. According to the evaluations, we determined the best correlation using regression with profiled power consumption and performance counters of well-known HPC benchmarks. The results show that this power-performance correlation is valid for multiple different system configurations.

We plan to advance this research to predict power consumption in various code regions of an application and eventually to improve overall power (energy) efficiency. Our work will then focus on developing a runtime power-performance adaptation framework, which can dynamically tune power and performance behaviors.

Chapter 7

PADD: Power Aware Domain Distribution

Modern data centers usually have computing resources sized to handle expected peak demand. However, peak demand events are rare, and average demand is generally far less than peak. Although there is no systematic analysis of server utilization across wide range of data centers, a recent study estimates the average utilization to be between 5% and 20% [94]. Even in highly customized IT environments, it is hard to achieve 50% utilization. Therefore, a large number of computing resources are idle much of the time. This presents an opportunity for power savings if we can shut down or hibernate physical machines not needed for the current workload while still providing the required continuity of service. Fortunately, several modern virtualization systems [95, 96] provide the ability to migrate running virtual machines without interruption. Combining live virtual machine migration with the ability to power physical machines on and off turns the data center into a dynamically resizable pool of compute resources.

Powering servers on and off is a well-studied technique for *fluid* workloads such as transaction processing. Such workloads consist of many, short-lived, independent requests which are serviced by homogeneous software running on multiple physical systems. However, many environments do not run fluid workloads. Instead, they execute coarse-grain workloads, such as database, batch, hosting or business logic systems which do not use a single workload distributor or which require heterogeneous software. Increasingly, the performance requirements of such workloads are specified in terms of *service level agreements (SLAs)* which give the throughput requirements for

The research in this chapter was supported in part by an University Partnership award from IBM.

each application in the computing environment.

In this chapter, we use simulation to evaluate the power and performance effects of dynamic workload consolidation for such workloads in a virtualized environment. The workloads are separated into multiple *domains*. Data center power is reduced by packing multiple domains onto fewer physical nodes during periods of low utilization and expanding them back onto more systems when the usage rises. We call this coarse-grained packing scheme *Power-Aware Domain Distribution (PADD)*. In this chapter, we explore the energy savings and performance impact of running heterogeneous domains in terms of demand and SLA.

One relatively new environment that may benefit from the PADD approach is the *Utility Grids*, which are clusters that are time- and space-shared by clients who will run an arbitrary workload. Amazon’s Elastic Computing Cloud (EC2) [97], NCSU’s Virtual Computing Lab (VCL) [98], and IBM Blue Cloud [99] are examples of this kind of grid. A key feature is that unlike HPC or business grids, the workloads that will be executed are not necessarily run-to-completion or throughput-oriented, but rather a mix. Some domains may run HPC-style apps and basically have a demand of 100% for the entire runtime. Others may be throughput-oriented and have fluctuating demand. Finally, some may act as human interfaces (shell, VNC, etc.), and have mostly low levels of demand. Given that the heavy use of virtualization by such grids, we expect to see a power-savings benefit from using PADD’s domain packing.

The simulation tool used in this chapter simulates dynamic domain packing based on traces of real-world workloads. It predicts both the power saved and the performance impact of a using PADD with a particular set of parameters for a particular workload as represented by a trace. By using a variety of traces, we are able to assess how generally applicable PADD is. The simulation also allows us to study how sensitive the results are to particular parameters such as buffer sizes and the time that it takes to migrate a partition or turn off a machine.

Our key contributions in this research are:

1. We developed a dynamic algorithm for placing domains to minimize total energy while avoiding SLA violations. Its key feature is the use of a two-level, adaptive buffering scheme which reserves capacity. Our scheme absorbs workload surges to avoid SLA violations, but adapting the buffer sizes to the workload reduces the amount of energy wasted on the buffers.
2. We provide a trace-driven simulation study of the energy benefits and the performance effects

Xen [95] uses the term *domain*, which is used in this chapter as a general, conceptual term for ease of exposition. Other terms in common use are *virtual machine (VM)* and *partition*.

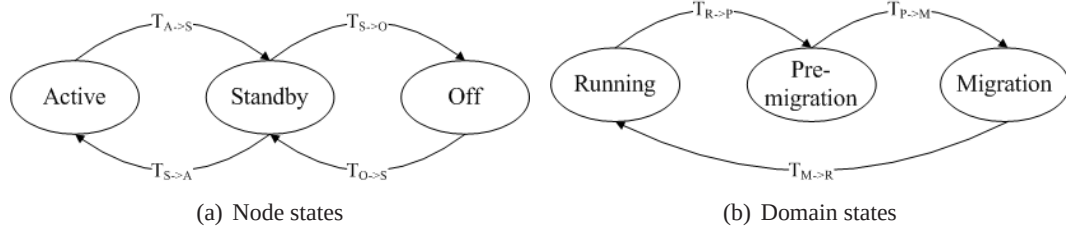


Figure 7.1: Node and Domain state diagrams. Each edge indicates the transition time delay.

of our algorithm along with studies of its sensitivity to various parameters.

In contrast to the prior work presented in Section 2.3, we study power management for a coarse-grained environment where the unit of work is a domain. We assume that domains have long lifetimes and contain state which must be maintained. Our approach is to migrate entire domains between nodes, instead of request distribution. This introduces two qualitative changes to the problem.

1. Unlike redirecting requests, there is a significant time and energy cost to migrating domains.
2. The number of domains does not change (or at least changes slowly) even though the computational demand of each domain may change quite rapidly.

The rest of this chapter is organized as follows. Section 7.1 presents our model of a computing environment with coarse-grain workloads, and Section 7.2 gives a precise description of the PADD algorithm, discusses the parameters affecting it and presents the basics of our simulation. Section 7.3 presents our simulation results while Section 7.4 suggests future work and concludes.

7.1 Model

This section offers some quantitative basis for our approach as well as defining the basic structure of the power and performance model used in our simulations.

7.1.1 Machine State and virtualization

PADD distributes domains onto the available physical nodes using dynamic migration and controls the number of physical nodes by turning them on and off. Alternatively, if the systems

Table 7.1: Parameters defined for PADD and its power and performance model

Domain parameters	
M	Number of domains
$D_i(t)$	CPU demand of domain i at time t , $1 \leq i \leq M$
DS_i	Domain state in $\{Running, Premigration, Migration\}$.
$T_{DS_i \rightarrow DS_j}$	Domain state transition time
Node parameters	
N	Number of physical machines
C_i	Processing capacity of machine i , $1 \leq i \leq N$
NS_i	A machine state in $\{Active, Standby, Off\}$.
$P_{NS_i}^i(U)$	Power consumption of machine i at U utilization in the state NS_i , $1 \leq i \leq N$
$T_{NS_i \rightarrow NS_j}$	Machine state transition time
Eff_i	Maximum power-performance efficiency of machine i , $1 \leq i \leq N$
Performance parameter	
$SLA_i(d, l)$	$d\%$ of demands of domain i , must be processed within l seconds. $1 \leq i \leq M$

support hibernation, entering and exiting hibernation state can be used to manage the number of active machines. Since our research aims at achieving power savings by packing multiple domains onto fewer physical nodes during periods of low demand, PADD tries to keep energy consumption as low as possible while meeting given performance requirements.

Running multiple domains with heterogeneous workloads may introduce additional overhead in the current virtualization technologies. The overhead mainly comes from sharing device accesses, expensive TLB misses, page faults, and memory handling [100]. Padala, *et al.* [101] have evaluated the overhead of the existing virtualization technologies. They conclude that the overhead depends heavily on the running applications, their resource utilization, and the virtualization mechanisms used. Since virtualization is widely accepted in the industry, we make no attempt to model it in our simulation.

7.1.2 Model description

We define the power and performance models used in our algorithm and its simulation. The diagrams in Figure 7.1 show the operating states and the transitions between them for nodes

and domains respectively. We first define the operating states and the transitions between them for nodes and domains. For a node, there are three (3) stable states: *Active (A)*, *Standby (S)*, and *Off (O)*. Real systems may define more states, but we abstract them and collapse them for simplicity. In many modern systems, there is a variety of intermediate power states between *Active* and *Off*. These intermediate states consume varying amounts of power and typically trade off power consumption to reduce the time to get back to the *Active* state. However, for simplicity, we only consider one intermediate state, *Standby*.

Nodes may make four (4) transitions between states – *Active* to *Standby* ($A \rightarrow S$), *Standby* to *Active* ($S \rightarrow A$), *Off* to *Standby* ($O \rightarrow S$), and *Standby* to *Off* ($S \rightarrow O$). The transition time is a hardware-dependent characteristic of the particular systems being used in the installation. One key question is how strongly the power and performance behavior of PADD depends on the transition times between operating states.

Each domain can be in one of three (3) operating states: *Running (R)*, *Premigration (P)*, or *Migration (M)*. We assume that domains are long-lived entities and do not consider transitions to an off state. A domain is in the *Premigration* state during the time that the system is preparing to move it to a different node by copying data from the current node to the new one. Clark [102] measured the overhead imposed by this state on a Xen platform running SPECWeb and found little measurable degradation, so our model assumes that performance is unaffected by premigration. However, a domain in the *Migration* state generally becomes out of service, and the downtime varies depending on workload. We assume that Migration stops the work on the source node and restarts it on the target resulting in a pause in processing. Our model assumes that *Migration* stops the work in the current node and restarts it in the target node, and it is accompanied by pause which causes a processing delay for current requests. The Xen measurements indicates that this delay is less than 1 second. Consequently, frequent migrations will result in significant performance degradation. We take this gap in processing into account in our simulation. We use our simulator to analyze the sensitivity of PADD to different migration times.

The capacity of a node is the processing done by a single CPU running at maximum frequency. The CPU frequency is fixed – we do not model DVFS and related techniques, which are largely orthogonal to this work. The power consumption is taken to be a function of the system utilization. Since DVFS can affect both the capacity and power consumption, we assume for simplicity that the operating frequency is fixed at nominal. Also, most environments are heterogeneous with systems with processors of varying capacities. In [103], the capacity is modeled as a multidimen-

sional vector of the available components (CPU, Memory, IO, *etc*). For simplicity, we assume that processing capacity can be normalized to a single value representing the CPU capacity. We hope to incorporate the insight of [103] in the future.

The performance requirement of a domain is specified by the service level agreement which is defined as $SLA(D, L)$. It specifies how much of the demand D (expressed as a percentage of the total) must be processed within a given time limit L (specified in seconds). Each domain may have its own SLA. At present, PADD constitutes a best effort to meet the given SLA, so violations are possible. In future work, we plan to bound the degree of SLA violation so that hard limits can be enforced.

Our problem is then to reduce energy consumption while offering the performance required by the SLAs. Finding the optimal domain distribution on every cycle is an example of a bin-packing problem, where the bin is a node and the bin's size is the node's capacity. This is an NP-hard problem [104]. The task is actually more nuanced than the abstract bin packing problem, because node transitions and domain migrations have time delays. Moreover, the number of active nodes is not fixed.

PADD is a heuristic method which approximates the optimal solution to the problem. Its key feature is that it uses two separate safety margins – per-node and whole system. We refer to these safety margins as *buffers*. The buffers are reserve capacity which allows PADD to deal with changing workloads and the latencies of the actions that it takes. Although the algorithm has a number of other parameters, the buffer sizes have the greatest impact. It turns out that the behavior of PADD is most sensitive to the amount of local buffering within each machine and the size of a global buffer of reserve capacity in the environment: the other parameters matter less. Our simulation offers a quantitative insight into our scheme and the parameters that we use to tune it.

7.2 Implementation

We define a number of policies affecting power and performance. Our simulation framework can parameterize these policies as a group of parameter settings. The policies can be categorized into three groups: buffering, time cost, and other decision policies.

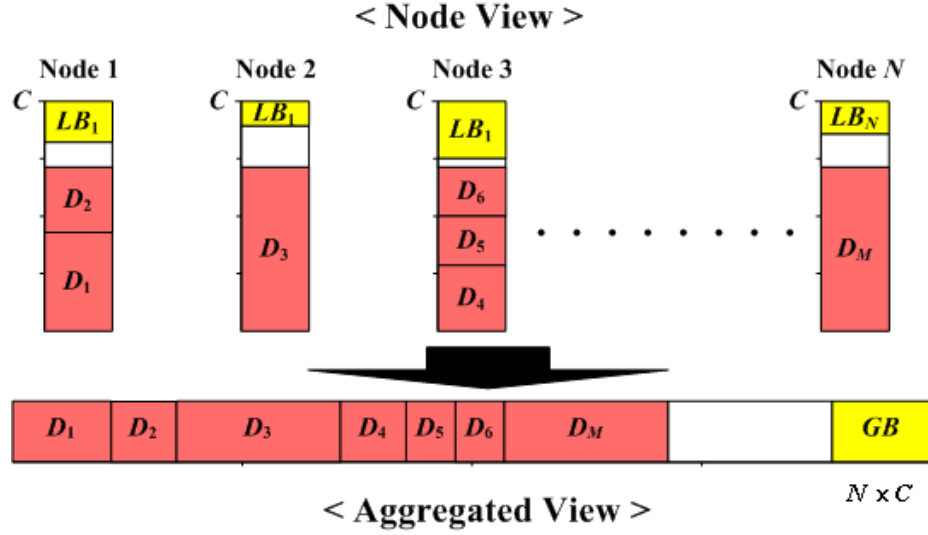


Figure 7.2: Each node has its own Local Buffer. The Global Buffer is a pool of additional capacity spread across the active nodes.

7.2.1 Buffering

The PADD framework uses a two-level buffering scheme to deal efficiently with variations in the workload.

1. Local Buffer (*LB*): A reserved portion of the processing capacity on a single node. Local buffer sizes are specified in fractions of a CPU, so that reserving 10% processor's worth of capacity gives a local buffer size of 0.4 if the capacity (*C*) is 4.0.
2. Global Buffer (*GB*): An separate reserve pool of processing capacity spread across the active machines in the environment. Global buffer sizes are also specified in fractions of a CPU. If a quad-core machine is modeled and each core has 1.0 capacity, reserving a whole machine yields a global buffer size of 4.0.

The Figure 7.2 illustrates how the buffering mechanism works. The local buffer prevents frequent and unnecessary domain migrations by absorbing smaller, transient variations in the aggregate demand of the domains on the machine. If there are k domains running in the node (x), when the aggregate CPU demand of all the domains starts to use *LB*, that is $C_x - \sum_{i=1}^k D_i(t) \leq LB$, it triggers an event to spread the demand over the physical nodes by migrating domains. However, in the interim, before any migrations can occur, the local buffer absorbs some or all of the additional

demand. Using local buffers protects the SLA in the event that the aggregate demand is changing rapidly. In addition, the global buffer makes additional capacity ready to deal with larger, more permanent changes caused by domains starting new workloads or going to new workload phases. Therefore, the migration delay caused by node transition can be avoided. Both buffers handle surges, but they handle different types of surges.

In addition to supporting fixed buffer sizes, our simulator also supports an adaptive buffering scheme. In order to control the buffer sizes dynamically, the system monitors any SLA violations in a node and across the whole environment. We then calculate the GB and LB for each node using the following equations. If an SLA violation occurs on a node, we increase the LB size of the node. Similarly, we increase the GB size when there is any SLA for any domain across all of the nodes. The GB and LB are dynamically determined as a function of SLA violations.

1. $GB = \alpha \times Violation(all_domains)$
2. $LB_i = \beta \times Violation(active_domains_i)$

The number of violations among domains on a node in the active state is counted when determining the local buffer to avoid frequent changes in the LB size due to domain migrations. Increasing α and β gives less energy savings but allows for stricter adherence to the performance requirements. In our simulations, α and β are empirically chosen as 1 and 0.1. These parameters are derived by finding the best non-adaptive test cases and determining the values that would guide the adaptive system to similar results. We note that this is not necessarily optimal and that it is worth further study. We evaluate the power and performance tradeoffs of the two types of buffering in the next section.

7.2.2 Time cost of transitions

As defined in Section 7.1, there exist four (4) transition states for a node. The transition time of each state is specified as input in our simulation. For a domain, *Premigration* and *Migration* states are considered to be temporary, transitional states. The time spent in these two states also can be set as configurable parameter. In reality, the time delay in these two states are a function of the intensity of workloads, along with virtualization technology.

Turning a node off always reduces the overall power consumption of the computing environment. During node transitions, the machine continues to consume power. We assume for

modeling purposes that the power consumption is constant for the entire duration of the transition and that the node does no useful computing during a transition. Moreover, the power during transition is generally higher than the power consumed when the machine is active but idle. As a result, we expect that frequent node transitions will result in more power consumption than staying in the idle state. To model this, we define a *break-even time* ($T_{break-even}$). The break-even time is the length of time that a node must be completely idle before it costs less in terms of power to turn it off than it does to leave it on. Empty nodes not needed for the global buffer remain on for $T_{break-even}$ before PADD turns them off. The break-even time is defined as the period, for which a node continues to stay in the idle state before its power down so that we can achieve energy savings. In other words, the breakeven time is obtained when the cost is equal to the benefit in terms of energy consumption. When changing power state in a machine, there exists a breakeven time period that can achieve power savings. The break-even time of the node (n) is calculated as follows.

$$T_{break-even}^n = \frac{(P_{O \rightarrow A}^n \times T_{O \rightarrow A} + P_{A \rightarrow O}^n \times T_{A \rightarrow O})}{P_A^n(0)}$$

7.2.3 Additional parameters

There are some other aspects of PADD which are also parameterized. These parameters refine the calculation of a domain's demand over time, the choice of domains to migrate and the assumed length of time that it takes to migrate a domain. These parameters are defined below.

1. Moving Window size (MW): size of the time window used to statistically summarize demands
2. Average Percentile (AP): percentile (%) to be ignored in the average of the demand. This can remove outliers in the demand data in order to avoid under-provisioning of system capacity.
3. Victim Selection (VS): how to choose a domain to be evicted when the remaining capacity in a node is less than LB . There are three (3) victim selection schemes are defined: maximum average demand, minimum average demand, and minimum standard deviation of demands.
4. Migration Delay (MD): time delay (sec) between migrations. This is used to avoid frequent migrations.

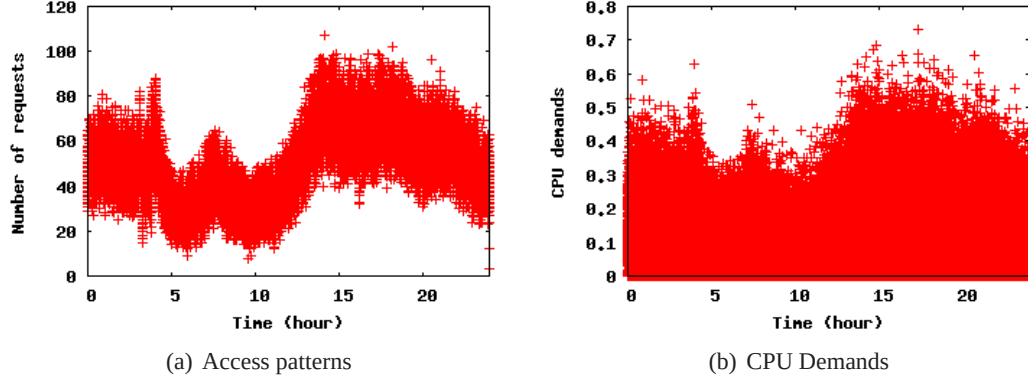


Figure 7.3: One day access logs of the sporting event web site and synthetically generated CPU utilizations for a single domain.

7.2.4 Algorithm

In Algorithm 1, we give the pseudo-code for the PADD algorithm. Periodically, the algorithm runs to adjust overall capacity and reduce power consumption. We use fractional units of a CPU for the metrics of both system capacity and domain demand. We calculate utilization by dividing the demand by the capacity.

The algorithm responds to the current demand by determining how many nodes to use and doing any migrations required. There may be SLA violations due to the fact that PADD is using too few nodes as the demand rises. We also simulate the performance effects of the migration. The *Premigration* state has no performance impact, but in the migration state, all the requested demands are delayed until it leaves the state. This may results in some SLA violations. The PADD simulation keeps track of these delays and makes a detailed analysis of SLA violations.

When determining a victim node due to over-provisioning (line 23), we find nodes on the three condition: no move in/out domain, no SLA violation in running domains, and idle for more than $T_{breakeven}$ time defined in Section 7.2.2. If there is under-provisioning (line 35), on the other side, we increase the number of active nodes. If heterogeneous nodes are given, we will select the node in the order of high power-performance efficiency.

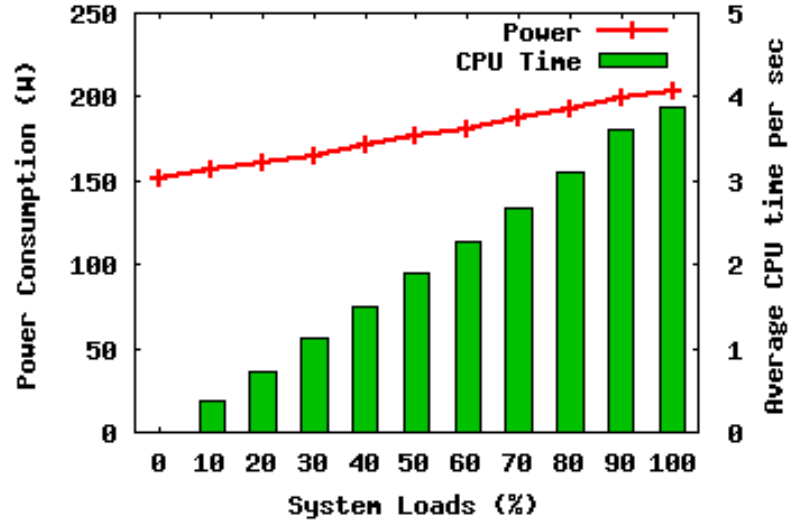


Figure 7.4: Average power consumption and CPU utilizations for 11 different load levels. The power consumption is obtained from the SPECpower_ssj2008 benchmark. While running it, the CPU utilizations are measured in the same machine.

7.3 Evaluation

7.3.1 Workload generation

In this chapter, we use traces obtained from the web site logs for a major international sporting event. Figure 7.3(a) shows the access patterns of the web site for a day. The y-axis represents the number of requests. The request patterns show that there exists significant variability in both daily and hourly accesses.

While we may be using traces derived from a web-based workload, it is important to note that we are simulating long-lived, stateful applications. We use a web-derived trace in the absence of direct measurements of such applications; this is a reasonable technique, because long-lived back-end applications are often driven by a transient web-based request stream, so they share the same usage pattern.

Since we do not have access to measured CPU utilizations, we generate synthetic CPU demands from the access traces by running the SPECpower_ssj2008 benchmark on a real system. We transform the request trace into a set of CPU utilizations using a calibration procedure that is based on measured workload intensity and measured CPU utilization for a representative program. The intensity levels used with the program range from 100% of maximum possible load to 0% in

Table 7.2: Parameter values for the simulation.

Param	Value	Param	Value
N	40	M	40
P_O	0 (W)	P_S	110 (W)
$P_{A \rightarrow S}$	151 (W)	$P_{S \rightarrow A}$	151 (W)
$P_{S \rightarrow O}$	187 (W)	$P_{O \rightarrow S}$	187 (W)
$T_{A \rightarrow S}$	1.0 (s)	$T_{S \rightarrow A}$	1.0 (s)
$T_{S \rightarrow O}$	30 (s)	$T_{O \rightarrow S}$	180 (s)
T_P	60 (s)	T_M	3 (s)
C	4.0	SLA	SLA(99,5),SLA(95,1)

increments of 10%. We count the transactions processed while measuring the CPU utilization, and then, based on these data, we do a linear interpolation to calculate for a particular request rate in the trace, the associated CPU utilization. We acknowledge that this is not ideal. But it is the best available approach. The benchmark generates throughput-oriented workloads and provides precise throughput control based on a measured maximum.

The system used in the generation of the CPU utilization is an AMD Opteron dual-core, dual-processor machine with 4 GB memory. While running the benchmark, we obtain CPU utilization and power draw for the different load levels as shown in Figure 7.4. Since the machine has 4 CPU cores, the total CPU time per second is always less than or equal to 4.0. The power consumption ranges between 151 and 203 watts. Based on this, we generate a synthetic CPU demand trace by applying a normal probability distribution. We also scale the number of requests in the traces to generate a more representative set of processor utilizations. Figure 7.3(b) shows the CPU demands generated with the access logs from Figure 7.3(a). In order to generate multiple CPU demands while maintaining the pattern, we amplified the coarse-grain trace. Multiple days of CPU demands are generated in the same way to simulate many domains running.

7.3.2 System configuration

Table 7.2 shows time delays and the power usage for all node state transitions, as measured on our hardware. These server characteristics are used as input to all the simulations.

In our evaluation, we simulate 40 domains, where each domain’s trace is taken from a different day’s access pattern. The samples in the trace are 1 second apart. We assume that the CPU demand in a domain can require up to 1.0 CPU per second. As a performance requirement,

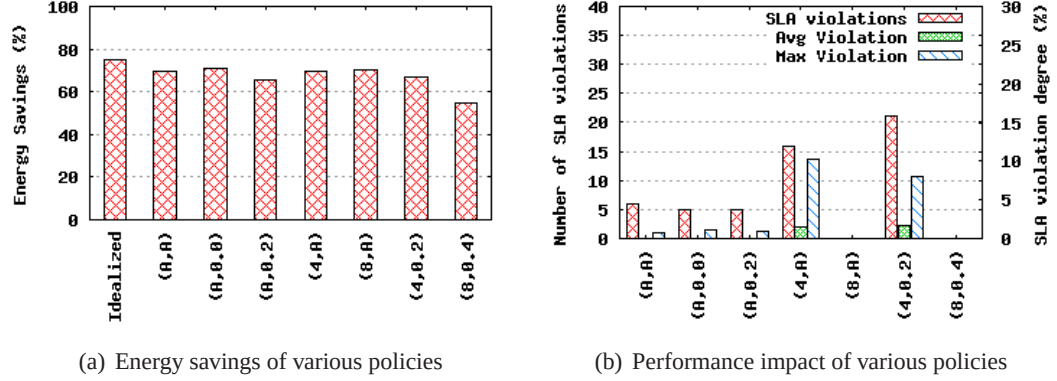


Figure 7.5: The overall comparison of the representative cases. (Util=14.5%, MW=60, AP=60, VS=1, MD=30)

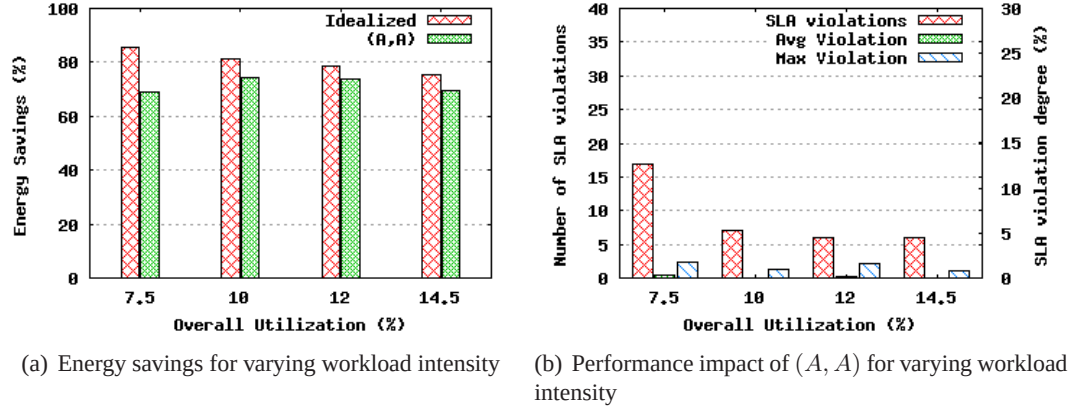


Figure 7.6: The comparison of the adaptive GB/LB with different workload intensity. (MW=60, AP=60, VS=1, MD=30)

we impose two SLAs. Half of the domains must process 99% of the incoming requests in 5 seconds while the other half must process 95% of them in 1 second. In this chapter, we simulate 40 physical nodes with the same capacity. We do not consider the case where the capacity (C) is different by nodes. This is in the highest priority in our future work.

7.3.3 Analysis

Next, we analyze the energy savings and the performance impact caused by different configurations. The energy savings are calculated by comparing against a base case in which there was no consolidation. We measure performance in terms of how well SLAs were respected. To this

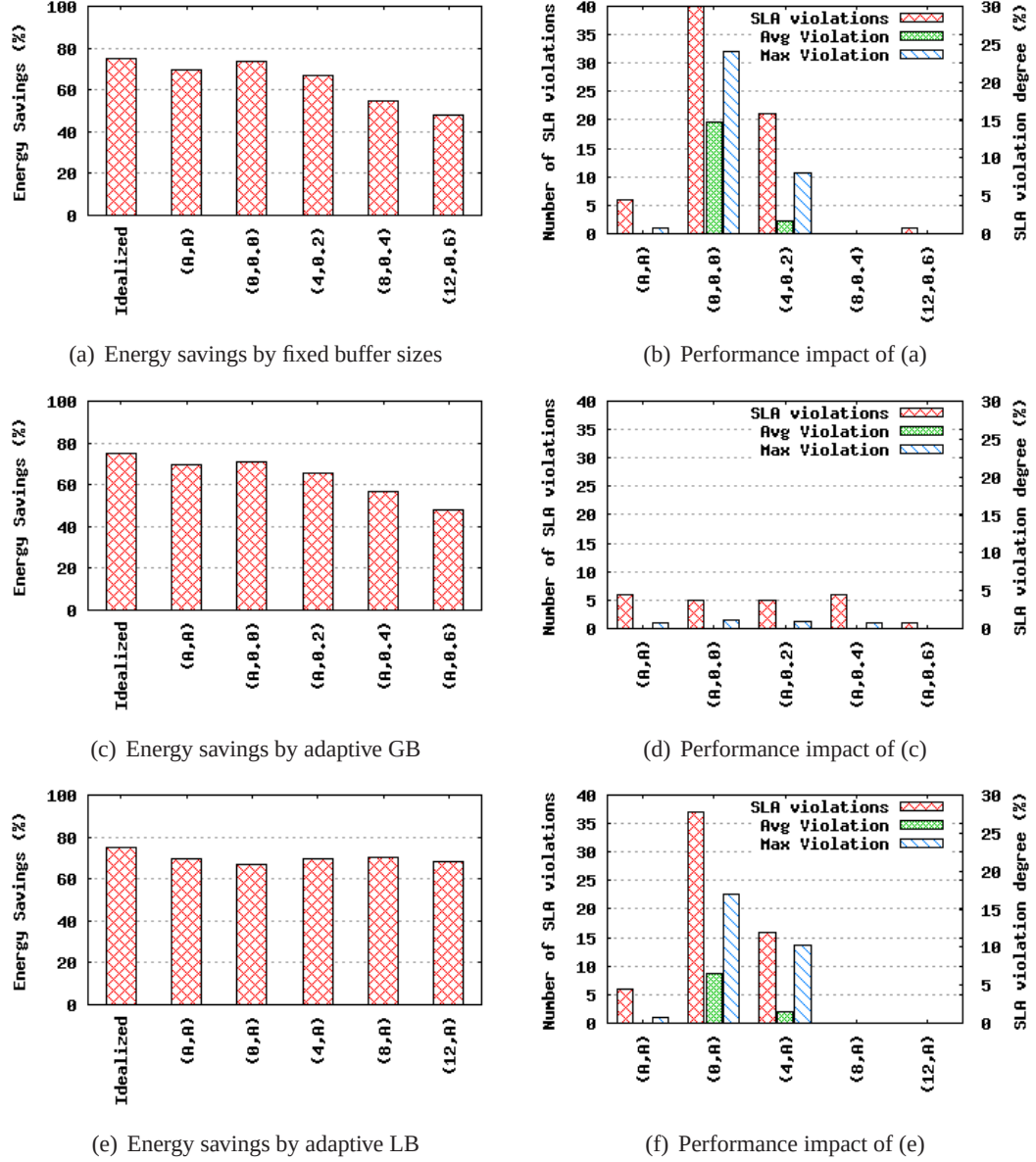
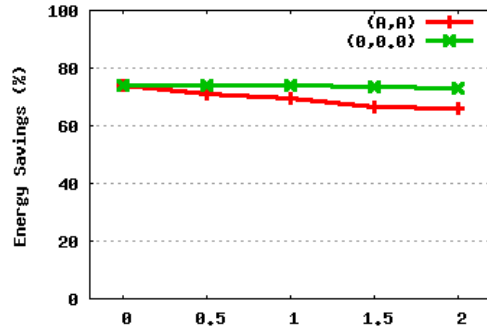
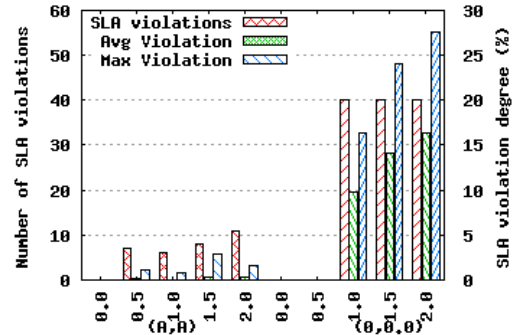


Figure 7.7: The comparison of adaptive buffering and fixed buffering. (Util=14.5%, MW=60, AP=60, VS=1, MD=30)

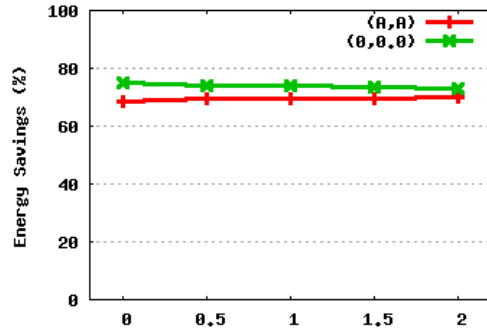
end, we define the *violation degree* (D_i) as the difference between the desired success rate and the measured rate in the domain i . For example, if a domain has SLA(95,1), then it needs to meet 95% of demands within one second. If only 93% of demands are met in the given time, then the violation degree is 2% ($95 - 93$). When considering multiple domains, we present either the *average violation*



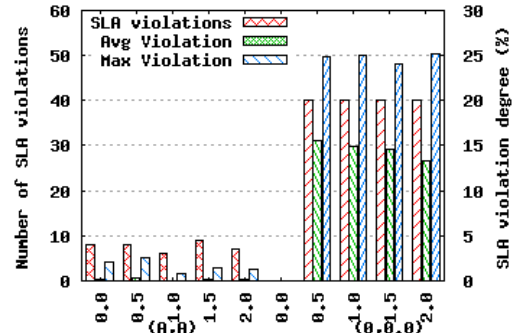
(a) Energy savings by different domain migration costs. Transition cost is set to 1.0.



(b) Performance impact of (a)

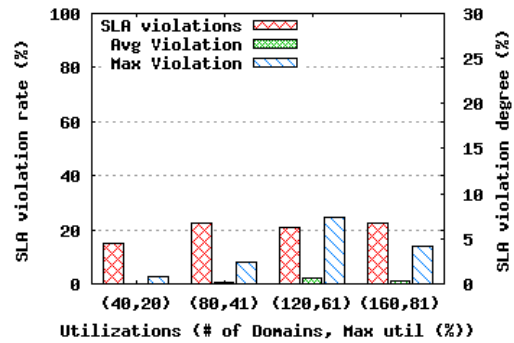
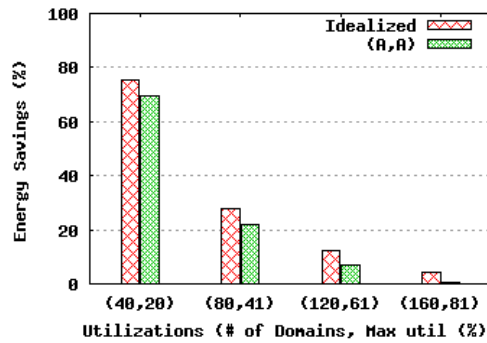


(c) Energy savings by different node transition costs. Migration cost is set to 1.0.



(d) Performance impact of (c)

Figure 7.8: The comparison of domain migration and node transition cost. (Util=14.5%, MW=60, AP=60, VS=1, MD=30)



(a) Energy savings by varying the number of domains. (b) Performance impact of (A,A) for different number of domains

Figure 7.9: The comparison of the adaptive GB/LB with different number of domains. On X-axis, (x,y) indicates the number of domains and the maximum aggregate utilization, respectively. (MW=60, AP=60, VS=1, MD=30)

degree ($\frac{\sum_i^M D_i}{M}$) or the maximum violation degree ($\max(D_i)$).

For each evaluation, we compare 5 different PADD approaches while fixing some parameter values.

1. *Idealized*: there are no migration and transition time delays. GB and LB are not necessary. The *Idealized* always gives the lowest energy consumption without performance hurt and represents the best tradeoffs.
2. *Adaptive GB and LB (A,A)*: GB and LB are dynamically determined depending on SLA violation at runtime.
3. *Adaptive GB and fixed LB (A,l)*: only GB is dynamically determined. The LB is fixed to l .
4. *Fixed GB and adaptive LB (g,A)*: only LB is dynamically determined. The GB is fixed to g .
5. *Fixed GB and LB (g,l)*: During simulation, GB and LB are the fixed values g and l , respectively.

Figure 7.5 shows the overall energy savings and performance impact of the representative cases. The best case is $(8, A)$ which indicates an adaptive LB with a fixed GB of 8.0 CPUs (2 nodes). This case does not have any SLA violations. The fully adaptive (A, A) case does nearly as well, achieving 92.5% of the energy savings of the *Idealized* system with almost no average violation degree. With fixed GB/LB, $(4, 0.2)$ achieves similar energy savings to (A, A) but it has a maximum violation degree of 10%. The $(8, 0.4)$ configuration over-provisions capacity and therefore achieves significantly lower energy savings.

In Figure 7.6, we show the energy savings of the adaptive GB/LB approach with respect to varying workload intensity. The average utilization is formulated as the sum of demands of all domains for simulation time divided by total CPU capacity provided. As the utilization increases, the energy savings of the system decreases by about 10%. The adaptive GB/LB approach achieves 90% of the energy savings of the *Idealized* system on average. At a utilization of 7.5%, there are more opportunities to consolidate domains, which causes more node transitions and domain migrations. This results in more energy savings but more SLA violations. The average and maximum violation degrees, however, are limited to 0.39% and 1.79%, respectively.

In Figure 7.7, we evaluate the buffering schemes: fixed GB/LB, adaptive GB, adaptive LB, and adaptive GB/LB. In particular, $(A,0)$ or $(0,A)$ can be regarded as one level adaptive buffering scheme.

Overall, the adaptive GB/LB outperforms the others by achieving energy savings within 7% of the *Idealized* system with limited performance penalty. Various fixed GB/LB configurations are shown in Figures 7.7(a) and 7.7(b). In this case, only (8, 0.4) and (12, 0.6) offer acceptable performance. With the adaptive GB configurations shown in Figures 7.7(c) and 7.7(d), we see that the choice of LB has little effect on the performance impact but a major effect on energy savings. The reason is that the role of LB is taken by GB. Once enough GB is given, performance is not likely hurt much by long node transition time. In fact, the increase of LB in the adaptive GB approach reduces energy savings by 23%. In the adaptive LB approach (Figures 7.7(e) and 7.7(f)), the reverse is true: varying GB has a much greater effect on performance than energy savings. This is because as GB approaches zero, migrations from overloaded servers are having to wait for compute resources to come online.

In Figure 7.8, we show how the domain migration and node transition costs affect energy savings and performance. Each time cost is normalized to the real measured time delay specified in Table 7.2. When increasing the domain migration time while transition delay is fixed, the energy savings in the adaptive GB/LB is reduced (Figure 7.8(a)). This is because more processing delays occurred followed by more domains involved in transitions. This performance penalty is clearly shown in the case (0, 0.0) in Figure 7.8(b). When the domain migration time becomes 0, there is almost no performance penalty for migration and the greatest energy savings is achieved. The only other potential source of performance degradation is unexpected variation of CPU demand, which we account for not only with the Global and Local Buffers, but also average percentile (60% in this case). Therefore, the given SLA requirements are generally met, although there can be some delays due to the transition time (e.g. when a busy node is waiting for additional nodes to come online). As node transition time increases while the migration cost is fixed at 1.0, we miss out on some power saving opportunities, resulting in diminished energy savings, as shown in Figure 7.8(c). However, the difference between 0.0 and 2.0 is negligible (less than 3%). In terms of performance impact, for fixed LB/GB, the increasing node transition cost actually has a positive effect, because it blocks unnecessary domain migration caused by transient spikes in demand. In our adaptive GB/LB approach, node transition cost does not significantly affect the overall performance. We also evaluated different policies with changing the parameters defined before. Our findings from those changes are less interesting. We did not find that the changes had much of an impact, compared with GB/LB sizes and management.

In Figure 7.9, we show the energy savings and performance impacts of the adaptive

GB/LB approach by increasing the number of domains. The same 40 demand traces are used for additional domains. Accordingly, the overall utilization increases as the number of domains increases. On the x-axis, (x,y) represents the number of domains (x) and the maximum utilization (y) respectively. As the number of domains increases along with overall utilization, the energy savings in the *Idealized* system decreases from 75.2% to 4.3%, due to reduced residual in processing capacity. Our adaptive scheme achieves energy savings within 6% of the *Idealized* system. The SLA violation rate ($\frac{\text{Number of SLA Violations}}{\text{Number of Domains}}$), however, remains below 22.5%. The average violation degree is 0.6% or less in all cases, although additional migrations caused by running more domains increase the maximum violation degree up to 7.4%.

7.4 Summary

This chapter presents an algorithm for consolidating domains in a virtualized environment running coarse-grained workloads to reduce overall energy consumption. We show through a simulation-based study that we get the best combination of energy savings and performance by using an adaptive buffering scheme to determine how much reserve capacity is needed. We use a two-level scheme with a Local Buffer on each node and an additional Global Buffer. A number of other dimensions were evaluated including the time to migrate a domain and the time to change the power state of a node, but none of them have as much effect on PADD as the size and management of the buffers. The best trade-off for effectively no performance loss and maximum energy savings is the adaptive GB/LB approach. More results are presented in [105].

Our future work is twofold. First, we want to run our simulator using traces from a wider variety of workloads and incorporate mixes of different workloads. We believe that mixing different workloads in the simulation is more realistic for emerging computing environments such as utility grids and clouds. Second, we made a number of simplifying assumptions which make our pool of nodes homogeneous. This is clearly unrealistic for any large installation. We need to extend our algorithm and our simulation to consider machines with different characteristics.

In the long term, the simulator will provide an understanding of the design principles involved in the PADD system, allowing us to develop a practical implementation.

Algorithm 1 Power Aware Domain Distribution

```

1: // Retrieve the node/domain states.
2: updateNodeandDomainStates()
3: // Find all the nodes where LB is below the specified level
4: for  $n = 1$  to  $N$  do
5:   if  $C_n < LB_n + \sum Avg(D_i)$  then
6:      $nodes \leftarrow n$ 
7:   end if
8: end for
9: for  $n$  in  $nodes$  do
10:  // Choose a victim domain in the node. see Sec. 7.2.3
11:   $domainID \leftarrow selectVictimDomain(n)$ 
12:  // Determine where to migrate the domain
13:   $targetNodeID \leftarrow findTargetNode(domainID)$ 
14:  // If there exists an active node that has sufficient LB
15:  if  $targetNodeID$  is valid then
16:     $beginMigration(domainID, targetNodeID)$ 
17:  end if
18: end for
19: // Check if there is enough provisioning in GB and LB
20:  $reqCap \leftarrow GB + \sum_{i=1}^N LB_i + \sum_{j=1}^D Avg(D_j) - \sum^{NS_i=Active} C_i$ 
21: if  $reqCap < 0$  then
22:  // If there is over-provisioning in total system capacity,
23:  // find the nodes where domains can be consolidated
24:   $nodelist = getTurnOffCandidates()$ 
25:  for  $nodeID$  in  $nodelist$  do
26:    if  $reqCap + C_{nodeID} < 0$  then
27:      // If all the domains in the node can fit into other nodes,
28:      // vacate the node by evicting all of them,
29:      if  $beginDomainConsolidation(nodeID)$  then
30:         $reqCap \leftarrow reqCap + C_{nodeID}$ 
31:      end if
32:    end if
33:  end for
34: else if  $reqCap > 0$  then
35:  repeat
36:     $nodeID = getNodeWithHighestEfficiency()$ 
37:     $powerupNode()$ 
38:     $reqCap \leftarrow reqCap - C_{nodeID}$ 
39:  until  $reqCap \leq 0$ 
40: end if

```

Chapter 8

Concluding remarks

In this section, we first describe a few future research directions focusing on power consumption analysis in a fine granularity. Then, we will conclude our dissertation by summarizing our accomplishments.

8.1 A runtime power-performance directed adaptation

When improving power-performance efficiency, the tradeoffs should be carefully considered. The tradeoffs vary by underlying hardware and software. As mentioned earlier, many dynamic adaptations of power and performance in previous work are achieved by proper performance guidance. Performance metrics account for performance behaviors, not power. Therefore, the systems have to learn the tradeoffs by asking for additional data when working for accurate power estimation with performance data. While earlier work mostly focuses on training a performance model, such as execution samplings, runtime power training has not been considered in online power-performance adaptation.

This work is to develop a runtime framework in which power-performance is adapted by continuous power and performance monitoring. The work is the extension of our previous work (Chapter 3) to advance our previous power savings work. Given the code regions we previously defined, one can collect both the performance data and the power data per region at runtime. The previous data is used for determining the p-state in the next region. If it achieves a desired power-performance efficiency, then it maintains the p-state. Otherwise, it adjusts the p-state either to increase the performance or to reduce the power. This process is repeated until the program termi-

nates. In this framework, one issue is that the best p-state should be determined quickly to adapt the system. Also, there is another challenge to reduce performance penalty by an incorrect decision. The performance penalty will be estimated on a performance model.

We believe that this approach is useful in that any metric can be specified for preferred power-performance tradeoffs. In our previous work (Chapter 3), the system is customized for EDP (Energy Delay Product), because the performance is the primary concern in HPC (High Performance Computing). Depending on applications, different power-performance efficiency metrics can be adopted, such as Energy, EDP, or ED^2P (Energy Delay Square Product). This framework can be tuned for any power-performance metric, because it relies on actual power consumption for future adaptation in each code region.

To the best of our knowledge, a runtime power-performance guided adaptation is an unexplored research work. This approach is useful in that the framework requires no or minimal a priori information about both a hardware and a software. If power and performance data can be measured at runtime, the system training or learning process can be removed. In addition, we expect to find more power savings opportunities by fine-grained, accurate power monitoring. In a performance-directed method, the power savings is often conservative because the performance penalty by misprediction tends to be relatively large. This approach, however, can reduce this penalty by continuous power monitoring and immediate adaptation.

To sum up, there are three anticipated contributions in this work. First, one can achieve a better power-performance adaptation in any region we did not explore in the previous work, such as a memory intensive region. Second, a runtime adaptation can be achieved without extensive prior data through sampling and learning processes. Third, the in-depth analysis of power-performance adaptation can be made to explain our improvement.

8.2 Dynamic power efficiency adaptation on multi-core architecture

The performance of modern microprocessors has continued to increase exponentially over decades by fast clock speeds. Recently, the advance is limited by increasing power consumption and the design focus in microprocessor architecture shifts from increasing clock speeds to using multiple cores on a single chip [74, 106]. These trends increase performance with smaller power increase. Existing parallel programming paradigms, such as MPI or OpenMP, support a smooth transition of an existing application to take advantage of multiple cores. However, multiprocessors

on a chip compete for the shared resources (in cache and memory). This may cause low performance scalability with significant power growth. Porterfield *et al.* [107] measured the power-performance efficiencies of NPB OMP benchmarks on an AMD Phenom machine. The results show that the power savings rise as the number of cores increases. But the performance gap decreases by achieving less speedup. Therefore, the system with multi-core architecture often limits power-performance efficiency more than expected.

This work is to design a framework adapting the power-performance efficiency of an OpenMP application on multi-core architecture. OpenMP is an application programming interface (API) supporting shared-memory parallel programming. Our previous work did not consider OpenMP applications for power-performance improvement. There are two dynamic power-performance control techniques for OpenMP applications on multi-core architecture. One is p-state transition for CPU frequency scaling. The other is to change the number of threads in a parallel region. A change in number of threads results in a change of the number of running cores. Therefore, the two technique will be controlled in a combined way. When properly executed, this can improve power-performance efficiency. This approach, however, comes with two difficulties. First, it needs to identify where the techniques should be beneficial while running an application. Different workloads usually result in different power-performance efficiencies. Therefore, the change of workloads should be dynamically detected for applying the techniques. Second, a fast and efficient decision needs to be made in order to avoid penalty by delay. Every time the region starts and ends, we have to take an immediate action so that it improves the efficiency as anticipated. By incorporating fine-grained monitoring, our framework will determine the power efficient sweet spots by exploring the two dimensions.

The contribution of this work can be anticipated in two ways. First, one can evaluate the potential impacts of all power-performance adaptation techniques available on multi-core architecture including DVFS, number of threads (specific to OpenMP), and number of active cores. When adapting power-performance, it is worthy to understand the power-performance impact for each technique. Second, the runtime adaptation by continuous monitoring and tuning can achieve more power efficiency than the previous approach using power estimation. This approach also has the advantage of requiring less priori information for sophisticated adaptation.

8.3 Conclusion

We presented five projects for improving power and performance efficiency in parallel and distributed systems. Each project made successful contributions to realize power awareness depending on the characteristics of applications running on recent computing systems. First, we developed a transparent, adaptive system for reducing the p-state in MPI communication phases. The system shows significant energy savings with little performance degradation. Second, we proposed a way to determine the minimum energy consumption by applying voltage and frequency scaling on the fly. Our efficient scheduling algorithm showed that the running time can be reduced significantly with our algorithm. Third, we evaluated the internal power monitoring device, PowerMon. It is designed to perform fine-grained, accurate real-time power monitoring and eventually resolve and highlight differing energy efficiencies of phases within iterations of scientific applications. Fourth, we developed a power-performance prediction model that can correlate power and performance data on a chip multiprocessor architecture. In our evaluations, we show that our power-performance model overall achieves accurate power estimations for multiple different configurations. Finally, we developed and simulated an adaptive algorithm to save power by dynamically migrating virtual machines onto fewer physical machines when possible. According to our simulations demonstrating power and performance changes, this adaptive scheme achieves significant energy savings, compared to the idealized system that has no performance penalty. As future work, we focus on finding an adaptive solution balancing power and performance depending on the performance needs of multi-threaded application.

Bibliography

- [1] Mark E. Femal. Non-uniform power distribution in data centers for safely overprovisioning circuit capacity and boosting throughput. Master's thesis, North Carolina State University, Raleigh, NC, May 2005.
- [2] Richard Goering. Current physical design tools come up short. *EE Times*, April 14 2000.
- [3] Wu-chun Feng and Kirk Cameron. The Green500 List: Encouraging Sustainable Supercomputing. *IEEE Computer*, 40(12), December 2007.
- [4] Shekhar Borkar. Design challenges of technology scaling. *IEEE Micro*, 19(4):23–29, 1999.
- [5] Nandini Kappiah, Vincent W. Freeh, and David K. Lowenthal. Just in time dynamic voltage scaling: Exploiting inter-node slack to save energy in MPI programs. In *ACM Supercomputing conference*, 2005.
- [6] Krisztian Flautner, Steven K. Reinhardt, and Trevor N. Mudge. Automatic performance setting for dynamic voltage scaling. In *Mobile Computing and Networking*, pages 260–271, 2001.
- [7] Padmanabhan Pillai and Kang G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *ACM Symposium on Operating Systems Principles*, pages 89–102, 2001.
- [8] Trevor Pering, Tom Burd, and Robert Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. pages 76–81.
- [9] David P. Helmbold, Darrell D. E. Long, and Bruce Sherrod. A dynamic disk spin-down technique for mobile computing. In *Mobile Computing and Networking*, pages 130–142, 1996.

- [10] R. Krashinsky and H. Balakrishnan. Minimizing energy for wireless web access with bounded slowdown. In *MobiCom 2002*, Atlanta, GA, September 2002.
- [11] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, Amin Vahdat, and Ronald P. Doyle. Managing energy and server resources in hosting centres. In *Symposium on Operating Systems Principles*, 2001.
- [12] Eduardo Pinheiro, Ricardo Bianchini, Enrique V. Carrera, and Taliver Heath. Load balancing and unbalancing for power and performance in cluster-based systems. In *Workshop on Compilers and Operating Systems for Low Power*, September 2001.
- [13] E.N. (Mootaz) Elnozahy, Michael Kistler, and Ramakrishnan Rajamony. Energy-efficient server clusters. In *Workshop on Mobile Computing Systems and Applications*, Feb 2002.
- [14] Charles Lefurgy, Karthick Rajamani, Freeman Rawson, Wes Felter, Michael Kistler, and Tom W. Keller. Energy management for commercial servers. *IEEE Computer*, pages 39–48, December 2003.
- [15] Elmootazbellah Elnozahy, Michael Kistler, and Ramakrishnan Rajamony. Energy conservation policies for web servers. In *Usenix Symposium on Internet Technologies and Systems*, 2003.
- [16] Vivek Sharma, Arun Thomas, Tarek Abdelzaher, and Kevin Skadron. Power-aware QoS management in web servers. In *IEEE Real-Time Systems Symposium*, Cancun, Mexico, December 2003.
- [17] Tibor Horvath, Tarek Abdelzaher, Kevin Skadron, and Xue Liu. Dynamic voltage scaling in multi-tier web servers with end-to-end delay control. In *IEEE Transactions on Computers*, November 2004.
- [18] Jeffrey O. Kephart, Hoi Chan, Rajarshi Das, David W. Levine, Gerald Tesauro, Freeman Rawson, and Charles Lefurgy. Coordinating multiple autonomic managers to achieve specified power-performance tradeoffs. In *IEEE International Conference on Autonomic Computing*, June 2007.
- [19] Mark E. Femal and Vincent W. Freeh. Boosting data center performance through non-uniform power allocation. In *International Conference on Autonomic Computing*, June 2005.

- [20] E. Pinheiro, R. Bianchini, E. Carrera, and T. Heath. Dynamic cluster reconfiguration for power and performance. In M. Kandemir L. Benini and J. Ramanujam, editors, *Compilers and Operating Systems for Low Power*. Kluwer Academic Publishers, 2002.
- [21] E. Pinheiro, R. Bianchini, E. Carrera, and T. Heath. Load balancing and unbalancing for power and performance in cluster-based systems, 2001.
- [22] X. Feng, R. Ge, and K. Cameron. Power and energy profiling of scientific applications on distributed systems, 2005.
- [23] Enrique V. Carrera, Eduardo Pinheiro, and Ricardo Bianchini. Conserving disk energy in network servers. In *Intl. Conference on Supercomputing*, June 2003.
- [24] Qingbo Zhu, Francis M. David, Christo Devaraj, Zhenmin Li, Yuanyuan Zhou, and Pei Cao. Reducing energy consumption of disk storage using power-aware cache management. In *High-Performance Computer Architecture*, February 2004.
- [25] Athanasios E. Papathanasiou and Michael L. Scott. Energy efficiency through burstiness. In *Workshop on Mobile Computing Systems and Applications*, October 2003.
- [26] Chris Gniady, Y. Charlie Hu, and Yung-Hsiang Lu. Program counter based techniques for dynamic power management. In *The 10th International Symposium on High-Performance Computer Architecture*, 2004.
- [27] Xizhou Feng, Rong Ge, and Kirk W. Cameron. Power and energy profiling of scientific applications on distributed systems. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, April 2005.
- [28] Seung Woo Son, Guangyu Chen, Mahmut Kandemir, and Alok Choudhary. Dynamic compilation for reducing energy consumption of i/o-intensive applications. In *International workshop on Language and Compilers for Parallel Computing*, 2005.
- [29] Rong Ge, Xizhou Feng, and Kirk W. Cameron. Performance-constrained distributed DVS scheduling for scientific applications on power-aware clusters. In *ACM Supercomputing conference*, 2005.

- [30] Rong Ge, Xizhou Feng, Wuchun Feng, and Kirk W. Cameron. CPU MISER: A performance-directed, run-time system for power-aware clusters. In *International Conference on Parallel Processing*, 2007.
- [31] Chung hsing Hsu and Wu chun Feng. A power-aware run-time system for high-performance computing. In *Supercomputing*, November 2005.
- [32] Vincent W. Freeh, David K. Lowenthal, Rob Springer, Feng Pan, and Nandani Kappiah. Exploring the energy-time tradeoff in MPI programs on a power-scalable cluster. In *International Parallel and Distributed Processing Symposium*, April 2005.
- [33] Vincent W. Freeh, David K. Lowenthal, Feng Pan, and Nandani Kappiah. Using multiple energy gears in MPI programs on a power-scalable cluster. In *Principles and Practices of Parallel Programming*, June 2005.
- [34] Nandani Kappiah, Vincent W. Freeh, and David K. Lowenthal. Just in time dynamic voltage scaling: Exploiting inter-node slack to save energy in MPI programs. In *Supercomputing*, November 2005.
- [35] Robert C. Springer IV, David K. Lowenthal, Barry Rountree, and Vincent W. Freeh. Minimizing execution time in MPI programs on an energy-constrained, power-scalable cluster. In *ACM Symposium on Principles and Practice of Parallel Programming*, March 2006.
- [36] Guilin Chen, Konrad Malkowski, Mahmut Kandemir, and Padma Raghavan. Reducing power with performance constraints for parallel sparse applications. In *Workshop on High-Performance, Power-Aware Computing*, April 2005.
- [37] C-H. Hsu and U. Kremer. The design, implementation, and evaluation of a compiler algorithm for CPU energy reduction. In *Programming Languages, Design, and Implementation*, June 2003.
- [38] Matthew Curtis-Maury, Filip Blagojevic, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. Prediction-based power-performance adaptation of multithreaded scientific codes. In *IEEE Transactions on Parallel and Distributed Systems (TPDS)*. IEEE Computer Society, 2008.

- [39] Matthew Curtis-Maury. *Improving the Efficiency of Parallel Applications on Multithreaded and Multicore Systems*. PhD thesis, Department of Computer Science, Virginia Tech, 2008.
- [40] Min Yeol Lim and Vincent W. Freeh. Determining the minimum energy consumption using dynamic voltage and frequency scaling. In *The 3rd Workshop on High-Performance, Power-Aware Computing*, March 2007.
- [41] Fen Xie, Margaret Martonosi, and Sharad Malik. Bounds on power savings using runtime dynamic voltage scaling: An exact algorithm and a linear-time heuristic approximation. In *International Symposium on Low Power Electronics and Design*, August 2005.
- [42] N.D. Adiga et al. An overview of the BlueGene/L supercomputer. In *Supercomputing*, November 2002.
- [43] M. Warren, E. Weigle, and W. Feng. High-density computing: A 240-node Beowulf in one cubic meter. In *Supercomputing*, November 2002.
- [44] Chris Gniady, Y. Charlie Hu, and Yung-Hsiang Lu. Program counter based techniques for dynamic power management. In *HPCA*, pages 24–35, 2004.
- [45] Chris Gniady, Ali Raza Butt, and Y. Charlie Hu. Program-counter-based pattern classification in buffer caching. In *OSDI*, pages 395–408, 2004.
- [46] Ali Raza Butt, Chris Gniady, and Y. Charlie Hu. The performance impact of kernel prefetching on buffer cache replacement algorithms. In *SIGMETRICS*, pages 157–168, 2005.
- [47] M. Huang, J. Renau, and J. Torellas. Positional adaptation of processors: Application to energy reduction. In *International Symposium on Computer Architecture*, June 2003.
- [48] A. Dhodapkar and J. Smith. Comparing phase detection techniques. In *International Symposium on Microarchitecture*, pages 217–227, December 2003.
- [49] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *Architectural Support for Programming Languages and Operating Systems*, October 2002.
- [50] Ken Kennedy and Ulrich Kremer. Automatic data layout for distributed-memory machines. *ACM Transactions on Programming Languages and Systems*, 20(4):869–916, 1998.

- [51] Umit Rencuzogullari and Sandhya Dwarkadas. Dynamic adaptation to available resources for parallel computing in an autonomous network of workstations. In *Principles and Practice of Parallel Programming*, June 2001.
- [52] M. Steinder, I. Whalley, D. Carrera, I. Gaweda, and D. Chess. Server virtualization in autonomic management of heterogeneous workloads. *Integrated Network Management, 2007. IM '07. 10th IFIP/IEEE International Symposium on*, 2007.
- [53] N. Bobroff, A. Kochut, and K. Beaty. Dynamic Placement of Virtual Machines for Managing SLA Violations. *Integrated Network Management, 2007. IM '07. 10th IFIP/IEEE International Symposium on*, pages 119–128, 21 2007-Yearly 25 2007.
- [54] E. N. Elnozahy, Michael Kistler, and Ramakrishnan Rajamony. Energy-Efficient Server Clusters. In *2nd Workshop on Power-Aware Computing Systems*, 2002.
- [55] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, and Amin M. Vahdat. Managing Energy and Server Resources in Hosting Centers. In *the Eighteenth ACM Symposium on Operating Systems Principles*, 2001.
- [56] Karthick Rajamani and Charles Lefurgy. On Evaluating Request-Distribution Schemes for Saving Energy in Server Clusters. In *IEEE Performance Analysis of System and Software*, 2003.
- [57] Ripal Nathuji and Karsten Schwan. VPM Tokens: Virtual Machine-Aware Power Budgeting in Datacenters. In *HPDC '08: Proceedings of the 17th international symposium on High performance distributed computing*. ACM, 2008.
- [58] Akshat Verma, Puneet Ahuja, and Anindya Neogi. pMapper: power and migration cost aware application placement in virtualized systems. In *Middleware '08: Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*. Springer-Verlag New York, Inc., 2008.
- [59] Akshat Verma, Puneet Ahuja, and Anindya Neogi. Power-aware dynamic placement of HPC applications. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*. ACM, 2008.

- [60] D. J. Bradley, R. E. Harper, and S. W. Hunter. Workload-based power management for parallel computer systems. In *IBM Journal of Research and Development*, 2003.
- [61] Gong Chen, Wenbo He, Jie Liu, Suman Nath, Leonidas Rigas, Lin Xiao, and Feng Zhao. Energy-Aware Server Provisioning and Load Dispatching for Connection-Intensive Internet Services. In *5th Symposium on Networked Systems Design USENIX Association and Implementation (NSDI)*, 2008.
- [62] Soraya Ghiasi and Wes Felter. CPU Packing for Multiprocessor Power Reduction. In *Power Aware Computer Systems (PACS)*, 2003.
- [63] Mohammad Banikazemi, Dan Poff, and Bulent Abali. PAM: A Novel Performance/Power Aware Meta-scheduler for Multi-core Systems. In *Supercomputing conference (SC)*, 2008.
- [64] Vincent W. Freeh, Tyler K. Bletsch, and Freeman L. Rawson. Scaling and Packing on a Chip Multiprocessor. In *Workshop on High-Performance, Power-Aware Computing*, 2007.
- [65] Vincent W. Freeh and David K. Lowenthal. Using multiple energy gears in MPI programs on a power-scalable cluster. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 164–173, New York, NY, USA, 2005. ACM.
- [66] Min Yeol Lim, Vincent W. Freeh, and David K. Lowenthal. Adaptive, transparent frequency and voltage scaling of communication phases in mpi programs. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 107, New York, NY, USA, 2006. ACM.
- [67] Ripal Nathuji and Karsten Schwan. VirtualPower: Coordinated Power Management in Virtualized Enterprise Systems. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2007.
- [68] Chung-Hsing Hsu and Wu-chun Feng. Effective dynamic-voltage scaling through CPU-boundedness detection. In *Fourth IEEE/ACM Workshop on Power-Aware Computing Systems*, December 2004.
- [69] Rolf Rabenseifner. Automatic profiling of MPI applications with hardware performance counters. In *PVM/MPI*, pages 35–42, 1999.

- [70] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS parallel benchmarks. RNR-91-002, NASA Ames Research Center, August 1991.
- [71] ASCI Purple Benchmark Suite. <http://www.llnl.gov/asci/platforms/purple/rfp-benchmarks/>.
- [72] A. H. Land and A. G. Doig. An automatic method for solving discrete programming problems, 1960.
- [73] Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1984.
- [74] Kunle Olukotun and Lance Hammond. The future of microprocessors. *Queue*, 3(7):26–29, 2005.
- [75] Power Egg. [http://www.itwatchdogs.com/DataSheets/PowerEgg\(8.5x11\)033006.pdf](http://www.itwatchdogs.com/DataSheets/PowerEgg(8.5x11)033006.pdf).
- [76] Watts Up. <http://www.wattsupmeters.com>.
- [77] ADM1191. <http://www.analog.com/en/power-management/power-monitors/adm1191/products/product.html>.
- [78] Attiny45. http://www.atmel.com/dyn/products/product_card.asp?part_id=3618.
- [79] USB-I2C transceiver. http://www.harbaum.org/till/i2c_tiny_usb/index.shtml.
- [80] Linux hardware monitoring project. <http://www.lm-sensors.org/>.
- [81] John D. McCalpin. Stream: Sustainable memory bandwidth in high performance computers. <http://www.cs.virginia.edu/stream/>.
- [82] The TCP/UDP bandwidth measurement tool. <http://dast.nlanr.net/Projects/Iperf/>.
- [83] Patrick Schmid and Achim Roos. How can battery runtime be shorter? <http://www.tomshardware.com/reviews/ssd-hdd-battery,1955-2.html>.
- [84] Jon Gerow. Power Supply Efficiency.
- [85] Kirk W. Cameron, Rong Ge, and Xizhou Feng. High-performance, power-aware distributed computing for scientific applications. *Computer*, 38(11):40–47, 2005.

- [86] Kirk W. Cameron, Rong Ge, and Xizhou Feng. Designing Computational Clusters for Performance and Power. *Advances in Computers*, 69(2), May 2007.
- [87] Wu-chun Feng, Xizhou Feng, and Rong Ge. Green Supercomputing Comes of Age. *IT Professional*, 10(1), January 2008.
- [88] Perfmon2: the hardware-based performance monitoring interface. <http://perfmon2.sourceforge.net/>.
- [89] The OpenMP API specification for parallel programming. <http://www.openmp.org>.
- [90] C. Spearman. The proof and measurement of association between two things. *Amer. J. Psychol*, 15, 1974.
- [91] Karan Singh, Major Bhadauria, and Sally A. McKee. Real time power estimation and thread scheduling via performance counters. *SIGARCH Comput. Archit. News*, 37(2):46–55, 2009.
- [92] Benjamin C. Lee and David M. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. *SIGOPS Oper. Syst. Rev.*, 40(5):185–194, 2006.
- [93] N.R. Draper and H. Smith. Applied Regression Analysis. *Wiley-Interscience*, 1998.
- [94] Werner Vogels. Beyond Server Consolidation. *ACM Queue*, 2008.
- [95] Xen Virtualization Technology. <http://www.xen.org/>.
- [96] VMware ESX Server. <http://www.vmware.com/products/vi/esx/>.
- [97] Elastic Compute Cloud (Amazon EC2). <http://aws.amazon.com/ec2/>.
- [98] Virtual Computing Lab. <http://vcl.ncsu.edu/>.
- [99] IBM Blue Cloud Computing. <http://www-03.ibm.com/press/us/en/pressrelease/22613.wss>.
- [100] Ulrich Drepper. The Cost of Virtualization. *ACM Queue*, 2008.
- [101] Pradeep Padala, Xiaoyun Zhu, Zhikui Wang, Sharad Singhal, and Kang G. Shin. Performance Evaluation of Virtualization Technologies for Server Consolidation. Technical report, HP Lab., April 2007.

- [102] Christopher Clark, Keir Fraser, Steven Hand, Jakob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live Migration of Virtual Machines. In *2nd Symposium on Networked Systems Design and Implementation*, 2005.
- [103] Aameek Singh, Madhukar Korupolu, and Dushmanta Mohapatra. Server-Storage Virtualization: Integration and Load Balancing in Data Centers. In *Supercomputing conference (SC)*, 2008.
- [104] Richard Neapolitan and Kumarss Naimipour. *Foundations of Algorithms*. 1997.
- [105] Min Yeol Lim, Tyler Bletsch, Vincent W. Freeh, and Freeman Rawson. PADD: Power-Aware Domain Distribution. In *Technical Report To be assigned, IBM Research*, 2009.
- [106] Kent Milfeld, Kazushige Goto, Avi Purkayastha, Chona Guiang, and Karl Schulz. Effective use of commodity multi-core systems in HPC. In *The 8th LCI International Conference on High-Performance Clustered Computing*, 2007.
- [107] Allan Porterfield, Rob Fowler, and Mark Neyer. Maestro: Dynamic runtime power and concurrency adaptation. In *Workshop on Managed Many-Core Systems (MMCS)*, 2008.