PREDICTION STRATEGIES FOR POWER-AWARE COMPUTING ON MULTICORE PROCESSORS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Karan Singh

August 2009

© 2009 Karan Singh

ALL RIGHTS RESERVED

PREDICTION STRATEGIES FOR POWER-AWARE COMPUTING
ON MULTICORE PROCESSORS

Karan Singh, Ph.D.
Cornell University 2009

Diminishing performance returns and increasing power consumption of single-threaded processors have made chip multiprocessors (CMPs) an industry imperative. Unfortunately, low power efficiency and bottlenecks in shared hardware structures can prevent optimal use when running multiple sequential programs. Furthermore, for multithreaded programs, adding a core may harm performance *and* increase power consumption. To better use otherwise limitedly beneficial cores, software components such as hypervisors and operating systems can be provided with estimates of application performance and power consumption. They can use this information to improve system-wide performance and reliability. Estimating power consumption can also be useful for hardware and software developers. However, obtaining processor and system power consumption information can be nontrivial.

First, we present a predictive approach for real-time, per-core power estimation on a CMP. We analytically derive functions for real-time estimation of processor and system power consumption using performance counter and temperature data on real hardware. Our model uses data gathered from microbenchmarks that capture potential application behavior. The model is independent of our test benchmarks, and thus we expect it to be well suited for future applications. For chip multiprocessors, we achieve median error of 3.8% on an AMD quad-core CMP, 2.0% on an Intel quad-core CMP, and 2.8% on an Intel eight-core CMP. We implement the same approach inside an Intel XScale simulator and achieve median error of 1.3%.

Next, we introduce and evaluate an approach to throttling concurrency in parallel programs dynamically. We throttle concurrency to levels with higher predicted efficiency using artificial neural networks (ANNs). One advantage of using ANNs over similar techniques previously explored is that the training phase is greatly simplified, thereby reducing the burden on the end user. We effectively identify energy efficient concurrency levels in multithreaded scientific applications on an Intel quad-core CMP. We improve the energy efficiency for many of our applications by predicting more favorable number and placement of threads at runtime, and improve the average ED^2 by 17.2% and 22.6% on an Intel quad-core and an Intel eight-core CMP, respectively.

Last, we propose a framework that combines both approaches. With the impending shift to many-core architectures, systems need information on power and energy for more energy-efficient use of all cores. Any approach utilizing this framework also needs to be scalable to many cores. We implement an infrastructure that can schedule for power efficiency for a given power envelope, and/or a given thermal envelope. We expect the framework to scale well with number of cores. We perform experiments on quad-core and eight-core platforms. We schedule for better power efficiency by suspending or slowing down (via DVFS) single-threaded programs, or throttling concurrency for multithreaded programs. We utilize the per-core power predictor to schedule applications to remain under a given power envelope. We modify the scheduler policies to take advantage of all power saving options to enforce the power envelope, while minimizing performance loss.

BIOGRAPHICAL SKETCH

Karan Singh was born in August 1983 in Chandigarh, India. He went to St. John's High School and finished his schooling in India, before heading out to cajun country for his undergraduate studies at Louisiana State University in August 2001. There he learned the ways of the crawfish boil and football tailgates, and received a B.S. in Computer Engineering and a B.S. in Electrical Engineering in May 2005. He graduated summa cum laude and was awarded a Tau Beta Pi fellowship for graduate school. Karan then opted to continue his studies in snowy Ithaca and enrolled in the MS/PhD program at Cornell's Computer Systems Laboratory in June 2005. He received his MS in August 2007, and his PhD in August 2009.

To my parents, Paramjit Singh and Dr. Rajinder Kaur,

and my brother, Bikram Singh.

ACKNOWLEDGMENTS

I would first like to express my gratitude towards my advisor, Prof. Sally McKee. Her leadership, support, attention to detail, hard work, and scholarship set an example I hope to match some day. Next, I am indebted to Prof. David Koppelman at LSU for introducing me to Computer Architecture and for sparking my interest in this field. I thank my committee members, Prof. David Albonesi and Prof. Anthony Reeves, for providing insightful comments on this work. Their feedback has been very valuable in improving this thesis. I thank Matthew Curtis-Maury, Bronis de Supinski, and Martin Schulz for their participation and support. I thank the Fusion group (Pete, Brian, Vince, Chris, Cat, Raymond, and, of course, Major) for their feedback and commaraderie. Finally, I thank my family and friends. This thesis would not have been possible without their constant support and faith in me.

Part of this work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48 and under National Science Foundation awards CCF-0444413 and CPA E70-8321. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation, the Lawrence Livermore National Laboratory, or the Department of Energy.

TABLE OF CONTENTS

Biographical Sketch . . . iii
Dedication . . . iv
Acknowledgments . . . v
Table of Contents . . . vi
List of Tables . . . viii
List of Figures . . . ix

1 Introduction 1

2 Real-Time Per-Core Power Estimation for CMPs 5
2.1 Methodology . . . 6
2.2 Microbenchmarks . . . 8
2.3 Event Selection . . . 9
2.4 Temperature Effects . . . 11
2.5 Forming the Model . . . 12
2.6 Experimental Setup . . . 15
2.7 Evaluation . . . 17

3 Power-Aware Thread Scheduling 23
3.1 Simple Policy . . . 24
3.2 Maximum Instructions/Watt Policy . . . 24
3.3 Per-Core Fair Policy . . . 24
3.4 User-based Priorities Policy . . . 25
3.5 Evaluation . . . 26
3.6 What about DVFS? . . . 33

4 Multithreaded Scalability and Predicting Concurrency 35
4.1 Analysis of Application Scalability: Four Cores . . . 36
4.2 Analysis of Application Scalability: Eight Cores . . . 41
4.3 Predicting Concurrency . . . 43
4.4 Overview of Artificial Neural Networks . . . 45
4.5 Evaluation . . . 47

5 Concurrency Throttling 51
5.1 Methodology . . . 53
5.2 Evaluation . . . 55

6 Echo: A Framework for Efficient Power Management 60
6.1 Multiprogrammed Workloads . . . 60
6.2 Single-threaded and Multithreaded Mixture Workloads . . . 68
6.3 Scalability: More Cores and Fine-Grained DVFS . . . 71

7 Related Work **76**
7.1 Power Prediction 76
7.2 Performance Prediction 78
7.3 Power-Aware Scheduling 80
7.4 Concurrency Throttling 81

8 Conclusions and Future Work **84**

A Speeding Simulations **87**
A.1 Setup 88
A.2 Evaluation 88
A.3 Generalized Model for Multiple Frequency Levels 90

B Power Predictor Results without Temperature Input (AMD) **92**

C Power Predictor Results with DVFS Scaling to 1.1 GHz (AMD) **93**

D Power Predictor Results with DVFS Scaling to 2.0 GHz (Intel) **94**

Bibliography **95**

LIST OF TABLES

2.1 PMCs Categorized by Architecture and Ordered (Increasing) by Correlation (for AMD Phenom 9500) . . . 10
2.2 PMCs Categorized by Architecture and Ordered (Increasing) by Correlation (for Intel Q6600) . . . 10
2.3 PMCs Categorized by Architecture and Ordered (Increasing) by Correlation (for Dual Intel E5430) . . . 10
2.4 AMD Phenom 9500 Machine Configuration Parameters . . . 16
2.5 Intel Q6600 and Dual Intel E5430 Machine Configuration Parameters . 17

3.1 Multiprogrammed Workloads for Evaluation . . . 26

4.1 Machine Configuration Parameters . . . 36

6.1 Multiprogrammed Workloads for Evaluation . . . 65
6.2 Mixture Workloads for Scheduler Evaluation . . . 68
6.3 Median Errors for Per-Frequency and Generalized Power Models . . . 72

A.1 Median Errors for Per-Frequency and Generalized Power Models . . . 90

LIST OF FIGURES

2.1 Die Photos for Intel Q6600 [2] (left), and AMD Phenom 9500 [3] (right) 7
2.2 L3 Cache Miss Rates for SPEC 2006 on AMD Phenom 8
2.3 Power vs. Temperature on the AMD 4-Core CMP 11
2.4 An Illustrative Example of Best-Fit Continuous Approximation Functions (left), and a Better Fitting Piece-Wise Function (right) 13
2.5 Measured vs. Predicted Power for AMD Phenom 9500 18
2.6 Median Errors for AMD Phenom 9500 18
2.7 Measured vs. Predicted Power for Intel Q6600 19
2.8 Median Errors for Intel Q6600 19
2.9 Measured vs. Predicted Power for Dual Intel E5430 (8 cores) 20
2.10 Median Errors for Dual Intel E5430 (8 cores) 20
2.11 Cumulative Distribution Function (CDF) Plot Showing Fraction of Space Predicted (y-axis) under a Given Error (x-axis) 21

3.1 Scheduler Setup and Use 23
3.2 Given Workloads, Policies, and Envelopes for AMD Phenom 27
3.3 Given Workloads, Policies, and Envelopes for Intel Q6600 27
3.4 Runtimes for Workloads on AMD Phenom (Normalized to No Power Envelope) 28
3.5 Runtimes for Workloads on Intel Q6600 (Normalized to No Power Envelope) 30
3.6 Runtimes for Workloads on Dual Intel E5430 (Normalized to No Power Envelope) 31
3.7 Temperature Across Policies for a Sample Workload 32
3.8 Runtimes for Workloads when using DVFS in combination with *simple* on AMD Phenom (Normalized to No Power Envelope) 34

4.1 Execution times by Hardware Configuration (the bottom-right graph shows the average normalized execution time across all benchmarks) . 37
4.2 Power and Energy Consumption by Hardware Configuration (the bottom-right graphs shows the geometric mean of the normalized energy and power consumption across all benchmarks) 39
4.3 Execution Times by Hardware Configuration 41
4.4 Power and Energy Consumption by Hardware Configuration 42
4.5 IPCs observed during Phases of *sp* for each Thread Configuration on the 4-core System 44
4.6 IPCs observed during Phases of *sp* for each Thread Configuration on the 8-core System 44
4.7 Simplified Diagram of a Fully Connected, Feed-Forward ANN 45
4.8 Example of a Hidden Unit with a Sigmoid Activation Function 46
4.9 Cumulative Distribution Function (CDF) of Prediction Error for the 4-core System (left), and the 8-core System (right) 48

4.10 Percent of Phases for which each Ranking Configuration is Selected on the 4-core System (left), and the 8-core System (right) 49

5.1 Runtime System for Concurrency Throttling 54
5.2 Execution Time, Power Consumption, Energy Consumption, and ED^2 of Prediction-Based Adaptation Compared to Alternative Execution Strategies 56
5.3 Execution Time, Power Consumption, Energy Consumption, and ED^2 of Prediction-Based Adaptation Compared to Alternative Execution Strategies 58

6.1 The Echo Runtime System 61
6.2 Runtimes for Workloads When Using DVFS in Combination with All Policies on AMD Phenom (Normalized to No Power Envelope) 66
6.3 Runtimes for Workloads When Using DVFS in Combination with All Policies on Intel 8-Core (Normalized to No Power Envelope) 67
6.4 Runtimes for Mixture Workloads on the Intel 8-Core System (Normalized to No Power Envelope) 69
6.5 Energy for Mixture Workloads on the Intel 8-Core System (Normalized to No Power Envelope) 70
6.6 Cumulative Distribution Function (CDF) of Prediction Error for the Generalized Model 73
6.7 Runtimes for Mixture Workloads on the Intel 8-Core System with variation in DVFS (Normalized to No Power Envelope) 74

A.1 XEEMU XScale Simulator Results 89
A.2 Simulation Runtime for Modified Simulator Normalized to Original . . 90
A.3 Cumulative Distribution Function (CDF) of Prediction Error for the Generalized Model 91

B.1 Measured vs. Predicted Power for AMD Phenom 9500 92
B.2 Median Errors for AMD Phenom 9500 92

C.1 Measured vs. Predicted Power for AMD Phenom 9500 93
C.2 Median Errors for AMD Phenom 9500 93

D.1 Measured vs. Predicted Power for Dual Intel E5430 (8 Cores) 94
D.2 Median Errors for Dual Intel E5430 (8 Cores) 94

CHAPTER 1
INTRODUCTION

Power and thermal issues have become first-order constraints that limit performance and processor frequency. As a result, focus has shifted to chip multiprocessors (CMPs) to improve performance without pushing the power envelope. This trend is largely motivated by two observations: first, more performance is expected for a fixed transistor budget through on-chip, thread-level parallelism than through further exploitation of ILP; and second, the replication of less complex circuitry results in potentially more energy-efficient processors. As a result, chip manufacturers are producing multicore processors with a large number of cores per chip – or *many*-core processors. CMPs trade higher frequencies for more cores. Current predictions estimate CMPs with 10's to 100's of cores becoming available in the next decade [43], and Intel has already demonstrated a working prototype with 80 cores [51].

Multicore microprocessors represent an inflection point for software, since they rely on high numbers of parallel threads or processes to take full advantage of the cores available. When developing new software or optimizing current software for such a platform, energy efficiency is now a critical part of performance analysis. A further, often overlooked requirement is that software needs to scale gracefully with the number of cores, and threads need to interact with the hardware in non-destructive manners. If a multithreaded application is unable to take advantage of all cores provided by the processor, then either the application should be further parallelized and optimized to improve scalability on that particular architecture, or the cores should be allocated differently among the running programs, allocating cores to other programs that might need them, or leaving some cores idle to conserve energy. Given expected performance at different thread concurrency levels, the OS can scale the number of threads for a given multithreaded

program. If made aware of power consumption per process in a system, the OS can prioritize processes based on constraints on power and temperature. It can budget power per process, or schedule processes to remain under a given power envelope.

In Chapter 2, we propose a predictive approach for real-time per-core power estimation on a CMP. We use Performance Monitoring Counters (PMCs) to estimate power consumption of any processor via analytic models. Performance counters on chip are generally accurate [52](if used correctly), and they provide significant insight into the processor performance at the clock-cycle granularity. PMCs are already incorporated into and exposed to user space on most modern architectures. Accurately estimating real-time power consumption enables the OS to make better real-time scheduling decisions, administrators to accurately estimate the maximum number of usable threads for data centers, and simulators to accurately estimate power without actually simulating it. Additionally, a power meter is not required per system. Our analytic model can be queried on multiple systems regardless of the programs or inputs used. This is possible because our model uses microbenchmark data independent of program behavior. We write these microbenchmarks to gather PMC data that contribute to the power function. We use these data to form our power model equations. We thus estimate power for single threaded and multithreaded benchmark suites.

In Chapter 3, we leverage our power model to perform runtime, power-aware process scheduling. We suspend and resume processes based on power consumption, ensuring that a given power envelope is not exceeded. We propose and evaluate four scheduling policies and observe the resulting behavior. Estimating per-core power consumption is challenging, since some resources are shared across cores (such as caches, the DRAM memory controller and off-chip memory accesses).

In Chapter 4, we perform an in-depth analysis of the scalability of a set of multithreaded scientific applications that have already been extensively optimized for parallelism and locality. We perform our study on an Intel quad-core and an Intel eight-core CMP. Our findings indicate that while ample parallelism is available in the studied applications, threads interfere destructively for shared on-chip resources. This often results in negligible performance gains through the use of more than two cores, or even significant performance losses when concurrency exceeds some threshold. Somewhat surprisingly, poor scaling occurs even at just four cores, indicating that future many-core microprocessors may expose severe scaling limitations. Furthermore, we observe that the scalability of individual applications is *phase-sensitive*, in that different phases of the parallel code in an application exhibit radically different scaling properties. A phase is a user-defined region of parallel code encapsulating either a collection of parallel loops or a collection of basic blocks executed concurrently by multiple threads. Simultaneous with the performance consequences of poor scalability comes an increasing trend in power usage when using more cores. We propose and evaluate an ANN-based performance predictor to identify the desired level of concurrency and the optimal thread placement. The ANNs are trained offline to model the relationship among performance counter event rates observed while sampling short periods of program execution and the resulting performance with various levels of concurrency. The derived ANN models allow us to perform online performance prediction for phases of parallel code with low overhead by sampling performance counters.

In Chapter 5, we propose and evaluate an approach to throttle concurrency in parallel programs dynamically. The proposed infrastructure can detect program phases that may not scale well and determines the level of concurrency that will improve performance as well as efficient architecture-aware placement of threads onto specific processor cores for each phase. Concurrency throttling improves energy efficiency by virtue of higher

performance with sustained or reduced power consumption when processor cores are left idle. We identify dynamically more energy efficient concurrency levels and achieve higher performance with lower energy consumption in those parallel codes.

In Chapter 6, we propose a framework that combines both approaches. We implement an infrastructure that can schedule for power efficiency for a given power envelope, and/or a given thermal envelope. We target current systems, and present techniques for processors that support DVFS, as well as for those that do not. We schedule for better power efficiency by suspending or slowing down (DVFS) single-threaded programs, or throttling concurrency for multithreaded programs. We utilize the per-core power predictor to schedule applications to remain under a given power envelope. We modify the scheduler policies to take advantage of all power-saving approaches (DVFS, suspension, throttling). We discuss scalability to many-core platforms, and propose a generalized power model for systems with support for multiple levels of DVFS. Such an approach to dynamic power and energy management would serve well in current and emerging power-aware systems.

We present related work in Chapter 7. We summarize all chapters and discuss future work in Chapter 8. Overall, this thesis presents a framework for adaptive power management of single-threaded and multithreaded workloads. We present results for enforcing power envelopes with minimal loss in performance. However, the framework can be expanded to enforce energy or thermal constraints as well. With increasing focus on adaptive power management for multicore and many-core processors, this thesis presents practical techniques on real systems that can be vital to current and emerging power-aware systems.

CHAPTER 2

REAL-TIME PER-CORE POWER ESTIMATION FOR CMPs

Current infrastructures do not support runtime power measurement of a given core. We can use power meters to retrieve total system power only. System simulators provide in-depth information, but are extremely time consuming and prone to error. Obtaining such detailed simulators is difficult, since many are commercial, in-house, and available only to the computer architects. Current hardware can be enhanced to measure the current and power draw of a CPU socket, but per-core measurement is difficult because current CMP designs have all cores sharing the same power plane. Embedding measurement devices on-chip is not a feasible option either. The Intel Core i7 features per-core power monitoring at the chip-level but still does not expose this to the user [26].

We achieve per-core estimation with current infrastructure via Performance Monitoring Counters (PMCs). We estimate power consumption using analytic models formed using PMC data. Most modern architectures support PMCs on-chip and expose them to the user. PMCs are generally accurate (if used correctly) [52], and can provide data at the clock-cycle granularity. Given real-time power estimates, the OS can make better scheduling decisions, administrators can estimate the optimal number of threads for data centers to promote energy efficiency, and simulators can estimate power without power simulations. We use a power meter during model formation only. Our model is based on PMC data from microbenchmarks that are application independent. Our analytic model can be queried on multiple identical systems and can predict power for all programs or inputs used. We also account for core temperature. The PMC data and temperature form the variables in the power model equations.

Previous work has considered PMCs for power estimation of uniprocessors [32, 14]. We use real CMP hardware for per-core power, accounting for the impacts of temper-

ature on power. We estimate power for single threaded and multithreaded programs on two different quad-core platforms, an Intel Q6600, and an AMD Phenom 9500, and a dual-processor Intel E5430 quad-core platform with eight cores. We achieve median errors of 2.0%, 2.4%, and 3.5% for the SPEC-OMP, SPEC 2006, and NAS benchmark suites, respectively, on the Intel Q6600. NAS, SPEC 2006, and SPEC-OMP, show median error of 3.5%, 4.5%, and 5.2%, respectively, on the AMD Phenom platform. For the Intel E5430 eight-core platform, we obtain median errors of 2.8%, 3.5%, and 3.9%, for SPEC-OMP, SPEC 2006, and NAS, respectively. We achieve accurate per-core estimates of multithreaded and multiprogrammed workloads on CMPs with shared resources (L2/L3 caches, memory controller, memory channel, and communication buses). We achieve real-time power estimation, without the need for off-line benchmark profiling. Through the use of three different CMPs we demonstrate the portability of the approach to other platforms.

2.1 Methodology

We examine the processor dies in Figure 2.1 to find features that contribute to power consumption. For the AMD Phenom, the shared L3 and private L2 caches take up significant area. For the Intel Q6600, the L2 caches take up almost half the area. Thus, we expect cache miss counters to correlate with power consumption rate. For example, monitoring L2 misses on the AMD Phenom allows us to track use of the L3 caches, since L2 cache misses often result in L3 cache misses, which then lead to off-chip memory accesses. We find that the L3 cache has a large miss rate, since it is essentially a non-inclusive victim cache (Figure 2.2). Similarly, the floating point (FP) units and front-end logic comprise a significant portion of the die. Monitoring instructions retired and their type allows us to follow power consumption in the FP or INT units. Addition-

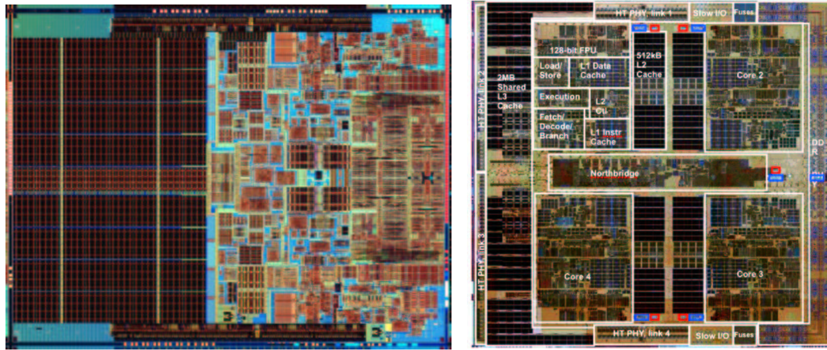


Figure 2.1: Die Photos for Intel Q6600 [2] (left), and AMD Phenom 9500 [3] (right)

ally, tracking instructions retired gives us an idea of the overall performance and power of the CMP. Since the Intel Q6600 is a high performance processor, we expect the out-of-order logic to contribute to the power consumption as well. Even though there is no counter that gives us this information directly, monitoring resource stall rates (stalls due to branches, full load/store queues, reorder buffers, reservation stations) can provide some insight. An increase in CPU stalls indicates stalled issue logic which means potentially reduced power consumption. Conversely, stalls in the reservation station or reorder buffer imply increased use of CPU logic (and power) to extract instruction-level parallelism. For example, if a fetched instruction stalls, the out-of-order logic tries to find another instruction to execute. It needs to examine more reservation stations to check for the new instruction's dependences and uses more dynamic power. The Intel E5430 processor layout is quite similar to that of the Intel Q6600. Based on our observations, we separate the PMCs into the smallest set that covers important contributions to power consumption and derive four categories: *FP Units*, *Memory Traffic*, *Processor Stalls*, and *Instructions Retired*.

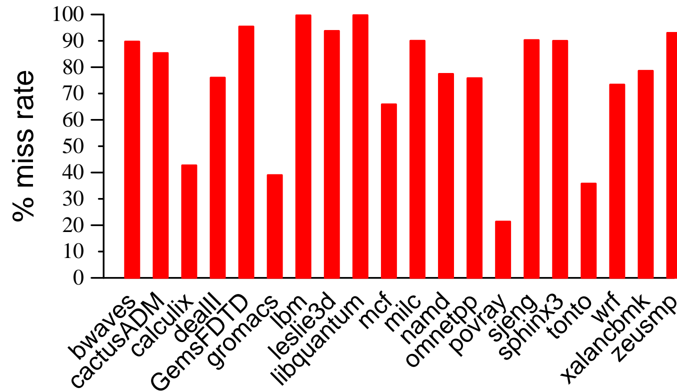


Figure 2.2: L3 Cache Miss Rates for SPEC 2006 on AMD Phenom

2.2 Microbenchmarks

We write our microbenchmarks to stress the four PMC categories we have derived. We do not use any code from our test benchmark suites, since the model needs to be application independent. We explore the space spanned by the four categories and attempt to cover common cases as well as extreme boundaries. The resulting counter values have large variations ranging from zero to several billion depending on the benchmark. For example, CPU-bound benchmarks have few cache misses, and integer benchmarks have few FP operations. The microbenchmarks are grouped by a large *for* loop and a *case* statement that branches to different code nests as we iterate through the loop index. The executed code consists of assign statements (moves), and arithmetic/FP operations. We compile with no optimization to prevent redundant code removal.

When collecting data, we run four copies of the microbenchmarks and collect the data for any single core since we find all cores to exhibit similar PMC data. Our code is generic and does not borrow from any of the benchmark suites we use for testing. We expect future application behavior to fall within the same space and we claim our approach to be independent of the benchmark suite. We test our approach on the NAS,

SPEC OMP, and SPEC 2006 benchmark suites. Since we employ an empirical process, the claim is backed by the quality of predictions.

2.3 Event Selection

We run our microbenchmarks and collect power and performance data for the PMCs that fall into our four categories: *FP Units*, *Memory Traffic*, *Processor Stalls*, and *Instructions Retired*. We use a Watts Up Pro power meter [22] to measure total system power and *pfmon* [23] to collect PMC data. The specific categories and the Phenom PMCs that fall within them are shown in Table 2.1. The PMCs in each category are in increasing order of correlation with power. The Intel Q6600 and the dual Intel E5430 counters are shown in Tables 2.2 and 2.3, respectively. We order the PMCs according to Spearman's rank correlation to measure the relationship between each counter and power. Spearman's correlation does not require assumptions about frequency distributions of variables. This is useful when forming the model in Section 2.5. Correlation can be positive or negative. We choose the top PMC from each category:

AMD Phenom 9500 – e_1: L2_CACHE_MISS:ALL, e_2: RETIRED_UOPS,
e_3: RETIRED_MMX_AND_FP_INSTRUCTIONS:ALL,
e_4: DISPATCH_STALLS

Intel Q6600 – e_1: L2_LINES_IN, e_2: UOPS_RETIRED,
e_3: X87_OPS_RETIRED, e_4: RESOURCE_STALLS

Dual Intel E5430 – e_1: LAST_LEVEL_CACHE_MISSES, e_2: UOPS_RETIRED,
e_3: X87_OPS_RETIRED, e_4: RESOURCE_STALLS

We claim that these four PMCs sufficiently predict core and system power in real-time. This is backed by results in Section 2.7. Next, we discuss the role of temperature in such a model, and we then form the model based on the data collected.

Table 2.1: PMCs Categorized by Architecture and Ordered (Increasing) by Correlation (for AMD Phenom 9500)

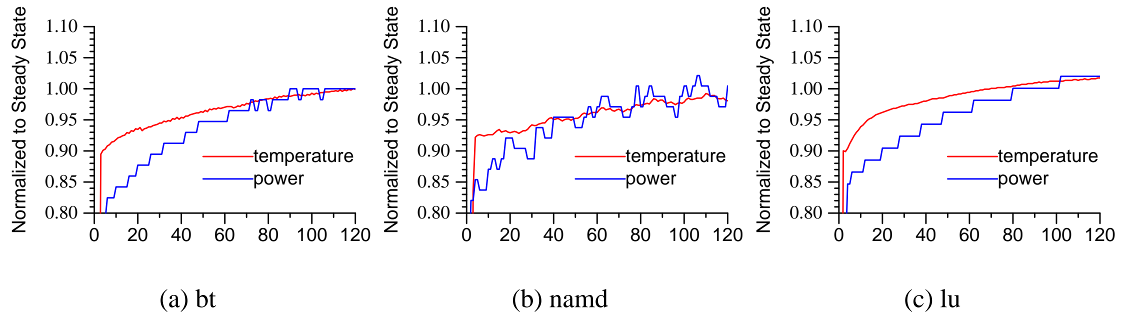
FP Units	DISPATCHED_FPU:ALL
0.23	RETIRED_MMX_AND_FP_INSTRUCTIONS:ALL
Inst Retired	RETIRED_BRANCH_INSTRUCTIONS:ALL
	RETIRED_MISPREDICTED_BRANCH_INSTRUCTIONS:ALL
	RETIRED_INSTRUCTIONS
0.39	RETIRED_UOPS
Stalls	DECODER_EMPTY
-0.20	DISPATCH_STALLS
Memory	DRAM_ACCESSES_PAGE:ALL
	DATA_CACHE_MISSES
	L3_CACHE_MISSES:ALL
	MEMORY_CONTROLLER_REQUESTS:ALL
0.33	L2_CACHE_MISS:ALL

Table 2.2: PMCs Categorized by Architecture and Ordered (Increasing) by Correlation (for Intel Q6600)

FP Units	SIMD_INSTR_RETIRED
0.11	X87_OPS_RETIRED:ANY
Inst Retired	MISPREDICTED_BRANCH_RETIRED
	BRANCH_INSTRUCTIONS_RETIRED
	INSTRUCTIONS_RETIRED
0.76	UOPS_RETIRED:ANY
Stalls	RAT_STALLS:ANY
	SNOOP_STALL_DRV:ALL_AGENTS
-0.38	RESOURCE_STALLS:ANY
Memory	LAST_LEVEL_CACHE_REFERENCES
	BUS_IO_WAIT:BOTH_CORES
	LAST_LEVEL_CACHE_MISSES
0.57	L2_LINES_IN:ANY

Table 2.3: PMCs Categorized by Architecture and Ordered (Increasing) by Correlation (for Dual Intel E5430)

FP Units	SIMD_INSTR_RETIRED
0.16	X87_OPS_RETIRED:ANY
Inst Retired	MISPREDICTED_BRANCH_RETIRED
	BRANCH_INSTRUCTIONS_RETIRED
	INSTRUCTIONS_RETIRED
0.68	UOPS_RETIRED:ANY
Stalls	SNOOP_STALL_DRV:ALL_AGENTS
	RAT_STALLS:ANY
-0.47	RESOURCE_STALLS:ANY
Memory	LAST_LEVEL_CACHE_REFERENCES
	BUS_IO_WAIT:BOTH_CORES
	L2_LINES_IN:ANY
0.48	LAST_LEVEL_CACHE_MISSES



(a) bt

(b) namd

(c) lu

Figure 2.3: Power vs. Temperature on the AMD 4-Core CMP

2.4 Temperature Effects

We are interested in the effect of temperature on system power. Ideally, power consumption does not increase over time. However, since static power is a function of voltage, process technology, and temperature, increasing temperature leads to increasing leakage power, and adds to total power. We concurrently monitor the temperature and power of the CMP to see their relationship. Figure 2.3 graphs temperature (in Celsius) and power consumption (in watts) over time. Results are normalized to their steady-state values. Benchmarks *bt*, *lu* and *namd* are run across all four cores of the CMP, with results capped at 120 seconds. For *namd*, four instances are run concurrently since it is single-threaded. Performance counters and program source code are examined to ensure the work performed is constant over time. The programs exhibit varying increases in power and temperature over time. Clearly, temperature and power affect each other. Not accounting for temperature could lead to increased error in power estimates. However, like the AMD Phenom, not all CMPs support per-core temperature sensors. We use chip temperature readings for the AMD Phenom, and per-core readings for the Intel Q6600. We believe availability of per-core temperature sensors on the Intel platform helps improve prediction accuracy.

2.5 Forming the Model

We form our model based on the collected microbenchmark data. We normalize the PMC to the elapsed cycle count and get an *event rate*, r_i, for each counter. The prediction model uses these rates and rise in core temperature, T, as input. We collect PMC values and temperature every second. We model per-core power using our piece-wise model based on multiple linear regression. We produce the following function (Equation 2.1), mapping rise in core temperature T and observed event rates r_i to core power P_{core}:

$$\hat{P}_{core} = \begin{cases} F_1(g_1(r_1), ..., g_n(r_n), T), & if \text{ condition} \\ F_2(g_1(r_1), ..., g_n(r_n), T), & else \end{cases} \quad (2.1)$$
$$\text{where} \quad r_i = e_i/(cycle\ count)$$

$$F_n = p_0 + p_1 * g_1(r_1) + ... + p_n * g_n(r_n) + p_{n+1} * T \quad (2.2)$$

The function consists of linear weights on transformations of event rates (Equation 2.2). The transformations can be linear, inverse, logarithmic, or square root. They make the data more amenable to linear regression and help prediction accuracy. We choose a piece-wise linear function because we observe significantly different behavior for low PMC values. This allows us to keep the simplicity of linear regression and capture more detail about the core power function. For example, were we to form a model for the data in Figure 2.4(a), we would find that neither a linear nor exponential transformation fits the data. However, were we to break the data into two parts, we would find a piece-wise combination of the two fits much better, as in Figure 2.4(b). We determine weights for function parameters using a least squares estimator as in Contreras et al. [14]. Each part of the piece-wise function is a linear combination of transformed event rates (Equation 2.2).

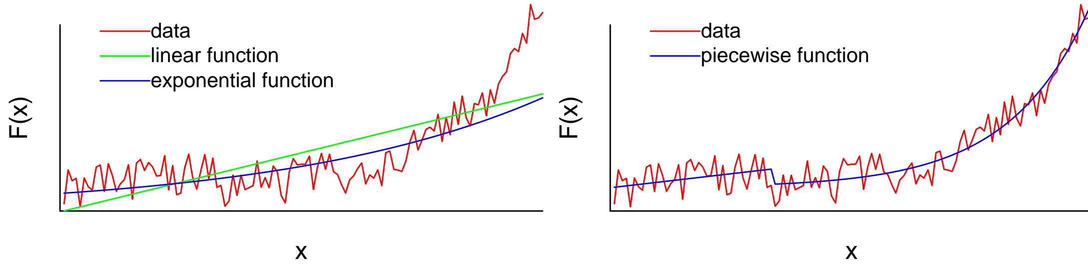


Figure 2.4: An Illustrative Example of Best-Fit Continuous Approximation Functions (left), and a Better Fitting Piece-Wise Function (right)

$$\hat{P}_{core} = \begin{cases} 7.699 + 0.026 * log(r_1) + 8.458 * r_2 + -3.642 * r_3 + 14.085 * r_4 + 0.183 * T, \\ \hfill r_1 < 10^{-6} \\ 5.863 + 0.114 * log(r_1) + 1.952 * r_2 + -1.648 * r_3 + 0.110 * log(r_4) + 1.044 * T, \\ \hfill r_1 \geq 10^{-6} \end{cases} \tag{2.3}$$

$$where \quad r_i = e_i/2200000000 \ (1s = 2.2B \ cycles)$$

For the AMD Phenom, we obtain the piece-wise linear model shown in Equation 2.3. We find the function behavior to be significantly different for very low values of the L2 cache miss counter compared to the rest of the space. We break our function based on this counter. Since the L3 is non-inclusive, most L2 misses trigger off-chip accesses contributing to total power. We also observe that the power grows with increasing retired uops, since the CPU is doing more work. All counters have positive correlation with power, except for the retired FP/MMX instructions PMC. This is expected, since such instructions have higher latencies; this class of instructions reduces the throughput of the system, resulting in lower power use. The dispatch stalls PMC correlates positively with power. This can be due to reservation stations or reorder buffer dispatch stalls, where the processor attempts to extract higher degrees of instruction level parallelism

(ILP) from the code. Dynamic power increases from this logic overhead. Finally, we observe a positive correlation between temperature and core power. This is expected since increase in temperature leads to increase in leakage power, and adds to total power.

$$\hat{P}_{core} = \begin{cases} 5.280 + -0.132 * log(r_1) + 3.993 * r_2 + -0.882 * r_3 + 4.419 * r_4 + 0.338 * T, \\ \hfill r_1 < 10^{-6} \\ 14.653 + 0.128 * log(r_1) + 1.563 * r_2 + -3.885 * r_3 + 0.284 * log(r_4) + 0.342 * T, \\ \hfill r_1 \geq 10^{-6} \end{cases} \tag{2.4}$$

$$where \quad r_i = e_i/2400000000 \; (1s = 2.4B \; cycles)$$

We obtain the piece-wise linear model shown in Equation 2.4 for the Intel Q6600. Here we also find the behavior of the L2 cache miss counter (L2_LINES_IN) to differ for very low values and break our function based on it. We observe that the power consumption is higher as more instructions are committed. The FP counter has negative correlation with power since such instructions have higher latencies; this class of instructions reduces the throughput of the system, resulting in lower power use. Power increases with more resource stalls. This can be the result of increased dynamic power consumption from logic overhead of extracting higher instruction level parallelism (ILP) from the code. Finally, we find a positive correlation between temperature and core power. Higher temperature leads to increased leakage power, and adds to the total power.

For the eight-core system, as before, we study the space, this time finding that the floating point counter is the best candidate for deciding where to split the data. We use the last level cache miss counter for the *Memory Traffic* category since it shows higher correlation. This differs from the quad-core models above. The rest of the counters are the same and exhibit similar relationships with power as the quad-core models. The eight-core piece-wise linear model is given in Equation 2.5.

$$
\hat{P}_{core} = \begin{cases} 4.227 + 0.035 * log(r_1) + 0.816 * r_2 + -1.747 * r_3 + 3.506 * r_4 + 0.673 * T, \\ \hfill r_3 < 10^{-6} \\ 10.799 + 0.003 * log(r_1) + 0.703 * r_2 + 0.030 * log(r_3) + 1.412 * r_4 + 0.360 * T, \\ \hfill r_3 \geq 10^{-6} \end{cases} \tag{2.5}
$$

$$
where \quad r_i = e_i/2670000000 \; (1s = 2.67B \; cycles)
$$

2.6 Experimental Setup

We evaluate our predictions using the SPEC 2006 [47], SPEC-OMP [6], and NAS [8] benchmark suites. We run all benchmarks to completion. We use *gcc* 4.2 to compile our benchmarks for a 64-bit architecture, using default optimization flags as specified in each suite. Our software platform consists of Linux kernel version 2.6.27, and the *pfmon* utility from the *perfmon2* library [23] to access hardware performance counters from user space. Table 2.4 details the system configuration for the AMD Phenom platform. This CMP supports one temperature sensor. The other two Intel platforms have four cores and eight cores, respectively, with full system details outlined in Table 2.5. Both CMPs have temperatures sensors on each core. We use the *sensors* utility from the *lm-sensors* library to obtain core temperature. We use a Watts Up Pro power meter [22] to gather power data. Our meter is accurate to within 0.1W, and updates once per second. The resolution of our predictions is one second to match the power meter, in order to verify measured vs. predicted power. We write a library that takes input from *pfmon* and *sensors*, and predicts power every second using the models derived in Section 2.5. The software using this library runs concurrently on the core that runs the OS, and contributes negligible overhead. System power is based on the processor being

Table 2.4: AMD Phenom 9500 Machine Configuration Parameters

Frequency	2.2 GHz
Process Technology	65 nm
Processor	AMD Phenom 9500 CMP
Number of Cores	4
L1 (Instruction) Size	64 KB 2-Way Set Associative
L1 (Data) Size	64 KB 2-Way Set Associative
L2 Cache Size (Private)	512 KB/core 8-Way Set Associative
L3 Cache Size (Shared)	2 MB 32-Way Set Associative
Memory Controller	Integrated On-Chip
Memory Width	64 bits/channel
Memory Channels	2
Main Memory	4 GB DDR2-800

idle, and measured by the power supply's current draw from the outlet. We measure the idle processor temperature to be 36C for both the AMD and Intel quad-core platforms. We measure the idle system power to be 84.1W for the AMD Phenom, and 141W for the Intel Q6600. We subtract the idle processor power of 20.1W [4] from the AMD idle system power to obtain an *uncore* (baseline without processor) power of 64W. This is used as an estimate of the base power consumption for the rest of the system (other than the CMP). Similarly, for the Intel machine, we subtract the idle processor power of 38W [1] to obtain an *uncore* power of 103W. We find idle processor temperature for the eight-core system to be 45C, with an *uncore* power of 122W. Changes in the *uncore* power itself (due to DRAM or hard drive accesses, e.g.) is included in the model predictions. Including temperature as an input to the model accounts for variation in *uncore* static power. We use the *uncore* power as the baseline when calculating per-core power. This assumption aids in faster model formation without the need for more complicated measuring techniques. We calculate per-core power by subtracting the *uncore* power and dividing by the number of cores in the CMP.

Our hardware performance counters have some limitations. One issue is the Intel platform does not concurrently support sampling of more than two general performance counters. *pfmon* supports time-splicing where one counter is measured half the time,

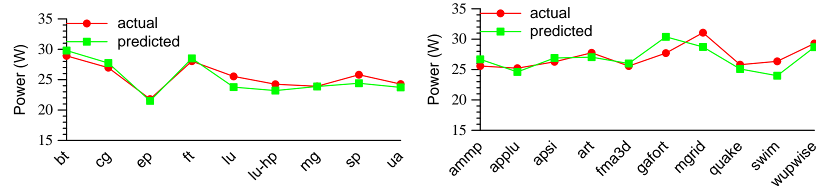
Table 2.5: Intel Q6600 and Dual Intel E5430 Machine Configuration Parameters

Machine	4-Core	8-Core
Frequency	2.4 GHz	2.0 GHz, 2.66 GHz
Process Technology	65 nm	45 nm
Processor	Intel Q6600 CMP	Intel Xeon E5430 CMP
Number of Cores	4, dual dual-core	8, dual quad-core
L1 (Instruction) Size	32 KB 8-Way Set Associative	32 KB 8-Way Set Associative
L1 (Data) Size	32 KB 8-Way Set Associative	32 KB 8-Way Set Associative
L2 Cache Size (Shared)	4 MB 16-Way Set Associative	6 MB 16-Way Set Associative
Memory Controller	Off-Chip, 2 channel	Off-Chip, 4 channel
Main Memory	4 GB DDR2-800	8 GB DDR2-800
Front Side Bus	1066 MHz	1333 MHz

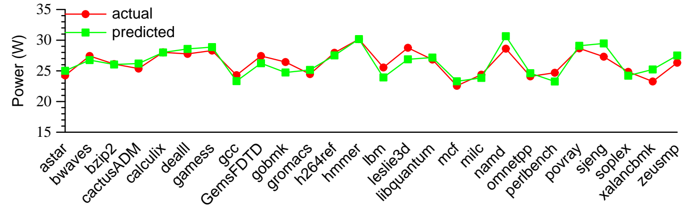
and its value is estimated as it would be for the whole time. This allows us to sample the four counters we need. The AMD processor can sample four counters simultaneously. Additionally, some statistics are only provided for the entire CMP and not for each individual core. Some PMCs could be further subdivided by type. For example, cache and DRAM accesses can be broken down into cache or page hits and misses, while dispatch stalls can be broken down by branch flushes, or full queues (reservation stations, reorder buffers, FP units).

2.7 Evaluation

We evaluate the accuracy of our power model using single and multithreaded benchmarks, using the entire CMP to test our results. We test our derived power model by comparing measured to predicted power in Figures 2.5(a), 2.5(b), 2.5(c) (AMD quad-core), Figures 2.7(a), 2.7(b), 2.7(c) (Intel quad-core), and Figures 2.9(a), 2.9(b), 2.9(c) (Intel eight-core) for NAS, SPEC-OMP, and SPEC 2006, respectively. Each multi-threaded benchmark is run across the entire CMP, and multiple copies are spawned for single-threaded programs. For single-threaded benchmarks, activity observed per core is similar, but this is not always the case for multithreaded benchmarks. We therefore

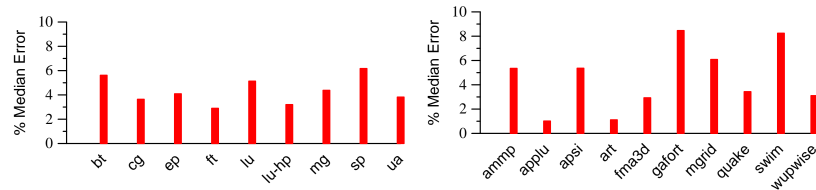


(a) NAS (b) SPEC-OMP

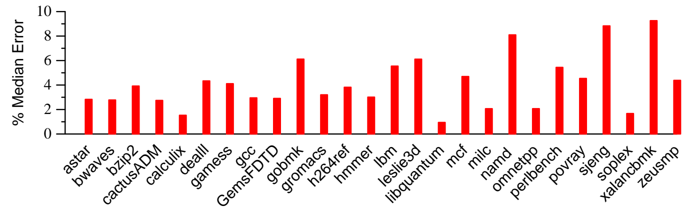


(c) SPEC 2006

Figure 2.5: Measured vs. Predicted Power for AMD Phenom 9500

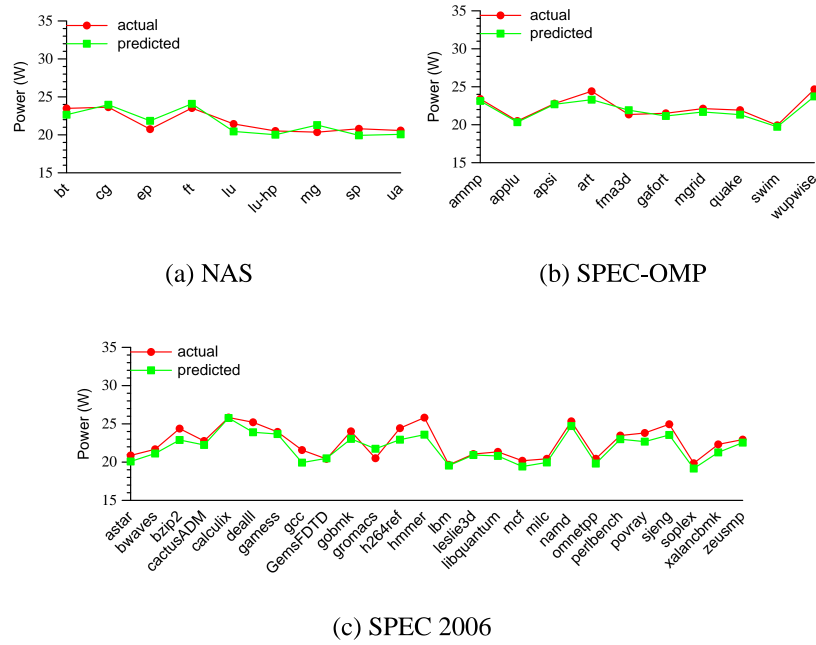


(a) NAS (b) SPEC-OMP



(c) SPEC 2006

Figure 2.6: Median Errors for AMD Phenom 9500

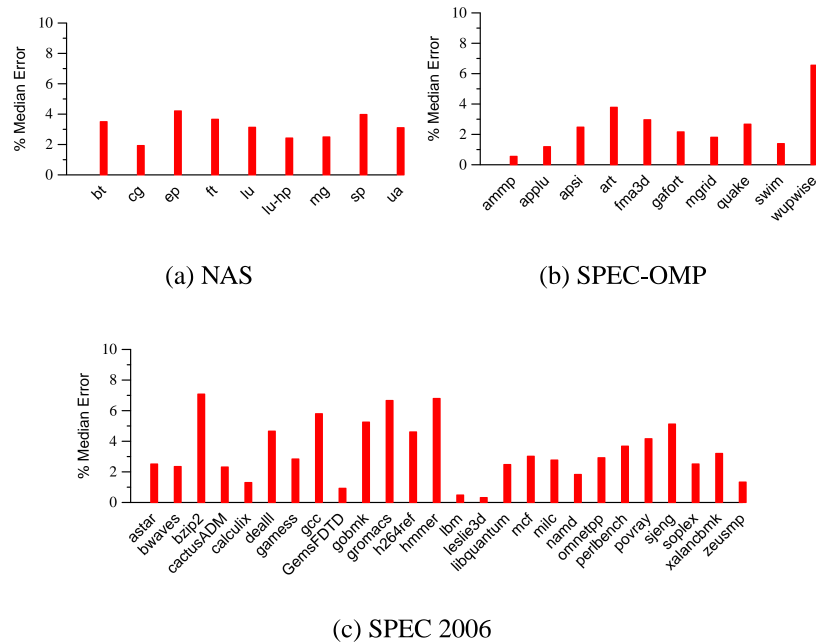


(a) NAS

(b) SPEC-OMP

(c) SPEC 2006

Figure 2.7: Measured vs. Predicted Power for Intel Q6600

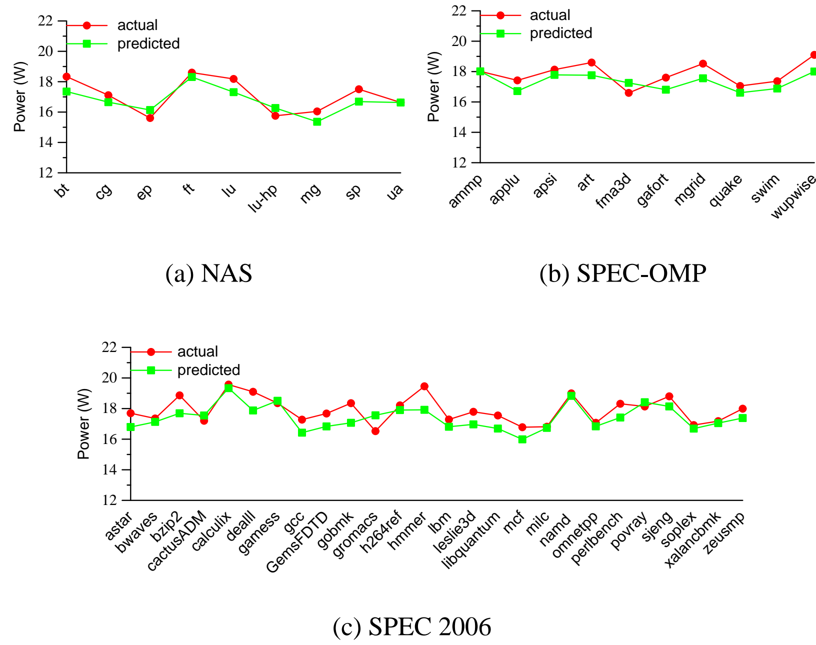


(a) NAS

(b) SPEC-OMP

(c) SPEC 2006

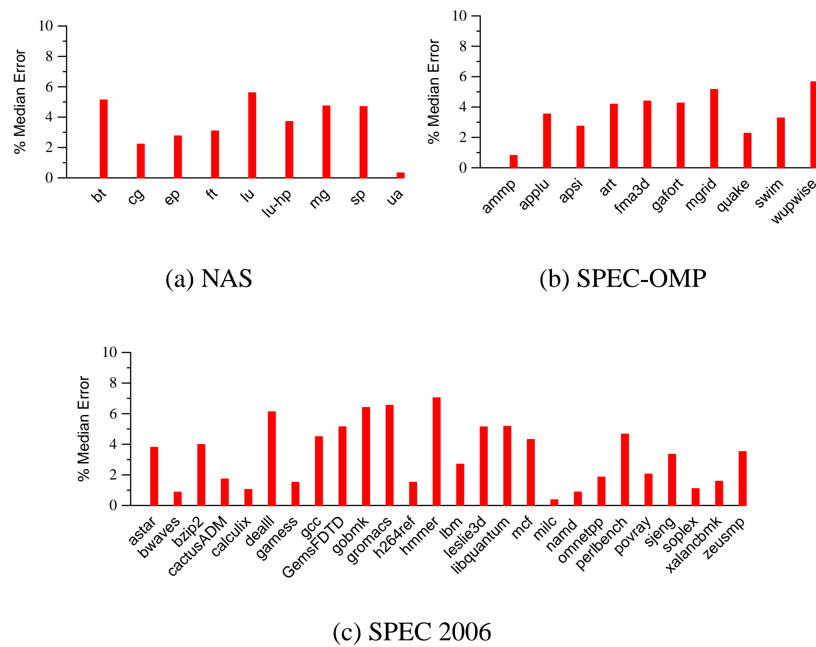
Figure 2.8: Median Errors for Intel Q6600



(a) NAS (b) SPEC-OMP

(c) SPEC 2006

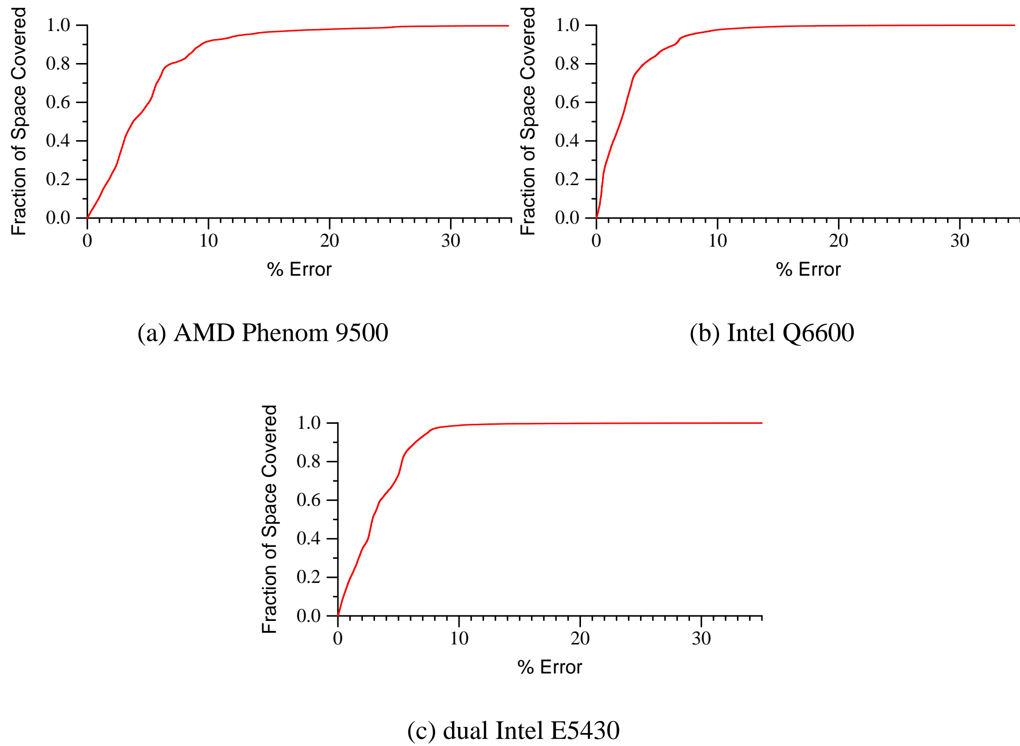
Figure 2.9: Measured vs. Predicted Power for Dual Intel E5430 (8 cores)



(a) NAS (b) SPEC-OMP

(c) SPEC 2006

Figure 2.10: Median Errors for Dual Intel E5430 (8 cores)



(a) AMD Phenom 9500

(b) Intel Q6600

(c) dual Intel E5430

Figure 2.11: Cumulative Distribution Function (CDF) Plot Showing Fraction of Space Predicted (y-axis) under a Given Error (x-axis)

account for error on all cores. Data are calculated per core, with error reported across all cores. Our estimation model tracks power consumption for each benchmark fairly well. Figures 2.6(a), 2.6(b), and 2.6(c) (AMD quad-core), Figures 2.8(a), 2.8(b), and 2.8(c) (Intel quad-core), and Figures 2.10(a), 2.10(b), and 2.10(c) (Intel eight-core) show percentage error for each suite. For the Intel quad-core machine, the prediction error ranges from 0.3% for *leslie3d* to 7.1% for *bzip2*. The eight-core system shows similar prediction error range from 0.3% (*ua*) to 7.0% (*hmmer*). For the AMD machine, he prediction error ranges from 0.9% for *libquantum* to 9.3% for *xalancbmk*. For the Intel Q6600, SPEC-OMP and SPEC 2006 have median error of 2.0% and 2.4%, respectively. NAS has slightly higher median error of 3.5%. NAS, SPEC 2006, and SPEC-OMP, show median error of 3.5%, 4.5%, and 5.2%, respectively, on the AMD Phenom platform. The

model for the eight-core system shows slightly higher median errors of 2.8%, 3.5%, and 3.9%, for SPEC-OMP, SPEC 2006, and NAS, respectively.

Figure 2.11 shows the Cumulative Distribution Function (CDF) for all three benchmark suites taken together, for each platform. This gives us a picture of the coverage of our model. For example, on the AMD quad-core platform, 92% of predictions across all benchmarks have less than 10% error. For the Intel quad-core platform, 85% of predictions across all benchmarks have less than 5% error and 97.5% of predictions show less than 10% error. 98.7% of all predictions on the Intel eight-core system show less than 10% error. When temperature is excluded, only 85% of predictions have less than 10% error. The CDF helps illustrate the model's fits, showing that most predictions have very small error. We attribute error in power estimates to parts of the counter space possibly unexplored by our microbenchmarks. We lower prediction error for outliers (e.g., *namd*, *sjeng*, and *xalancbmk* on the AMD quad-core) when we train on their power data, in addition to the microbenchmark data.

We use three different CMP platforms and obtain accurate per-core power estimates for the NAS, SPEC 2006, and SPEC-OMP benchmark suites. As a result, we demonstrate the portability of the approach. The models are independent of our test benchmarks. We achieve median error of 3.8% on an AMD quad-core CMP, 2.0% on an Intel quad-core CMP, and 2.8% on an Intel eight-core CMP.

CHAPTER 3

POWER-AWARE THREAD SCHEDULING

We present an application that uses the power predictor derived in Chapter 2 to schedule processes dynamically such that they run under a fixed power envelope (similar to a power manager proposed by Isci et al. [29]). We write four user-space schedulers (in C) that spawn a process on each core of the CMP, and monitors their behavior via *pfmon*. Figure 3.1 illustrates its setup and use. The processes are bound to a particular core and do not migrate to other cores during the course of execution. The program makes real-time predictions for per-core and system power based on collected performance counters, and suspends processes as the power envelope is breached. We implement four scheduling policies to choose a candidate for suspension. We consider three sets of multiprogrammed workloads, and collect data on the AMD Phenom, the Intel Q6600, and the dual Intel E5430 from Chapter 2.

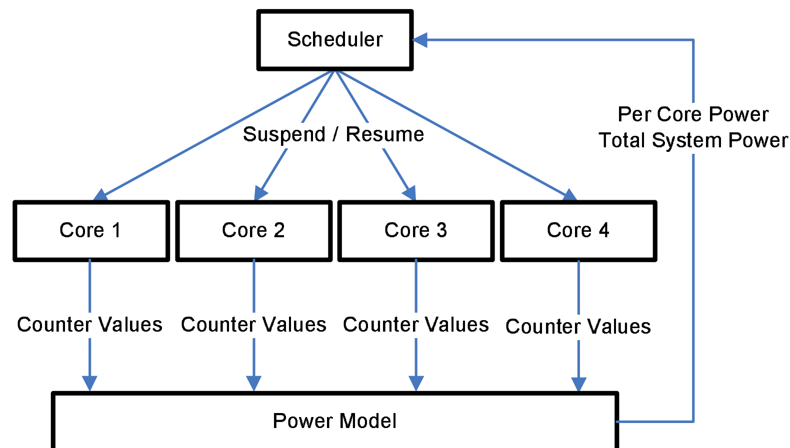


Figure 3.1: Scheduler Setup and Use

3.1 Simple Policy

This policy implements a blanket envelope on power consumption. It suspends the processes such that system power is just below the power envelope. For example, assume that current system power is 190W and the power envelope is 180W. For simplicity, we have to choose between two processes consuming 20W and 25W, respectively. The scheduler suspends the first process to bring system power down to 170W, rather than choosing the second process and being further away from the envelope (at 165W). When resuming a process, it again considers the process that pushes power consumption closest to the given envelope.

3.2 Maximum Instructions/Watt Policy

This policy attempts to achieve the most power efficiency under the given power envelope. When the envelope is breached, it suspends the process with the least instructions committed-per-watt. The instruction-to-power ratio is recorded at suspension for consideration later. When considering a process to resume from the suspended pool, it awakens the process with the most instructions committed-per-watt that remains under the envelope. Such a policy generally gives the best performance compared to others.

3.3 Per-Core Fair Policy

This policy is designed to give each core a fair share of the consumed energy. It maintains a running average of the power consumed by each core at a given time. On exceeding the power envelope, it suspends the process with the highest average consumed

power (or energy). The running average is updated constantly, and when it drops low enough the process is considered for resumption. The process with the lowest average that remains under the power envelope is awakened from the suspended pool. Such a policy can help regulate core temperature since it throttles cores with high power consumption. Since static power is a function of voltage, process technology and temperature, increasing temperature leads to increasing leakage power, and adds to total power. The temperature difference between cores is much smaller compared to other policies.

3.4 User-based Priorities Policy

This policy takes input from the user of the scheduler and considers process priority when suspending processes to remain under the envelope. For example, assume that current system power is 190W and the power envelope is 180W. For simplicity, we have to choose between two processes consuming 20W and 25W, respectively. The first process has higher priority than the second. The scheduler suspends the second process even though suspending the first would have been closer to the power envelope. When resuming a process, it again considers the process priority and resumes the highest priority process that remains under the envelope. Such a situation is desirable when the user has outside knowledge (e.g, runtime, phase behavior) that results in better performance. For example, it would be faster to give high priority to a short-running power-hungry process and get it out the way so that the rest of the processes can easily run under the power envelope.

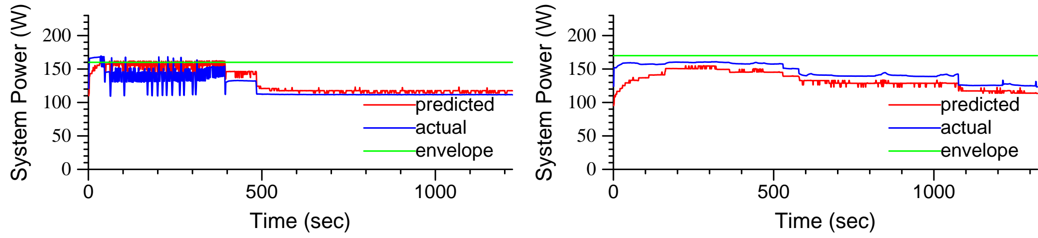
Table 3.1: Multiprogrammed Workloads for Evaluation

Benchmark Set	4-Core	8-Core
CPU-bound	ep, gamess, namd, povray	calculix, ep, gamess, gromacs, h264ref, namd, perlbench, povray
Average	art, lu, wupwise, xalancbmk	bwaves, cactusADM, fma3d, gcc, leslie3d, sp, ua, xalancbmk
Memory-bound	astar, mcf, milc, soplex	applu, astar, lbm, mcf, milc, omnetpp, soplex, swim

3.5 Evaluation

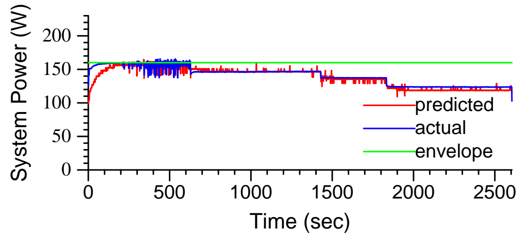
We leverage real-time power estimation to make power-aware scheduling decisions, suspending processes to maintain a given power envelope. We propose and evaluate four different scheduling policies and observe the resulting behavior. We use the power predicted for processes to schedule them within a multiprogrammed workload on the CMP. We run experiments on the AMD Phenom and the Intel Q6600 for a four-process multiprogrammed workload, and on the Intel eight-core machine for an eight-process multiprogrammed workload. We suspend processes to remain below the system power envelope. For these experiments, we assume the system power envelope to be degraded by 5, 10, or 15%. The runtimes are compared against running without a power envelope, and the envelope is then degraded from 5-15% of the workload's peak power usage. Lower envelopes render one or more cores inactive and the workload executes only three processes or fewer. We do not consider them in this work. If required, the scheduler can follow lower envelopes easily. We consider three sets of multiprogrammed workloads with varying degrees of *CPU intensity* (Table 3.1). We define *CPU intensity* as the ratio of instructions retired to last-level cache misses. The first set contains a multiprogrammed workload with the highest *CPU intensity* (CPU-bound), the second takes the benchmarks that exhibit average *CPU intensity* (Average), and the third contains the benchmarks with the lowest *CPU intensity* (Memory-bound).

Figures 3.2 and 3.3 show some representative examples of different policies and power envelopes for our quad-core platforms. We observe *measured* and *predicted*

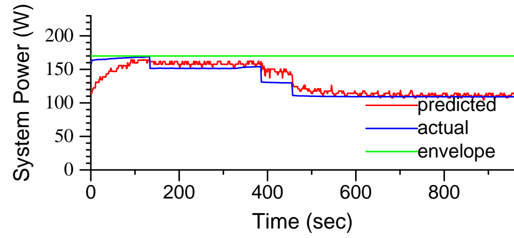


(a) CPU-bound, Simple, 90%

(b) Mem-bound, User-based Priorities, 95%

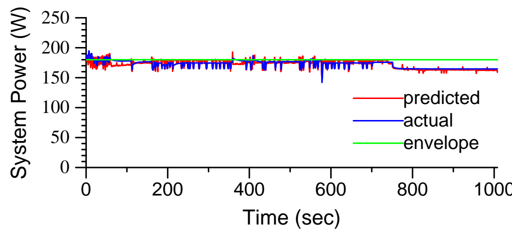


(c) Average, Max Inst/Watt, 90%

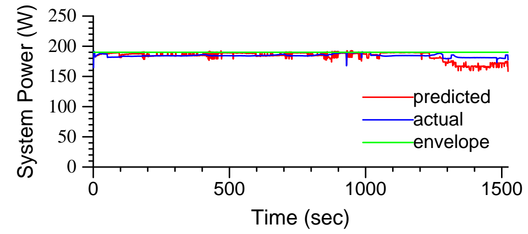


(d) CPU-bound, Per-Core Fair, 95%

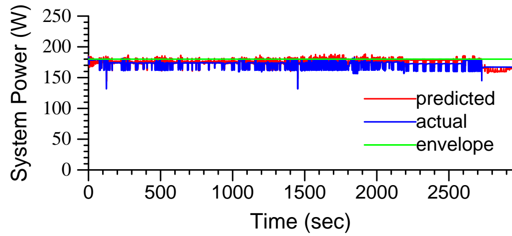
Figure 3.2: Given Workloads, Policies, and Envelopes for AMD Phenom



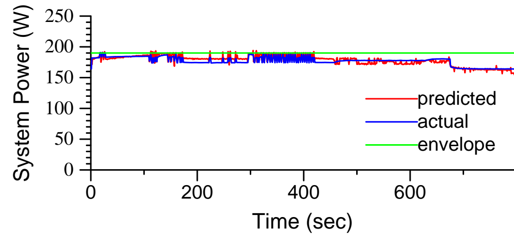
(a) CPU-bound, Simple, 90%



(b) Mem-bound, User-based Priorities, 95%

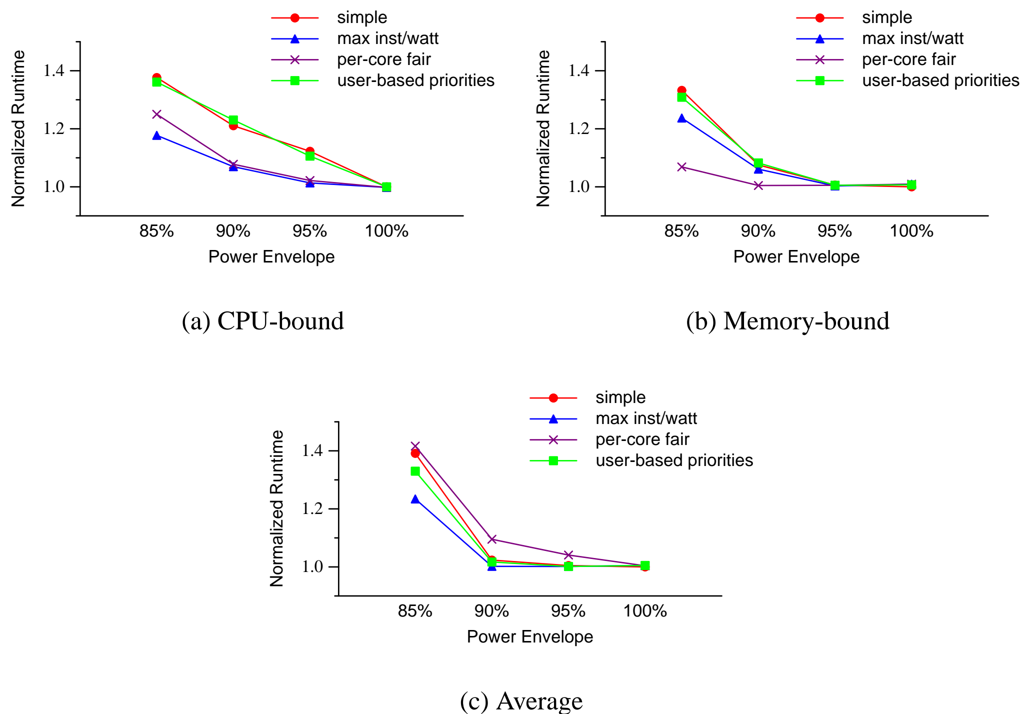


(c) Average, Max Inst/Watt, 90%



(d) CPU-bound, Per-Core Fair, 95%

Figure 3.3: Given Workloads, Policies, and Envelopes for Intel Q6600



(a) CPU-bound

(b) Memory-bound

(c) Average

Figure 3.4: Runtimes for Workloads on AMD Phenom (Normalized to No Power Envelope)

power match up well. We are able to follow the power envelope strictly, and do so entirely on the basis of our prediction-based scheduler. This obviates the need for a power meter, and would be an excellent tool for software-level control of per-core and system power.

Each of the policies is effective in completion of the workload under the given power envelope, with varying degrees of performance loss. First, we analyze the results from the AMD Phenom machine. Figure 3.4 exhibits normalized runtimes for the complete set of policies and power envelopes. For the CPU-bound workload in Figure 3.4(a), the *per-core fair* policy and the *max inst/watt* are both quite optimal and preserve performance. Both slow down the workload by about 7% at the 85% envelope mark. The *simple* and *user-based priorities* policies extend workload runtime by 37% with an 85%

envelope. For the memory-bound workload (Figure 3.4(b)), *per-core* fair beats all other policies. Since all benchmarks in this workload are memory intensive, and memory accesses take power, this policy regulates the bandwidth within the workload as a side-effect of regulating power per core. There is less contention on the bus, and they each execute faster. While the *max inst/watt* policy does better than the other two remaining policies, its goal of maximum throughput does not work in synergy with the high memory contention among the processes. In Figure 3.4(c), the *per-core fair* policy does not fare well. Its goal of regulating power per process is not necessarily the best optimal performance policy. The average workload is best executed with the *max inst/watt* policy. The performance loss is minimal for the 90% and 95% power envelopes, but quite significant (23%) at the 85% mark. At this point, some process is always under suspension and this lengthens the workload runtime. It is akin to running three processes together, and the remaining one after. This happens because the power envelope is too low to allow the applications in the workload to progress together. A solution to this problem would be to use dynamic voltage/frequency scaling. The same workload run in Section 3.6 at the 85% power envelope shows only a 2% performance loss.

Next, we analyze the results from the Intel Q6600 system. Figure 3.5 exhibits normalized runtimes for the complete set of policies and power envelopes. For the CPU-bound workload in Figure 3.5(a), the *max inst/watt* policy achieves the best performance, and the *user-based priorities* policy shows the worst. For the memory-bound workload (Figure 3.5(b)), performance improves by 2.6% and 5.8% for the 90% and 95% power envelopes, respectively. Without any power envelope, all processes compete for cache and memory bandwidth. When the power envelopes come into play, they throttle the processes to conserve power, and in the process also free cache and memory bandwidth, which helps speed up the execution of the running processes. This effect is not observed with the 85% envelope because now, even though there is less competition among pro-

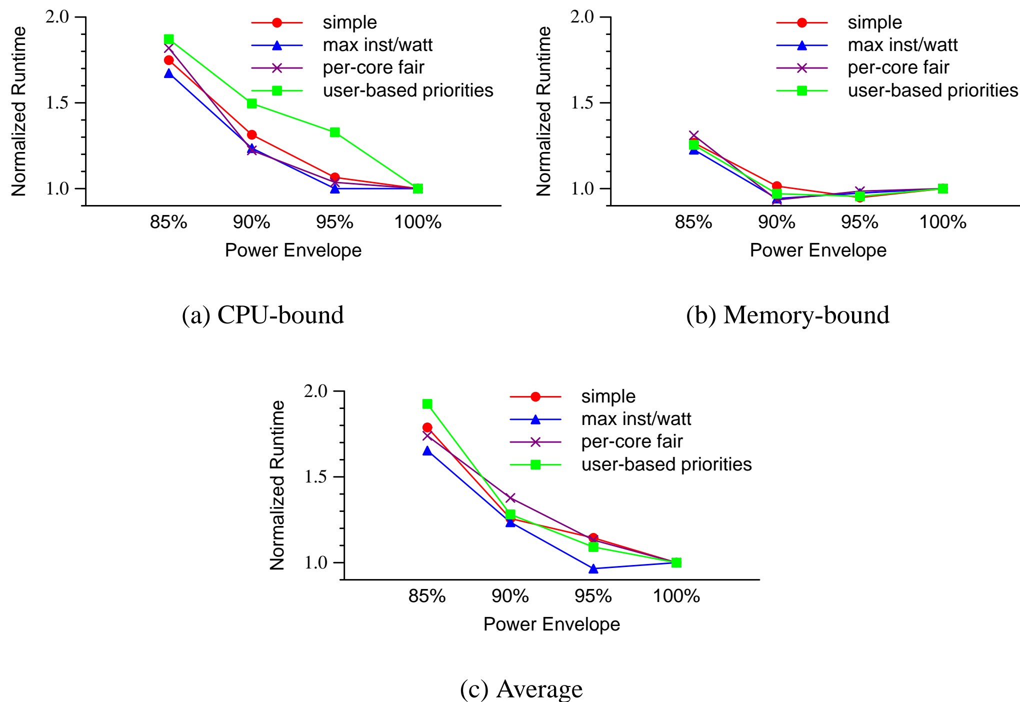
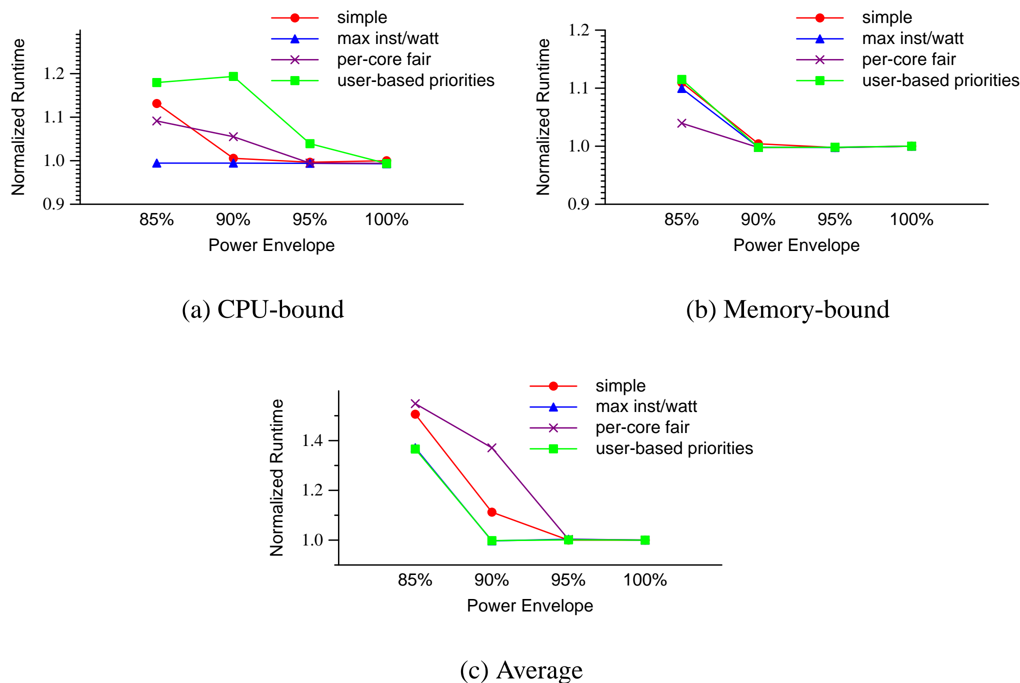


Figure 3.5: Runtimes for Workloads on Intel Q6600 (Normalized to No Power Envelope)

cesses, they cannot run at maximum speed, since they may breach the power envelope. The average workload in Figure 3.5(c) exhibits the most variation with policy, and *max inst/watt* achieves the best performance. We see behavior similar to the memory-bound workload for the 95% power envelope, but the effect diminishes as we decrease the envelope. The 85% envelope shows performance loss similar to when run on the AMD quad-core. Performance loss varies as the envelope is reduced, and shows that the choice of policy depends not only on the workload but on the given power envelope, as well.

The set of experiments on the eight-core system is more interesting, since we deal with eight programs on eight cores. There is more potential for saving power through suspension, and possibly less loss of performance. Again, each of the policies is effective in completion of the workload under the given power envelope, with varying degrees of performance loss. The performance loss is less than the quad-core set of experiments



(a) CPU-bound

(b) Memory-bound

(c) Average

Figure 3.6: Runtimes for Workloads on Dual Intel E5430 (Normalized to No Power Envelope)

for the CPU and memory bound workloads, and comparable for the average workload Figure 3.6 exhibits normalized runtimes for the complete set of policies and power envelopes on the eight-core system. For the CPU-bound workload in Figure 3.6(a), the *max inst/watt* policy preserves performance consistently, while the *user-based priorities* policy performs the worst. For the memory-bound workload, performance improves marginally (0.2%) for the 90% and 95% power envelopes, respectively. This behavior is similar to that observed for the quad-core memory-bound workload. Throttling processes to conserve power frees up cache and memory bandwidth which helps speed up the execution of the running processes. The effect is not as profound as the quad-core case because there are more processes involved, and the contention is high even if a couple of processes are suspended. The 85% envelope shows minimal loss for the *per-core fair* policy. The average workload in Figure 3.6(c) exhibits the most variation with

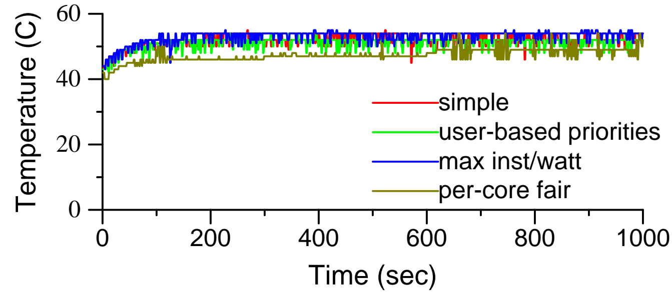


Figure 3.7: Temperature Across Policies for a Sample Workload

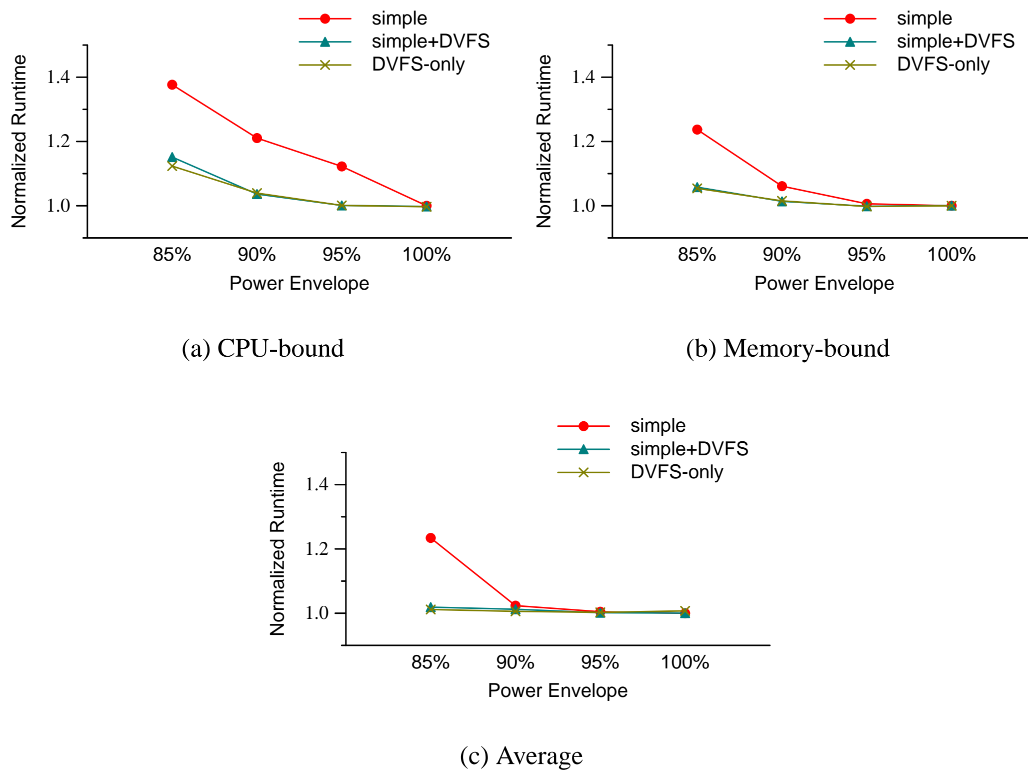
policy and *max inst/watt* achieves the best performance along with *user-based priorities*. This is a good example of how prior knowledge can assist performance, if correctly supplied via the policy. We see behavior similar to the memory-bound workload for the 90% and 95% power envelopes. Again, we observe that performance does not necessarily decrease with a decrease in the power envelope. This elucidates that the choice of policy depends on the workload as well as the power envelope.

A few more observations are noteworthy. The *simple* policy, as the name suggests, does not account for anything other than staying under the envelope. Therefore, the performance varies widely since it is not considered as a criterion when scheduling threads. The *per-core fair* policy regulates temperature as a side-effect of giving equal power over time to each core (Figure 3.7). For our workloads, *max inst/watt* generally gives best performance out of all the policies except in one case. For the *user-based priorities* policy, we have a fixed priority for each process based on the core to which it is bound. Core 0 is given the highest priority, while core 3 has the lowest. Processes are bound to cores in the order they appear in Table 3.1 and do not migrate during the course of workload execution. The performance under this policy varies widely with the workload. Choice of user-based priorities can greatly affect the performance, and can be useful when the user has insight into the workload itself.

3.6 What about DVFS?

An alternative to suspending processes to reduce power is to use dynamic voltage/frequency scaling (DVFS). For processors that support DVFS, it would generally be more energy efficient to scale the voltage or frequency of a core (as available) than to suspend the process. One advantage of scaling down a core is the drop in static power consumed. This drop could go towards executing the thread, albeit at a slower pace. This means we have more tolerance for higher dynamic power before reaching the given envelope. Additionally, it helps keep core temperature down in case of a thermal envelope. However, it is possible that such scaling might harm performance if there is a lot of contention for resources. In such cases, suspending one of the cores might actually speed up execution. The second advantage of DVFS is the faster switching time compared to suspension (100s vs. 10,000s of clock cycles). DVFS is a hardware-level feature, while the operating system is responsible for suspending and resuming the given process.

Not every processor offers the ability to perform DVFS. Of our two platforms, the AMD Phenom offers per-core dynamic voltage and frequency scaling between 1.1 GHz and 2.2 GHz. We form a power model for the machine running at 1.1 GHz with prediction error shown in Appendix C. We implement two more policies in our scheduler, *DVFS-only* and *simple+DVFS*. The *DVFS-only* policy replaces suspension in the *simple* policy, choosing to scale frequency for a core that brings the power closest to the envelope. This policy would not work in case of a particularly low envelope since even running all four cores at 1.1 GHz might still breach the power envelope. To counter this, we implement a *simple+DVFS* policy that chooses between DVFS and suspension depending on which one comes closest to the power envelope. We use the same envelopes as in Section 3.5.



(a) CPU-bound

(b) Memory-bound

(c) Average

Figure 3.8: Runtimes for Workloads when using DVFS in combination with *simple* on AMD Phenom (Normalized to No Power Envelope)

In Figure 3.8, we show the runtimes for the *simple*, *DVFS-only*, and *simple+DVFS* policies. They are all normalized to runtime with no power envelope. As expected, the *simple* policy lags behind the other two. For the memory-bound and average workloads, the *DVFS-only* and *simple+DVFS* perform similarly. For the CPU-bound workload, *DVFS-only* outperforms *simple+DVFS* marginally. This happens because suspending a CPU-bound process would affect its runtime much more than suspending a memory-bound process. DVFS allows for forward progress while suspension does not. This does not occur for the memory bound and average workloads because even when given the option to execute on a slower core, they do not progress much while waiting on memory. We explore DVFS briefly here, and perform more experiments with all four policies and a wider range of workloads in Chapter 6, where we examine our full framework.

CHAPTER 4

MULTITHREADED SCALABILITY AND PREDICTING CONCURRENCY

Processor vendors are providing increasing degrees of parallelism within a single chip. As a result, the scalability of multithreaded applications becomes a critical issue. Processors containing tens or even hundreds of cores will likely be available within the next decade [43], but whether modern scientific applications can capitalize on the parallelism afforded by these architectures is an open question. Given current trends with respect to number of cores on chip, we must consider the practical scalability and energy efficiency of representative applications for next-generation systems. We first present results showing that more concurrency is not always helpful, then we explain a method by which we can predict appropriate thread configurations for better performance and energy efficiency.

We present the performance impact and energy efficiency analysis of using additional cores for a range of parallel applications from the scientific domain. We use an Intel Q6600 quad-core and a dual-processor Intel E5320 quad-core platform as shown in Table 4.1. They are by no means many-core processors, but our experimental analysis indicates that scalability bottlenecks exist for many applications, even at such a small scale. The first machine has a single Intel quad-core processor. There are two 4 MB L2 caches, each shared between two of the cores. The second platform has two quad-core processors. Each pair of cores shares L2 cache. We refer to the two cores sharing a single L2 cache as tightly coupled, and cores not sharing a cache as loosely coupled.

In our evaluations, we use benchmarks from the NAS Parallel Benchmark suite version 3.2 [31] to represent modern scientific applications. The codes are implemented in either C or Fortran, have been parallelized using OpenMP, and have been extensively optimized for parallelism and locality [31]. We execute them under various levels of

Table 4.1: Machine Configuration Parameters

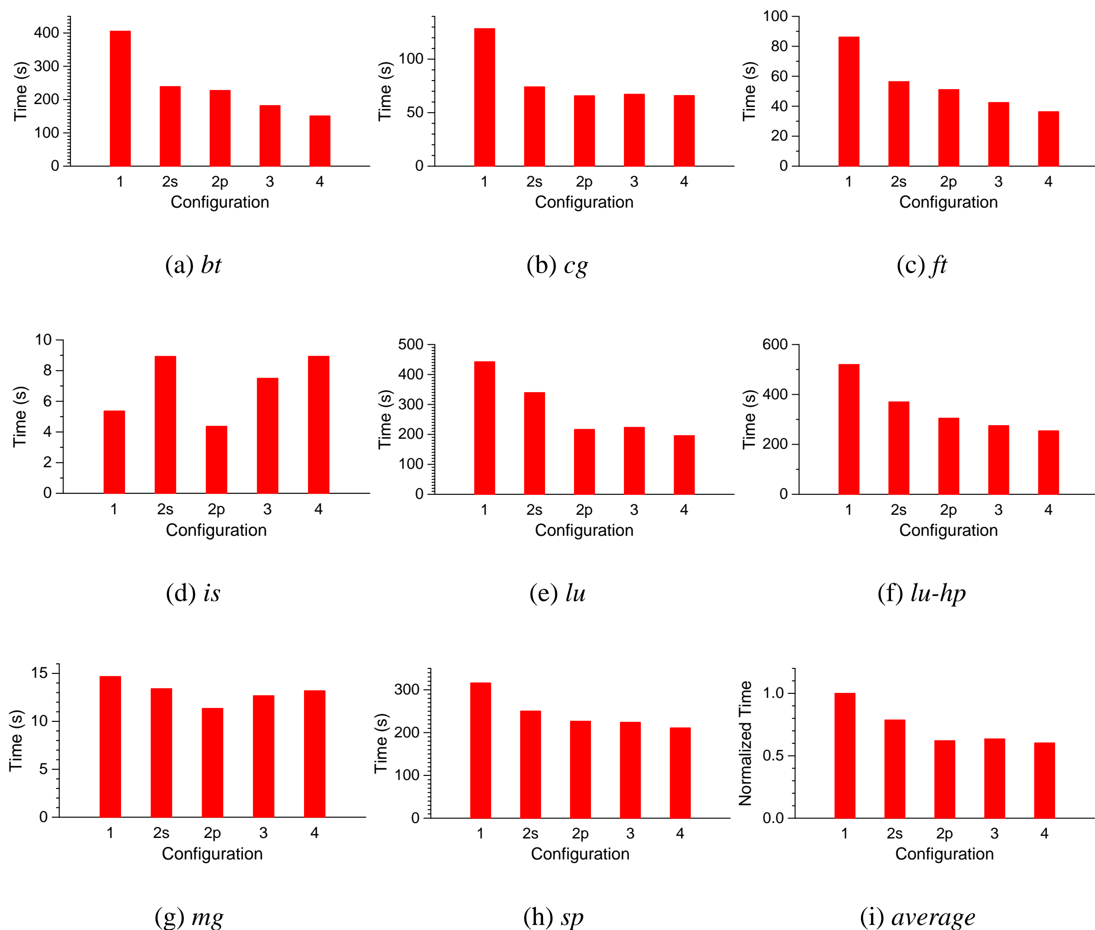
Machine	4-Core	8-Core
Frequency	2.4 GHz	1.86 GHz
Process Technology	65 nm	45 nm
Processor	Intel Q6600 CMP	Intel Xeon E5320 CMP
Number of Cores	4, dual dual-core	8, dual quad-core
L1 (Instruction) Size	32 KB 8-Way Set Associative	32 KB 8-Way Set Associative
L1 (Data) Size	32 KB 8-Way Set Associative	32 KB 8-Way Set Associative
L2 Cache Size (Shared)	4 MB 16-Way Set Associative	4 MB 16-Way Set Associative
Memory Controller	Off-Chip, 2 channel	Off-Chip, 4 channel
Main Memory	2 GB DDR2-800	4 GB DDR2-800
Front Side Bus	1066 MHz	1066 MHz

concurrency and under specific bindings of the threads to cores, performing experiments with five different thread configurations for the quad-core system: first, a single thread bound to a single core (configuration 1), two threads bound to two tightly coupled cores (configuration 2s (shared)), two threads running on two loosely coupled cores (configuration 2p (private)), three threads (configuration 3), and four threads running on all four cores (configuration 4). For the eight-core system, the notation (P, C) indicates execution using P processors and C cores per processor.

4.1 Analysis of Application Scalability: Four Cores

Figure 4.1 displays the execution times of our experiments. Many applications fail to scale beyond two threads executing on loosely coupled cores. In fact, of the eight benchmarks, only three (*bt*, *ft*, *lu-hp*) obtain substantial gains with the use of additional processor cores. The remaining benchmarks fall into two categories: those whose scalability curves flatten after two cores, and those who see large performance losses when using more cores. We examine each class of applications in turn.

The three applications that scale well are interesting because they show what can be achieved on this architecture. The fact that applications can improve their perfor-



(a) *bt* (b) *cg* (c) *ft*

(d) *is* (e) *lu* (f) *lu-hp*

(g) *mg* (h) *sp* (i) *average*

Figure 4.1: Execution times by Hardware Configuration (the bottom-right graph shows the average normalized execution time across all benchmarks)

mance through the use of each additional core demonstrates that scaling is not inherently limited on this quad-core system. However, applications might not scale due to the interaction with the underlying architecture. This group may provide insight into the types of program behavior that are amenable to multicore execution. Averaged over this application class, we observe a speedup of 2.37x compared to the sequential executions.

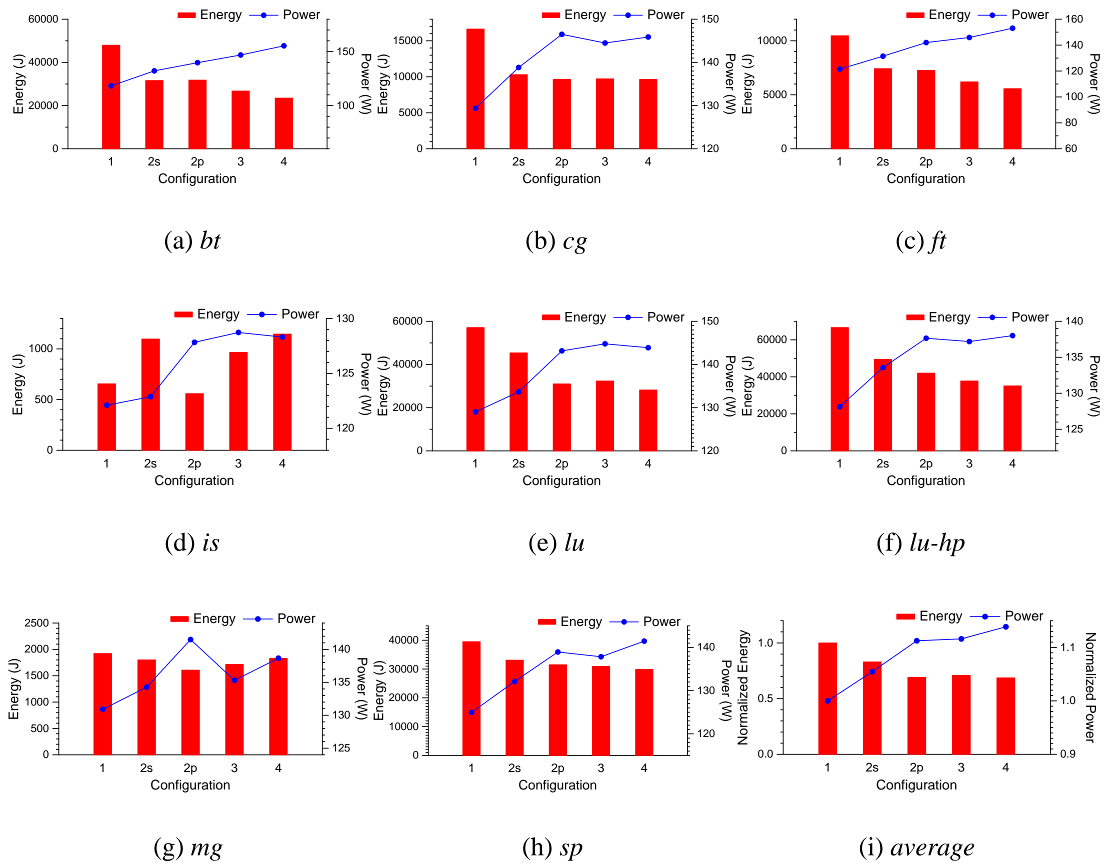
The second group of applications sees little performance gain or loss executing on more than two cores (*cg*, *lu*, and *sp*). Specifically, *cg* speeds up by 1.95x when using all four processor cores, however achieves the same speedup with only two threads when

executed on loosely coupled cores. Overall, this class of applications shows only a 7.0% average performance improvement from using four cores compared to two.

The final group of applications, with substantial performance losses through the use of more processor cores, provides the most interesting results. Both *mg* and *is* perform best with two threads on loosely coupled cores. The performance of *mg* with four threads is 11.3% faster than sequential execution, however *mg* with two threads is 14.0% faster than sequential execution. In contrast, *is* is extremely communication-intensive and bandwidth sensitive. The benchmark runs at a 40.0% performance loss using four threads compared to one, but its performance improves by 22.8% using two threads. The two-thread execution of *is* on loosely coupled cores is 2.04x faster than on tightly coupled cores, which suggests that the destructive interference in the shared L2, and the resulting memory bandwidth saturation, is largely to blame for the poor scalability of *is* on this machine.

Of all benchmarks, effective scaling only occurs up to two cores, with additional cores providing little to no gain. These results suggest that this architecture is not well suited for applications from the scientific domain. The poor scalability in these experiments is not an artifact of outdated systems, since we obtain results on a state-of-the-art system. If next-generation processors contain as many cores as generally expected, and the needs of scientific applications are not addressed, then the increased concurrency will likely lead to even poorer scalability than that observed here. Next, we address the power properties of the experimental platform and analyze the consequences of poor scalability on the resulting energy efficiency.

Figure 4.2 presents power and energy characteristics of our benchmarks (note that the y-axis does not begin at zero). For the five runs over that we measure execution times, we also collect energy consumption data using a Watts Up Pro power meter. We



(a) *bt* (b) *cg* (c) *ft*

(d) *is* (e) *lu* (f) *lu-hp*

(g) *mg* (h) *sp* (i) *average*

Figure 4.2: Power and Energy Consumption by Hardware Configuration (the bottom-right graphs shows the geometric mean of the normalized energy and power consumption across all benchmarks)

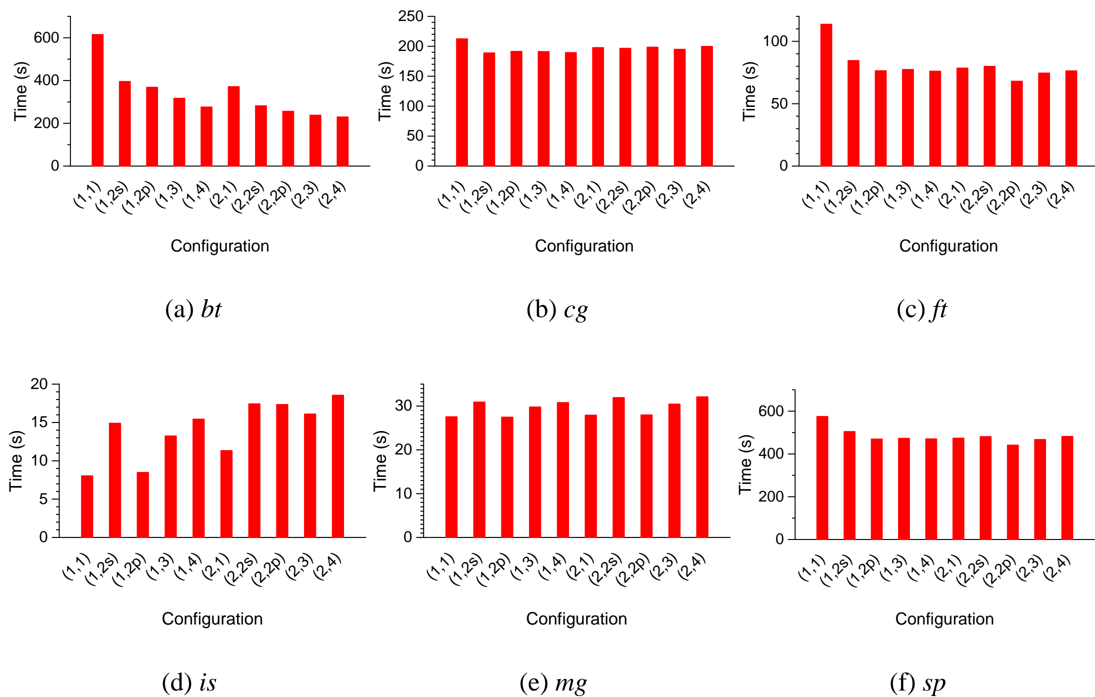
compute average power for each application using recorded execution time and energy consumption. Numbers reported here represent a full system power profile, including CPU, memory, power supply, and other components.

We confirm that using more cores leads to higher power consumption. Total system power consumed on four cores is 14.2% higher than on one core, as expected. Higher utilization with more concurrency will generally increase power, but the same contention responsible for poor scaling observed above reduces power consumption in several cases. This indicates that cores and other processor components remain idle for

extended time intervals. In such cases, measuring total system energy consumption during execution provides insight into whether throttling cores (i.e., decreasing number of threads) benefits both execution time and energy.

Applications that scale best show the largest increases in power consumption with more cores, while those applications that scale worst show negligible change in power (even power reductions). Consider *bt*, which achieves a 2.69x speedup on four cores with an associated 1.31x increase in power, the largest of any application, in both respects. However, a 2.04x decrease in energy consumption illustrates the potential energy efficiency of multicore architectures. For scalable applications, the performance increase is much greater than the power increase, and energy efficiency improves on more cores. On the other hand, *mg* performs best on two loosely coupled cores with a 1.29x speedup, which also represents its highest power thread configuration. The minimal decrease in power of 2.1% on four cores is dwarfed by the 18.1% increase in execution time, so the resulting energy efficiency on four cores drops considerably. *is* is 2.04x faster on configuration 2b than on configuration 4, and consumes slightly less power on fewer cores. These poorly scalable applications demonstrate the potential loss in energy efficiency when using all available cores. Applications with flat scalability curves simply fail to achieve increases in energy efficiency on this architecture.

Taken together, the applications show a minor decrease of 0.7% (geometric mean) in energy consumption scaling to four cores. Future generation systems with many cores will be further prone to scalability limitations, as applications will have to scale to more threads on architectures with a reduced compute-to-cache ratio [43].



(a) *bt* (b) *cg* (c) *ft*

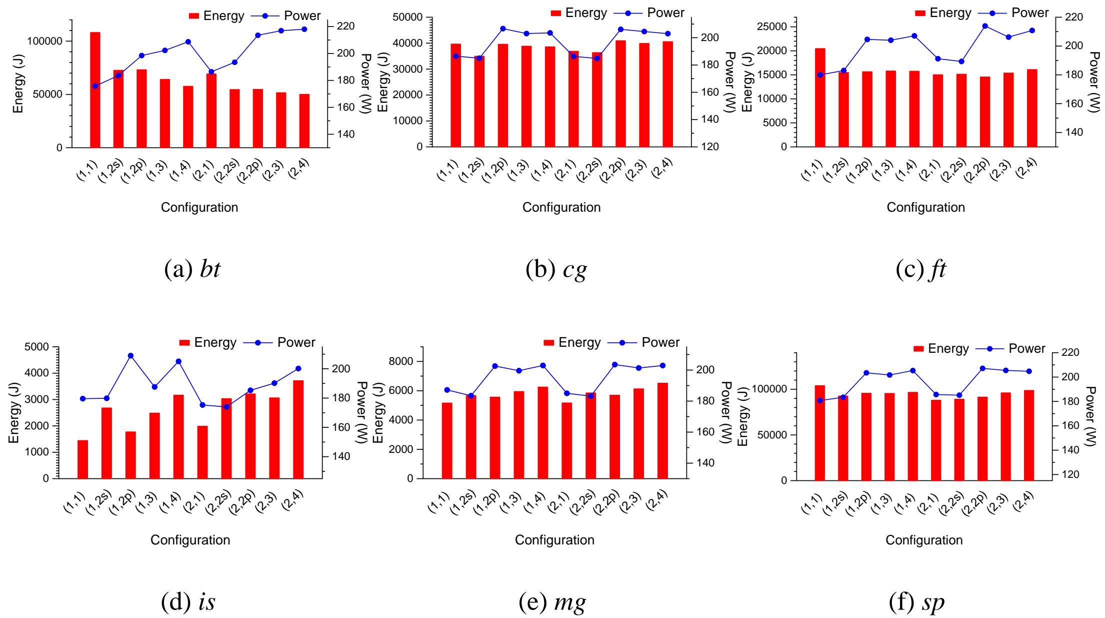
(d) *is* (e) *mg* (f) *sp*

Figure 4.3: Execution Times by Hardware Configuration

4.2 Analysis of Application Scalability: Eight Cores

Figure 4.1 shows the execution times of our experiments. We find that scalability is limited in most cases. Of the six benchmarks, only one (*bt*) scales to the number of cores available. As with the quad-core system, the remaining benchmarks fall into two categories: those whose scalability curves flatten after two cores, and those who suffer significant performance losses when using more cores. We examine each class.

bt illustrates that scalability on the quad-core processors is not inherently limited. It exhibits a high ratio of computation to memory activity. This application consumes least energy at full concurrency, because scaling achieves higher performance with incrementally higher power.



(a) *bt* (b) *cg* (c) *ft*

(d) *is* (e) *mg* (f) *sp*

Figure 4.4: Power and Energy Consumption by Hardware Configuration

cg, *ft*, and *sp* exhibit limited performance scaling from additional concurrency, and speedups plateau when threads are mapped across chip boundaries due to inter-chip communication latencies. These benchmarks have a mean speedup of 24.9% when using all cores. *ft* and *sp* obtain speedups of 42.2% and 19.5%, respectively, from the four cores of a single processor, but see minimal benefit from using the second processor. For these benchmarks, using fewer cores reduces energy consumption without sacrificing performance.

When run using all cores, *is* and *mg* slow down by 2.31x and 1.17x, respectively, due to memory intensity and limited memory bandwidth on our platform. Furthermore, *is* observes a 31.5% performance improvement when the entire cache is allocated to a single core, compared to sharing the cache between two cores. This is due to destructive interference in the shared L2 causing memory bandwidth saturation and poor scalability. *is* and *mg* both consume minimal energy using only a single thread, with additional concurrency increasing energy consumption by 157.1% and 26.3%, respectively.

Collectively, the benchmarks slow down by 4.7% (geometric mean) when scaled to maximum concurrency. Total system power consumption increases by 13.9% due to increased resource utilization. These effects combine to yield an average increase in energy consumption of 17.6%. Although multicore architectures are being marketed as an energy-efficient solution, clearly the efficiency in practice depends heavily on the scalability of a given application on a particular architecture. If multicore processors are to be adopted for use in the HPC arena, either the system will need to be improved for the known properties of HPC applications, or the applications themselves will need to be re-engineered for better performance on multicore architectures. The most energy-efficient configuration coincides with the most performance-efficient configuration for four out of the six benchmarks (*bt*, *cg*, *ft*, and *is*). For two benchmarks (*mg* and *sp*), we use fewer than the performance-optimal number of cores, to achieve substantial energy savings, at a marginal performance loss. For a given number of threads, performance can be very sensitive to the mapping of threads to cores (e.g. *bt*, *ft*, and *sp* when executed with two or four threads). Even if performance is insensitive to the mapping of threads to cores, power can be sensitive to these mappings. In *mg*, for example, distributing two threads across two sockets on the same die is less performance-efficient, but significantly more energy-efficient than distributing two threads between two dies.

4.3 Predicting Concurrency

Sections 4.1 and 4.2 demonstrate improved performance using fewer cores. This is due to limited scalability of several parallel execution phases. Execution properties are not static within an application [45]: many exhibit phased behavior, such that program characteristics vary at repeating intervals. In our test cases, program phases exhibit widely varying scalability and energy efficiency characteristics, even within a single applica-

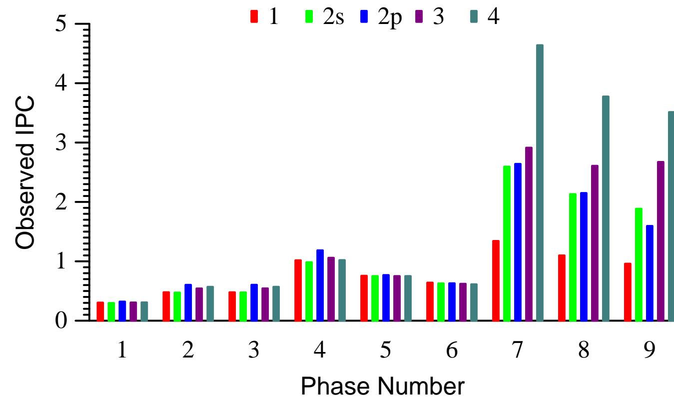


Figure 4.5: IPCs observed during Phases of *sp* for each Thread Configuration on the 4-core System

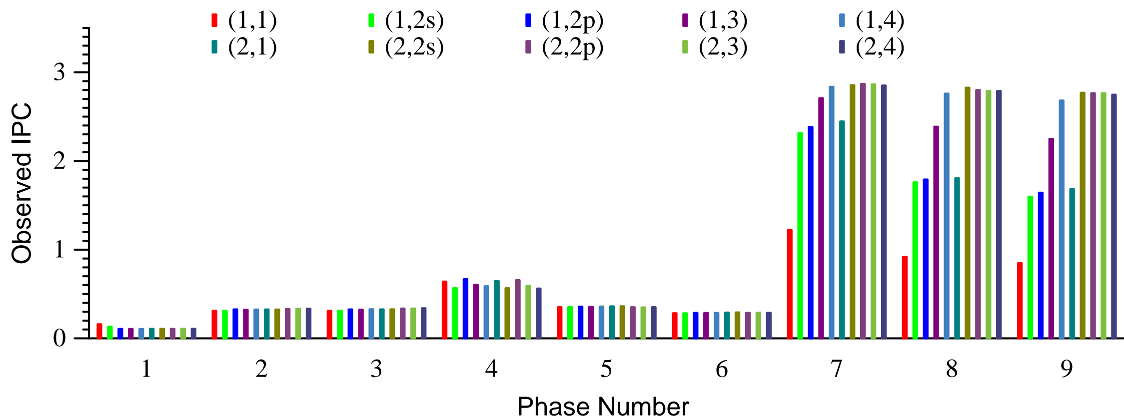


Figure 4.6: IPCs observed during Phases of *sp* for each Thread Configuration on the 8-core System

tion. This includes phases with collective operations that force processor serialization, phases that incur contention for shared on-chip or off-chip resources, and phases with inherently limited algorithmic concurrency. For example, Figure 4.5 presents IPCs for each phase of the *sp* application when executing on each thread configuration on the quad-core system. The graph demonstrates variations, with the maximum IPC for each phase ranging from 0.32 to 4.64, and the best performances coming on all configurations except those with three threads. On the eight-core system, the *sp* benchmark (Figure 4.6) contains phases that perform best at six distinct configurations, with full

concurrency yielding speedups ranging from 0.68x to 3.24x. We only show results for *sp* due to space limitations, but this diversity occurs for other benchmarks in similar proportions. Thus, the best configuration for any given program phase may differ from surrounding phases. Identifying poorly scalable phases at runtime could support dynamic concurrency throttling that executes each phase with a more efficient thread configuration. This motivates us to perform adaptation at the phase granularity, allowing for potentially better performance than any single configuration.

4.4 Overview of Artificial Neural Networks

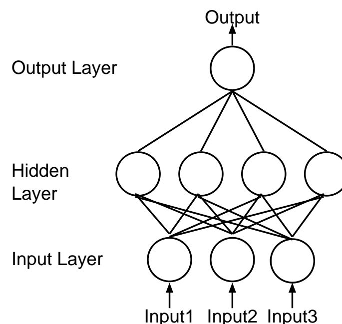


Figure 4.7: Simplified Diagram of a Fully Connected, Feed-Forward ANN

Machine learning studies algorithms that *learn* automatically through experience. For our problem, we focus on a particular class of machine learning algorithms called *artificial neural networks* (ANNs). Their many previous uses include microarchitectural design space exploration [27] [50], workload characterization [55], and compiler optimization [20]. ANNs automatically learn to predict one or more targets (here, IPC) for a given set of inputs. We choose ANNs because they are flexible and well suited for generalized nonlinear regression, and their representational power is rich enough to express complex interactions among variables: any function can be approximated to arbitrary

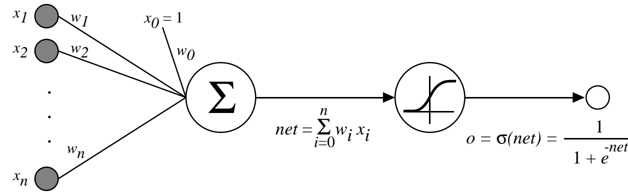


Figure 4.8: Example of a Hidden Unit with a Sigmoid Activation Function

precision by a three-layer ANN [40]. They require no knowledge of the target function, take real or discrete inputs and outputs, and deal well with noisy data.

An ANN consists of layers of *neurons*, or switching units: typically, an input layer, one or more hidden layers, and an output layer. Input values are presented at the input layer and predictions are obtained from the output layer. Figure 4.7 shows an example of a fully connected feed-forward ANN. Every unit in each layer is connected to all units in the next layer by weighted edges. Each unit applies an *activation function* to the weighted sum of its inputs and passes the result to the next layer. Figure 4.8 [40] shows a unit with a sigmoid activation function. One can use any nonlinear, monotonic, and differentiable activation function. We use the sigmoid activation function.

Training the network involves tuning edge weights via backpropagation, using gradient descent to minimize error between predicted and actual results. In this iterative process, the training samples are repeatedly presented at the input layer, and the error is calculated between the prediction and the actual target. The weights are initialized near zero and are updated using an update rule (similar to the one shown in Equation 4.1) in the direction of steepest decrease in error. As weights grow, the network becomes increasingly nonlinear.

$$w_{i,j} \leftarrow w_{i,j} - \eta \frac{\partial E}{\partial w_{i,j}} \tag{4.1}$$

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

4.5 Evaluation

For our experimental evaluation of ANN-based performance prediction, we use the two platforms (quad-core and eight-core) and benchmark suite as described earlier. Performance counters are collected using PAPI version 3.5. We could train our prediction model with various training sets of one or more benchmarks. We choose a single benchmark (*ua*) to train the model, trading potentially higher prediction accuracy for less training time. *ua* has a large number of phases and widely varying execution characteristics on a per phase basis, including IPC, scalability, locality, and granularity. In practice, the model would generally be trained a single time with a given set of training

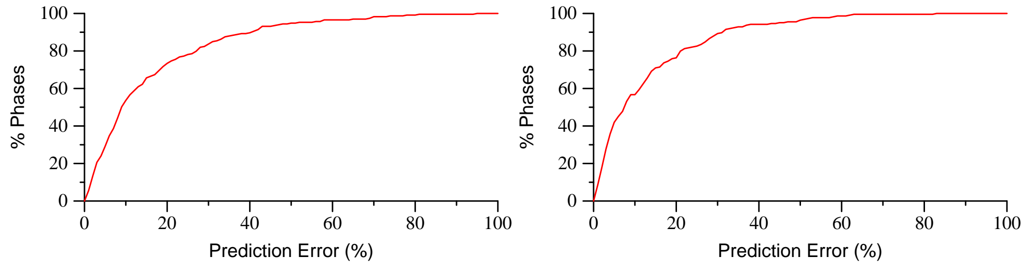


Figure 4.9: Cumulative Distribution Function (CDF) of Prediction Error for the 4-core System (left), and the 8-core System (right)

applications, and would subsequently be used for any desired application, with possible refinements to reflect data from the current workload.

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

For the eight-core system, we use phases from the *ua* benchmark for training and evaluate on phases from all remaining benchmarks. Our experimental platform only has two hardware event counter registers. In this case, we decrease the number of counters sampled to reduce the sample execution period. We record instructions retired and L1 data cache accesses only, since we find them to have the strongest correlation to IPC.

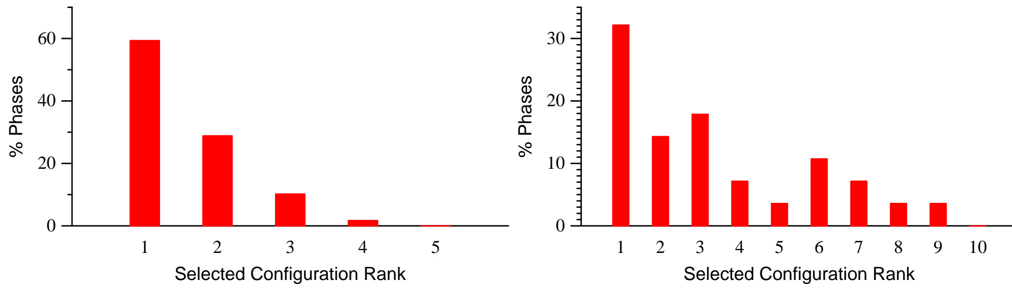


Figure 4.10: Percent of Phases for which each Ranking Configuration is Selected on the 4-core System (left), and the 8-core System (right)

Figure 4.9 gives a cumulative distribution function of the error of our ANN-based predictor, showing the percentage of samples that fall within increasingly higher levels of observed error. Specifically, we make predictions for each of the target thread configurations, and these results are accumulated over all predictions made. For each sample, we calculate error as $|(IPC_{obs} - IPC_{pred})/IPC_{obs}|$, where IPC_{obs} corresponds to the actual measured cumulative IPC and IPC_{pred} corresponds to the cumulative IPC predicted by the model. For the quad-core system, the median error is only 9.1%. Further, 53.6% of the predictions exhibit errors less than 10%. The median error on the eight-core machine is 7.5%. Here 56.7% of predictions exhibit less than 10% error, and 42.0% of predictions exhibit less than 5% error. We achieve These low error rates despite very complex scalability patterns.

An alternative metric for evaluating the accuracy of the predictor in the context of concurrency throttling is the rate at which the best configuration is selected. The left graph of Figure 4.10 shows the percentage of phases where each of the configurations is selected. In 59.3% of phases on the quad-core system, the best configuration is correctly identified, and the second best configuration is selected in an additional 28.8%. In only one case out of 59 is the second worst configuration selected, and the worst is never predicted as optimal.

The right graph of Figure 4.10 presents the percentage of phases for which the approach selects each of the configurations on the eight-core system. In each case, the predictor identifies nearly optimal configurations most of the time. The predictor selects the best configuration for 32.1% of phases and one of the top five for 75.0%. For phases with poor scalability it becomes difficult for the models to differentiate among multiple configurations with near-identical performance. However, we find that misprediction of the optimal configuration does not harm performance significantly, making the overall impact tolerable. These results show that ANN-based performance prediction can effectively identify optimal or near-optimal concurrency levels.

CHAPTER 5

CONCURRENCY THROTTLING

Concurrency throttling, like dynamic voltage and frequency scaling (DVFS), has beneficial processor power management properties. DVFS mainly targets dynamic power. Increases in static (leakage) power with each processor generation diminish DVFS's potential for reducing power without performance penalties. In contrast, concurrency throttling may still achieve substantial power savings on both fronts [18].

Runtime search methods can discover optimal or near-optimal concurrency levels for phases of parallel code separated by synchronization or communication operations. However, search methods may require many executions of a phase to converge to an optimal operating point. In particular, the number of executions depends both on the number and the topology of cores [15]. The topology of cores (and relationship to cache) is important because different mappings of a given number of threads on a given topology may vary dramatically in performance. With tiled embedded processors having 64-512 cores (Tilera's Tile64 and Rapport's Kilocore, already on market), exhaustive or heuristic search of program and system configurations becomes prohibitively expensive.

Runtime performance prediction overcomes limitations of direct search methods at the potential cost of reduced accuracy in identifying the best operating points. These approaches test fewer configurations to reduce online overhead, but their efficacy depends on prediction accuracy. We present scalability prediction models, evaluating them for prediction accuracy and success at identifying optimal configurations per phase.

Concurrency throttling is not feasible in all parallel applications and programming models. In principle, concurrency throttling can be applied transparently to applications where neither the parallel computation nor data distribution depend on the number

and topology of the processors. Shared-memory programming models such as OpenMP and Transactional Memory meet these requirements, whereas distributed-memory programming models such as MPI need application and/or runtime system modifications to benefit from concurrency throttling. Programming models where parallelism is expressed independently of number and type of processors are essential to simplifying the process of parallel programming [5]. Our contribution targets such models.

Chapter 4 demonstrates improved performance when using fewer cores. This is due to limited scalability of several parallel execution phases. We use ANN-based performance prediction to identify the desired level of concurrency and the optimal thread placement. The ANNs are trained offline to model the relationship among PMC event rates observed while sampling short periods of program execution and the resulting performance with various levels of concurrency. The derived ANN models allow us to perform online performance prediction for phases of parallel code, and we do so with low overhead by sampling PMCs. Our ANN approach removes the burden of managing the training phase and providing domain-specific knowledge, two steps that are crucial to regression-based prediction strategies [35].

We now describe the runtime system's performance prediction component that dynamically throttles concurrency to improve performance and energy efficiency. The system adapts applications by identifying better-performing numbers of threads and thread placements for each phase. Again, phases are collections of parallel loops or basic blocks assigned for execution to different threads. We use the same Intel quad-core experimental platform and benchmark suite as described in Chapter 4.

5.1 Methodology

We model the effects of changing concurrency and thread placement. Hardware PMC values collected during a brief sampling period at maximal concurrency become input to our ANN ensemble that predicts IPC for each phase on alternative configurations. The online sampling runs on as many cores as available to represent the greatest possible interference among threads, and resulting predictions estimate the degree to which contention will be reduced by throttling concurrency. We collect PMC values, $e_{i,S}$, for each sample configuration, S, and normalize observed values to the elapsed cycle counts, yielding *event rates*, $r_{i,S}$. Our prediction module produces the following function for each target configuration, T, mapping observed event rates on the sample configuration to the target configuration IPC:

$$I\hat{P}C_T = F_T(IPC_S, r_{1,S}, ..., r_{n,S}) \tag{5.1}$$

We sort predictions and select the configuration with the highest predicted IPC for the corresponding program phase. Once a configuration is selected, our runtime library ensures all subsequent executions of the same phase use the chosen concurrency and thread placement. Figure 5.1 illustrates the runtime system.

We derive the prediction module from ANNs that we train on the hardware counter values and IPCs from the target configurations. The PMC are selected as a collection that represent performance-critical resources, e.g., caches and buses. We choose training applications representing a variety of runtime characteristics, as identified by the PMCs. During the short training period, patterns in effects of event rates on training benchmark IPCs are observed and encoded in the ANN models.

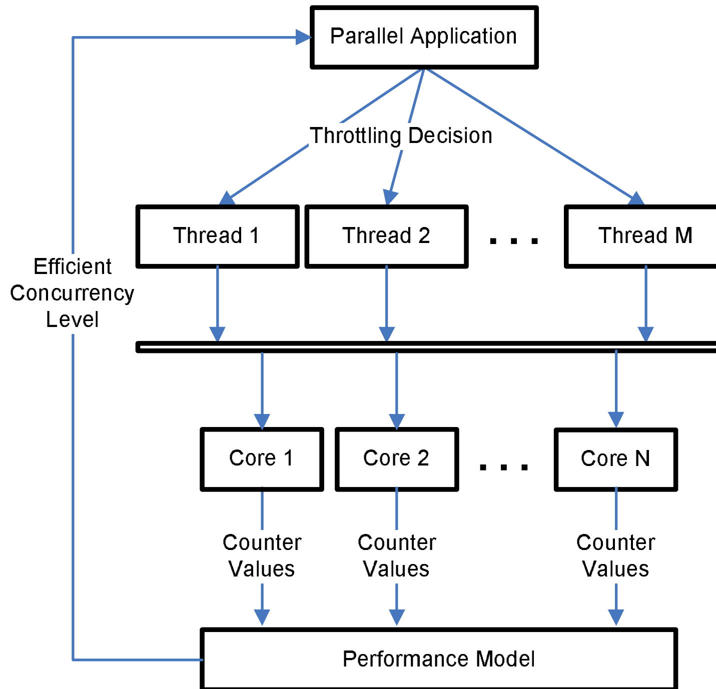


Figure 5.1: Runtime System for Concurrency Throttling

Our system currently supports applications parallelized using OpenMP and instrumented with calls into the runtime library. Parallel regions in OpenMP tend to have consistent execution properties, and they also represent the finest granularity at which the number of threads can be changed at runtime, therefore we use them as program phases. Library calls are added at the beginning and end of each phase to initialize our runtime system, to collect performance counter values, to make performance predictions and to enforce concurrency decisions made for each phase.

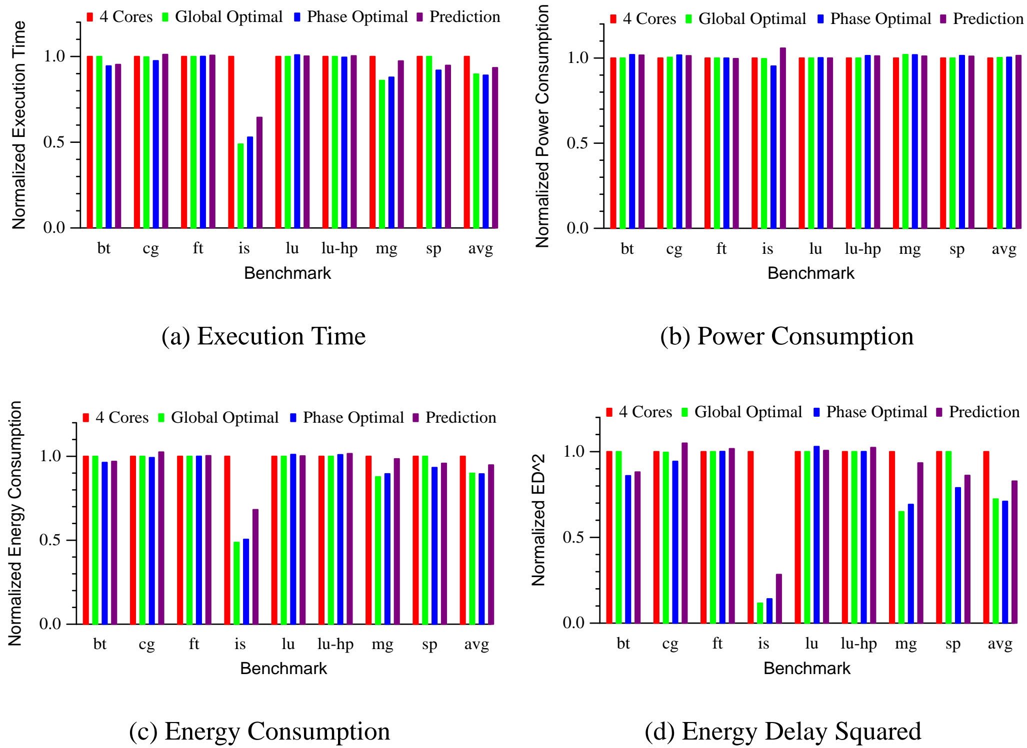
Previous work has experimented with both empirical search-based [17] and statistical prediction-based [16] determination of concurrency levels. Each of these strategies suffers from certain difficulties, and using ANNs in this context addresses these limitations. The configuration identification process for empirical searching [17] requires online testing of a potentially many configurations, which incurs large overheads that can reduce the gains through adaptation. While at most five configurations need to be

tested for empirical searching on our platform, future generation systems with many cores will require significantly more. Therefore, the benefits of prediction-based adaptation relative to searching will only grow in the future.

Regression-based models for performance prediction, on the other hand, have very low overhead. However, they require significant effort and machine-specific training in the derivation of effective models of performance [16] [35]. This labor-intensive training period may render regression-based approaches unsuitable for use in many contexts. Since our approach automatically develops a model based on a collection of samples without requiring user-input and domain-specific knowledge, the minor costs associated with using ANNs, along with the comparable online overhead of PMC collection and model evaluation, may make it more appropriate than regression-based models.

5.2 Evaluation

We first analyze results on the quad-core platform. Figure 5.2 displays the results of our prediction-based concurrency throttling approach normalized to execution on all cores, as well as those of alternative execution strategies. A popular metric in power-aware HPC is energy-delay-squared (ED^2), which considers power consumption but is more influenced by performance, commensurate with the heavy emphasis on performance in HPC. We compare against using all available cores for multithreaded execution, which would be the default for a performance-oriented developer. We present results for two approaches based on oracle-derived configurations. The first, global optimal, uses the best static configuration for an entire application. The second, phase optimal, uses the best configuration per phase. In each case, this information would not normally be available, but they serve as points of comparison to evaluate the library's effectiveness.



(a) Execution Time

(b) Power Consumption

(c) Energy Consumption

(d) Energy Delay Squared

Figure 5.2: Execution Time, Power Consumption, Energy Consumption, and ED^2 of Prediction-Based Adaptation Compared to Alternative Execution Strategies

By using our approach for low overhead identification of improved concurrency levels, we obtain an average performance gain of 6.5% compared to the default strategy of simply using all available cores. Even *bt*, which scaled well on the four core machine, sees a substantial gain of 4.7%. Our phase-aware adaptation strategy successfully identifies phases in *bt* that can be improved by concurrency throttling. Additionally, *sp* runs 5.2% faster when given more cores.

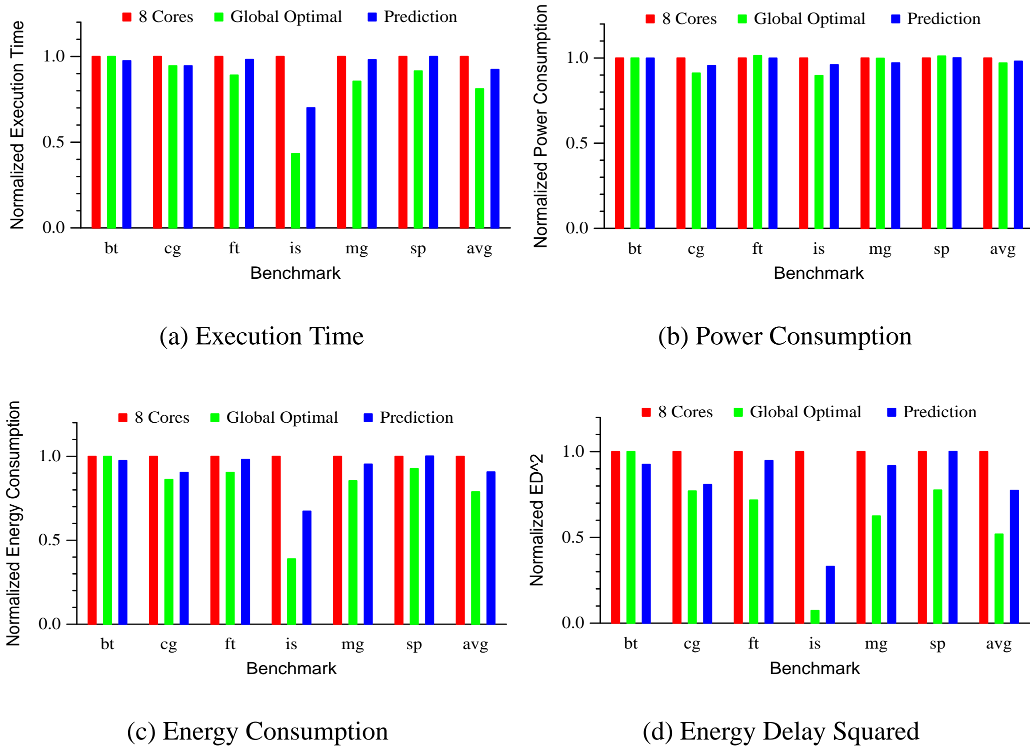
When compared to the two oracle-derived strategies, our runtime system falls short of these oracular approaches, coming in 2.5% and 4.9% slower (geometric mean) than the global and phase optimals, respectively. This shows potential benefits of improving prediction accuracy. Further, reduced online overhead of sampling is possible on ar-

chitectures with more counter registers to reduce the number of rotations necessary for event collection.

One surprising result is that no power is saved through concurrency throttling, on average. We appropriately leave cores idle, but it is likely that changing the binding of threads interferes with cache locality. This increases bus traffic and memory accesses, which increase off-chip power consumption. On-chip power consumption is reduced by small amounts, but this is overwhelmed by the off-chip increase. There are also cases, as pointed out in Chapter 4, where power increases from selecting reduced thread configurations with better performance. Together, these effects average increase power consumed by 1.5%. Given the considerable improvement in execution time, the total energy consumed goes down by an average of 5.2%. We expect more power savings as we add more cores to a CMP, since the cores represent a larger portion of total power consumed, and throttling may have the potential to save more of that power.

Given the large improvements in execution time, with very minor increases in power consumption, we obtain ED^2 savings of 17.2%. The most significant result occurs with *is* (71.6% improvement in ED^2), which shows that for applications that scale poorly, concurrency throttling is imperative to achieve energy efficiency. Further gains are possible, since the phase optimal execution improves performance by 29.0% compared to using four cores.

The eight-core platform shows even more promising results. With more cores, the possible savings from phase-level adaption improves. Figure 5.3 displays the results of prediction-based concurrency throttling normalized to execution with all available cores for each benchmark. Additionally, we present the geometric mean of the results. We compare against using all available cores for multithreaded execution. We also present results for an approach based on oracle-derived configurations (global optimal), where



(a) Execution Time

(b) Power Consumption

(c) Energy Consumption

(d) Energy Delay Squared

Figure 5.3: Execution Time, Power Consumption, Energy Consumption, and ED^2 of Prediction-Based Adaptation Compared to Alternative Execution Strategies

we use the best static configuration for an entire application. We exclude phase optimal because this information is not easily available. It requires an exhaustive search of the configuration space using complete application executions.

Using an ANN-based predictor yields a mean performance improvement of 7.4% over full concurrency. *is* exhibits a 30.3% performance improvement compared to running at eight threads. In all cases, the predictor maintains or improves performance relative to maximum concurrency. *bt* exhibits a modest 2.3% speedup, in spite of scaling fairly well to all cores. This demonstrates the potential advantage of performing adaptation at the phase level. When compared to the oracle-derived strategy (global optimal), our runtime system is 11.2% slower on average. This shows potential benefits of

improving prediction accuracy. There are cases where power increases through selecting reduced thread configurations with better performance. We reduce power consumed by 1.9% overall, which is an improvement over the four core case, where we increase power by 1.5%. This is expected, as mentioned earlier, since the CMP cores represent a larger portion of total power consumed, and throttling saves more of that power. Given the considerable improvement in execution time, the total energy consumption decreases by an average of 9.3%. With small decreases in power consumption, we reduce overall ED^2 by 22.6%. However, further gains are possible using this approach as global optimal execution improves performance by 48.1% compared to using all eight cores.

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

CHAPTER 6

ECHO: A FRAMEWORK FOR EFFICIENT POWER MANAGEMENT

We propose Echo, a framework for efficiently managing power consumption of multiprogrammed and multithreaded workloads. We build upon the power-aware scheduler (from Chapter 3), and include support for multithreaded programs. We utilize concurrency throttling (Chapter 5) to decrease or increase the number of threads. This works much like suspending and resuming processes in a multiprogrammed workload. Including concurrency throttling gives us the advantage of possibly improving performance while reducing the number of threads for a multithreaded program.

We address two types of systems: those that support DVFS, and those that do not. Results for experiments using suspension only are presented in Chapter 3. In this chapter, we performs experiments on machines that support DVFS: the AMD quad-core (Table 2.4), and the Intel eight-core (Table 2.5). Echo utilizes DVFS to maintain a given power envelope. When the DVFS option is exhausted, Echo suspends/resumes single-threaded programs, and performs concurrency throttling for multithreaded programs.

6.1 Multiprogrammed Workloads

We first consider multiprogrammed workloads, running within the Echo framework. The Echo framework uses the power predictor from Chapter 2 to schedule multiprogrammed workloads in real time, so they run within a specified power envelope. Figure 6.1 illustrates Echo's setup and use. We spawn a process on each core of the CMP. The processes are bound to a particular core and do not migrate to other cores during the course of execution. The system makes real-time predictions for per-core and system power based on collected performance counter data. We scale frequency to lower power

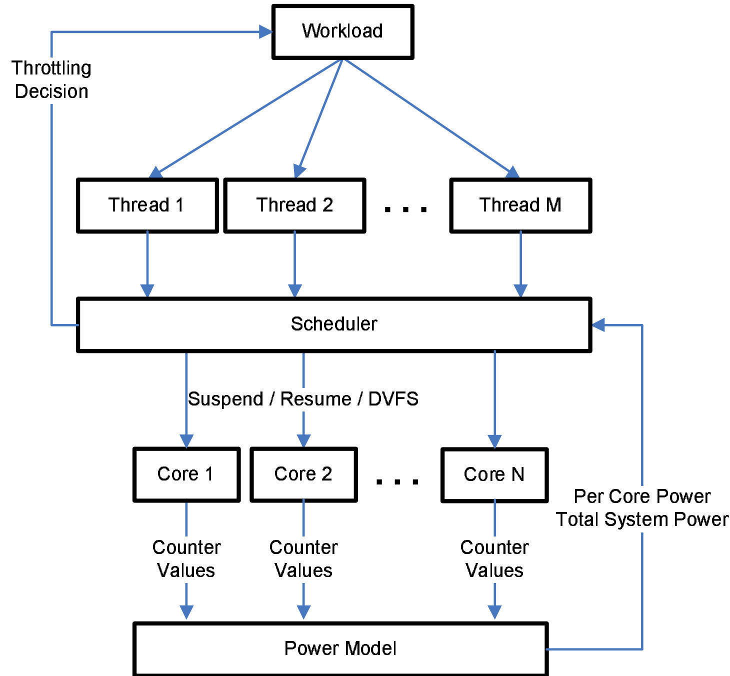


Figure 6.1: The Echo Runtime System

usage, and suspend processes after exhausting all possibilities for frequency scaling. We implement the four policies from Chapter 3, but we describe them here with the option to apply DVFS.

Simple (DVFS)

The *simple* policy implements a blanket envelope on power. It reduces frequency for processes such that system power is just below the power envelope (to try to maintain performance). If all voltage/frequency scaling options are exhausted, it starts suspending processes. For example, assume that current system power is 190W and the power envelope is 180W. Assume that the policy must choose between two processes consuming 20W and 25W, respectively. Decreasing frequency for the process core brings down power for the two processes to 10W and 12W, respectively. The manager reduces

frequency for the first process to bring system power down to 170W, rather than choosing the second process to be further below the envelope (at 167W). When increasing frequency for a process, it again considers the process that pushes power closest to the given envelope. The policy works similarly without DVFS. When all DVFS possibilities are exhausted, it starts considering processes to suspend.

Maximum Instructions/Watt (DVFS)

The *max inst/watt* policy attempts to achieve the most energy-efficiency under the given power envelope. When the envelope is breached, it reduces frequency for the process with the lowest number of instructions committed per watt of power consumed. The instructions to power ratio is recorded for later reference. When considering a process to increase core frequency from the pool of candidates, it increases frequency for the process with the highest number of instructions committed per watt such that the system remains within the power envelope. This policy generally gives the best overall performance compared to others. When all cores are running at the lower frequency, it starts considering processes to suspend. When power is available, it restores processes first (for fairness) before increasing frequency.

Per-Core Fair (DVFS)

The *per-core fair* policy gives each core a fair share of the consumed energy. It maintains a running average of the power consumed by each core at a given time. If the system exceeds the power envelope, the policy decreases frequency for the core with the highest average consumed power (or energy). The running average is updated at every

interval. We increase frequency for the core with the process having the lowest average, as long as the system remains under the power envelope. When the DVFS option is exhausted, the policy starts considering processes to suspend. When power is available, the policy resumes processes starting with the one with the lowest average power, and then it considers increasing frequency. Such a policy can help regulate core temperature, since it throttles cores with high power usage. Static power is a function of voltage, process technology, and temperature. Increased temperature leads to increased leakage power, and adds to total power. The temperature variation among cores is smaller for this policy compared to others.

User-based Priorities (DVFS)

The *user-based priorities* policy takes input from the user and considers process priority when scaling frequency to remain under the envelope. For example, assume that current system power is 190W and the power envelope is 180W. Assume that the policy must choose between two processes consuming 20W and 25W, respectively. Decreasing frequency for the each core brings power for the two processes down to 10W and 12W, respectively. The first process has higher priority than the second. The manager suspends the second process, even though suspending the first would keep the system closer to the power envelope. When resuming a process, it again considers the process priority, first resuming all suspended processes in order of priority, and then scaling frequency such that power remains under the envelope. This policy is desirable when the user has outside knowledge, e.g, runtime, that can be exploited to deliver better performance. For example, it would be efficient to give high priority to a short-running, power-hungry process. Once that process finishes execution, the management complexity is reduced, and the remaining processes could continue to run within the power envelope.

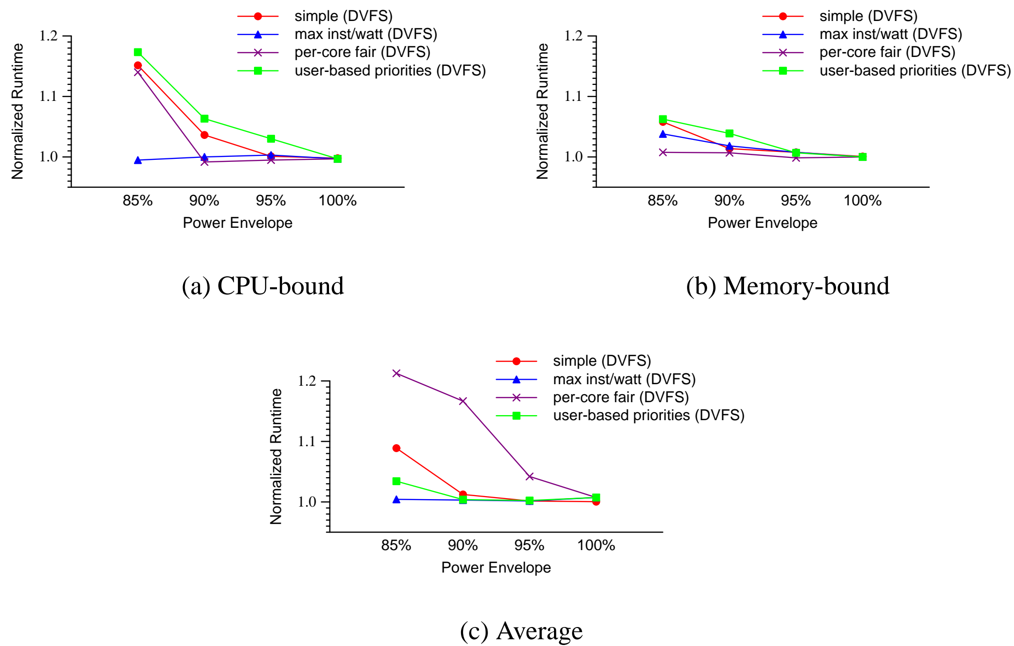
Evaluation

We use the power predicted for processes to enforce a power budget for a multiprogrammed workload on the CMP. We perform two sets of experiments. For the first set, we consider the AMD quad-core system. It supports DVFS, running at frequencies of 1.1 GHz and 2.2 GHz. Chapter 2 presents the power model for the system running at 2.2 GHz (with 3.8% median error). We form another power model (with 4.8% median error) for the system running at 1.1 GHz. Echo performs per-core frequency scaling to remain below the system power envelope. For the second set of experiments, we maintain the power envelope for the eight-core system that supports frequency scaling (2.0 GHz, 2.67 GHz). The power model from Chapter 2 for this system running at 2.67 GHz delivers a median error of 2.8%. For the system running at 2.0 GHz, we form another power model and it delivers a median error of 2.9%. The Echo framework can predict power for the two frequencies to aid in policy decisions. The eight-core platform does not support DVFS per core, but rather DVFS per CMP in the SMP. As a result, we perform DVFS on a chip-level (per four cores), instead of per core. When all core frequencies have been scaled down and the power envelope is still breached, the policies suspend processes to reduce power. For these experiments, we assume the system power envelope to be reduced by 5%, 10%, or 15%. The runtimes are compared against runtimes when not enforcing a power envelope, and the envelope is then reduced from 5-15% of the workload's peak power usage. We consider three sets of multiprogrammed workloads with varying degrees of *computation intensity* (Table 6.1). We define *computation intensity* as the ratio of instructions retired to cache misses. The first set contains a multiprogrammed workload with the highest *computation intensity* (CPU-bound), the second takes the benchmarks that exhibit average *computation intensity* (Average), and the third contains the benchmarks with the lowest *computation intensity* (Memory-bound).

Table 6.1: Multiprogrammed Workloads for Evaluation

Benchmark Set	4-Core	8-Core
CPU-bound	ep, gamess, namd, povray	calculix, ep, gamess, gromacs, h264ref, namd, perlbench, povray
Average	art, lu, wupwise, xalancbmk	bwaves, cactusADM, fma3d, gcc, leslie3d, sp, ua, xalancbmk
Memory-bound	astar, mcf, milc, soplex	applu, astar, lbm, mcf, milc, omnetpp, soplex, swim

Each of the policies is effective in completing the workload under the given power envelope, but with varying degrees of performance loss. Figure 6.2 presents normalized runtimes for the complete set of policies and power envelopes on the AMD quad-core CMP. Here the policies can utilize per-core DVFS. For the CPU bound workload in Figure 6.2(a), the *max inst/watt* policy delivers the best performance, and the *user-based priorities* policy delivers the worst. For the memory-bound workload, the *per-core fair* policy delivers best performance. Performance improves marginally (by 0.15%) for the power envelope of 95%. Without any power envelope, all processes compete with each other for cache and memory bandwidth. When the power envelopes come into play, the policy throttles processes to conserve power, and in the process also frees cache and memory bandwidth, which helps speed execution of the running processes. The effects are not as profound as when using suspension for the same workload on the same machine (Chapter 3), because the processes are still running, but at lower frequencies. Contention is higher than if one of the processes were suspended. The average workload in Figure 6.2(c) exhibits the least variation among policies, and *max inst/watt* achieves the best performance. We do not observe the workload-speedup effect (when throttling processes frees cache and memory bandwidth). The *user-based priorities* policy performs almost as well as the best case, while *per-core fair* deviates the most.



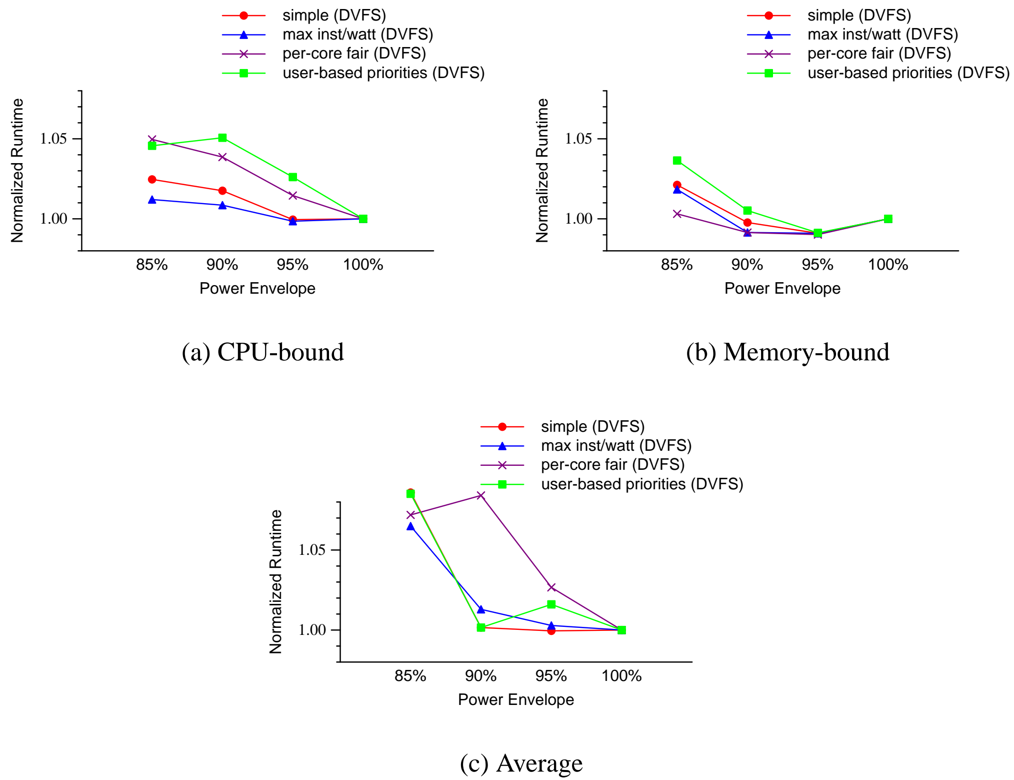
(a) CPU-bound

(b) Memory-bound

(c) Average

Figure 6.2: Runtimes for Workloads When Using DVFS in Combination with All Policies on AMD Phenom (Normalized to No Power Envelope)

The experiments on the eight-core system are more interesting, since we have eight processes and, therefore, more potential for saving power with minimal performance loss. The Echo framework chooses between frequencies of 2.0 GHz and 2.67 GHz to vary power usage. If DVFS is insufficient, process suspension is invoked. Again, each of the policies is effective in completing the workload under the given power envelope, with varying degrees of performance loss. The loss in performance is lower compared to the AMD quad-core experiments above, since more cores offer more opportunities for reducing power while maximizing performance. Compared to the experiments in Chapter 3, we deliver much better performance. Scaling frequency still allows progress, but suspension does not. Figure 6.3 shows normalized runtimes for the complete set of policies and power envelopes on the eight-core system. For the CPU-bound workload in Figure 6.3(a), the *max inst/watt* policy performs consistently well, but the *per-core fair* and *user-based priorities* policies slow down the workload by up to 5%. For the



(a) CPU-bound

(b) Memory-bound

(c) Average

Figure 6.3: Runtimes for Workloads When Using DVFS in Combination with All Policies on Intel 8-Core (Normalized to No Power Envelope)

memory-bound workload, performance improves marginally (0.9%) for the 90% and 95% power envelopes, respectively. This behavior is similar to that for the quad-core memory-bound workload. The 85% envelope shows minimal loss for the *per-core fair* policy. The average workload in Figure 6.3(c) exhibits the most variation with policy, and *max inst/watt* achieves the best performance. We see behavior similar to the memory-bound workload for the 95% power envelope. Performance loss varies as the envelope is reduced, and shows that the choice of policy depends not only on the workload but on the power envelope. The Echo runtime system provides the most efficiency within the given power envelope, utilizing DVFS first, and then suspension.

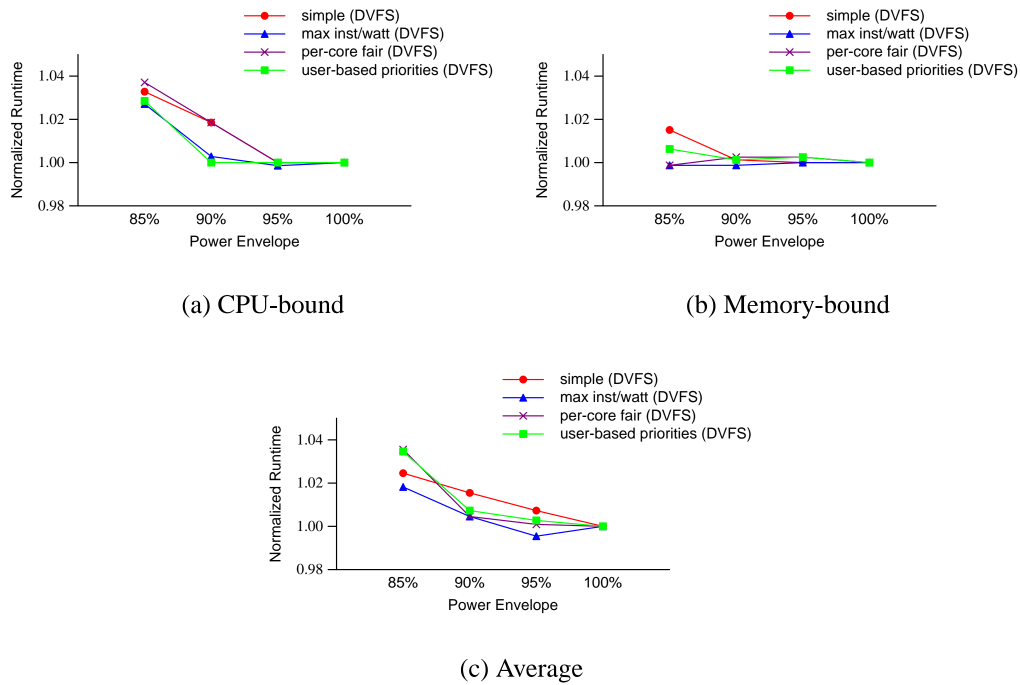
Table 6.2: Mixture Workloads for Scheduler Evaluation

CPU Intensity	Single-threaded component	Multithreaded component
CPU-bound	povray, gamess, namd, perlbench	bt
Average	bzip2, zeusmp, dealII, gcc	sp
Memory-bound	omnetpp, soplex, astar, mcf	cg

6.2 Single-threaded and Multithreaded Mixture Workloads

We now consider mixtures of single-threaded and multithreaded workloads, to run under the Echo framework. Each workload consists of a multithreaded benchmark and four single-threaded benchmarks. The workloads we use are outlined in Table 6.2. We perform experiments on the Intel eight-core system from Table 2.5. The multithreaded benchmark runs on one quad-core while the four single-threaded benchmarks run on the other quad-core. We run the workloads within a specified power envelope. The single-threaded processes are bound to a particular core and do not migrate to other cores during the course of execution.

The Echo framework makes real-time predictions for per-core and system power based on collected performance counter data, and utilizes DVFS to maintain a given power envelope. When the DVFS option is exhausted, Echo suspends/resumes single-threaded programs, and performs concurrency throttling for multithreaded programs. We implement policies similar to those in Section 6.1, except for *max inst/watt*. For the *max inst/watt* policy, we combine adaptive concurrency throttling with DVFS to achieve maximum throughput for workloads that include multithreaded benchmarks. When the power envelope is breached, we predict the most efficient concurrency level for the multithreaded benchmark in the workload. At the same time, we scale frequency such that power remains within the envelope. We reduce frequency for the process with the lowest number of instructions committed per watt of power consumed, and record instructions to power ratio. When considering a process to increase core frequency, the



(a) CPU-bound

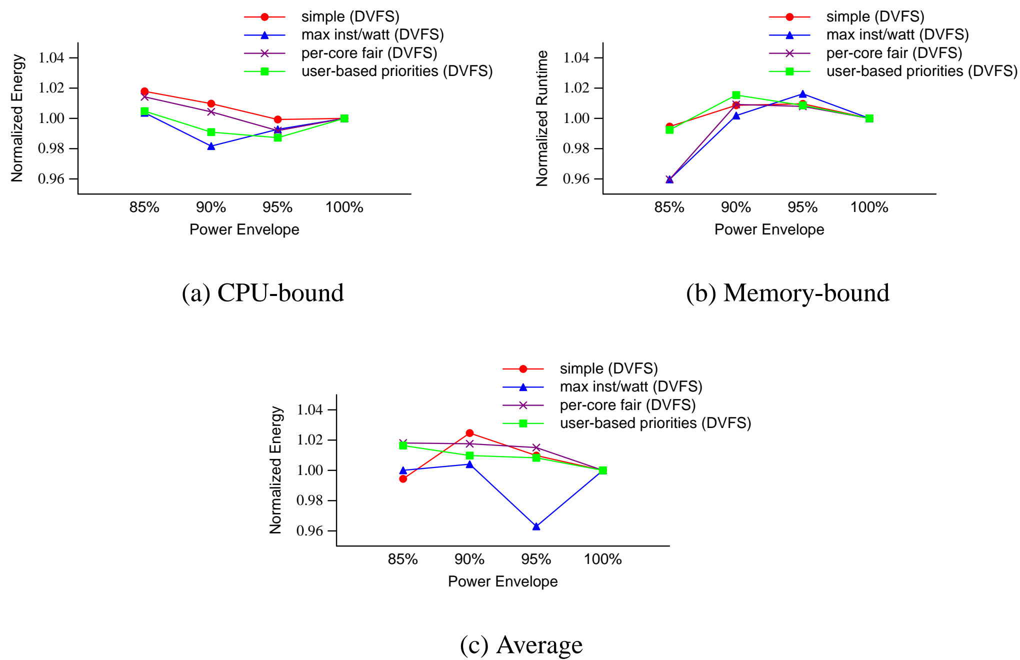
(b) Memory-bound

(c) Average

Figure 6.4: Runtimes for Mixture Workloads on the Intel 8-Core System (Normalized to No Power Envelope)

policy it increases frequency for the process with the highest number of instructions committed per watt that keeps the system within the power envelope. The advantage of performing concurrency throttling is primarily seen as runtime and energy savings. It might not necessarily lower power, which is why we also perform DVFS to enforce the power envelope.

We maintain the power envelope for the eight-core system that supports frequency scaling (2.0 GHz, 2.67 GHz). The policies consider DVFS to lower power usage. To aid in policy decisions, Echo predicts power for the two frequencies. When all core frequencies have been scaled down and the power envelope is still breached, the policies suspend processes/threads to reduce power. For all three workloads, we observe minimal performance loss (2-4%) for all policies. The *max inst/watt* policy (that includes dynamic concurrency throttling) generally surpasses other policies for all workloads.



(a) CPU-bound

(b) Memory-bound

(c) Average

Figure 6.5: Energy for Mixture Workloads on the Intel 8-Core System (Normalized to No Power Envelope)

This is expected, since the policy has the advantage of reducing power as well as improving performance for the multithreaded workload. For the CPU-bound workload, the *user-based priorities* and *max inst/watt* policies exhibit similar runtimes (Figure 6.4(a)). However, since the *max inst/watt* is also optimizing energy consumption, in addition to performance, we deliver energy savings compared to *user-based priorities* in Figure 6.5(a). For the memory bound workload in Figure 6.4(b), we achieve no significant performance loss for *max inst/watt* policy. The energy graph for this workload (Figure 6.4(b)) shows variations in energy usage. Energy decreases with decreases in the power envelope. Echo manages the multithreaded benchmarks to increase energy efficiency, and also overlaps periods of latency (due to memory misses) to better utilize the power budget. For all workloads, the *per-core fair* policy tries to spread the power budget equally among all cores. This increases overall energy usage compared to the best case for the CPU-bound and average workloads. However, for the memory bound

workload, the throttling in power lowers resource contention, and lowers execution time as well as energy. The *simple* policy enforces the power envelope well, and preserves performance by always trying to use as much power as possible. However, this does not translate into more energy efficiency. It lags behind the best case for all workloads.

6.3 Scalability: More Cores and Fine-Grained DVFS

As we head into the many-core era, we expect the Echo framework to continue to scale. The greater number of processes running on these cores shows much variability. This affords Echo finer control over the power envelope, and depending on the policy, the ability to do so with minimal performance loss. In Chapter 3, when using suspension for control of the power envelope, moving from the quad-core systems to the eight-core system reduces worst case runtime from 2x to 1.5x. We observe the same trend in Figures 6.2 and 6.3, with worst case runtime reducing from 1.21x to 1.08x. We expect this trend to continue as we add more cores. The adaptive concurrency throttling approach (Chapter 5) also allows for opportunities for saving energy and power. The average energy saved improves from 5.2% to 9.3% as we go from the Intel quad-core to the Intel eight-core platform. Average power saved increases from -1.5% to 1.9%.

In Chapter 2, we demonstrate the scalability of the power predictor by predicting power for quad-core and eight-core platforms. We use DVFS in the Echo framework to reduce the power envelope. We form two separate power models, one for a frequency of 2.0 GHz, and another at 2.67 GHz. When finer control DVFS is available, it becomes time consuming to form a model for each of the frequencies supported by the processor. We investigate forming a single generalized power model that needs only a subset of the frequencies to be sampled. We form a piece-wise model based on microbenchmark

Table 6.3: Median Errors for Per-Frequency and Generalized Power Models

Frequency	Per-Frequency Error	Generalized Error
2.00 GHz	2.9%	5.3%
2.33 GHz	3.3%	3.8%
2.67 GHz	2.8%	2.9%

runs at the the lowest and highest frequencies supported. We normalize the PMC value, e_i, to the elapsed cycle count and get an *event rate*, r_i, for each counter. The prediction model uses these rates, temperature, T, and frequency, F, as input. Since we are trying to predict power, an instantaneous metric, it is important to normalize the counters to the elapsed cycle count. This allows data from different frequencies to be used together to form the generalized model. We only use data from running at 2.0 GHz and 2.67 GHz. We interpolate for the frequencies that fall within these two. We model core power using our piece-wise model based on multiple linear regression. We produce the following function (Equation 6.1), mapping frequency, F, rise in core temperature, T, and observed event rates r_i to core power P_{core}:

$$\hat{P}_{core} = \begin{cases} 3.325 + 0.035 * log(r_1) + 0.617 * r_2 + -0.859 * r_3 + 3.101 * r_4 + 0.664 * T \\ \quad + 0.524 * F, \quad r_3 < 10^{-6} \\ 1.880 + 0.014 * log(r_1) + 0.399 * r_2 + 0.023 * log(r_3) + 1.437 * r_4 + 0.385 * T \\ \quad + 3.293 * F, \quad r_3 \geq 10^{-6} \end{cases} \tag{6.1}$$

$$where \quad r_i = e_i/(cycle\ count)$$

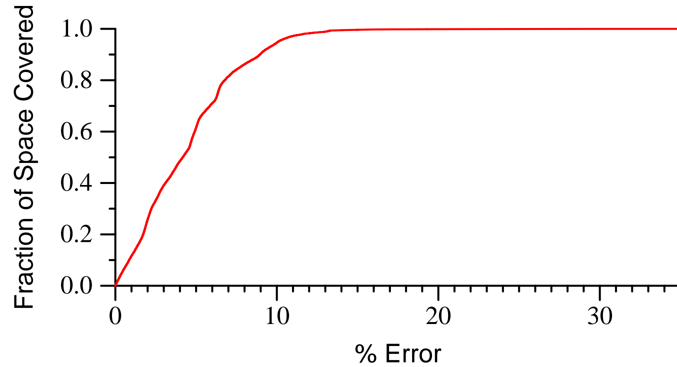
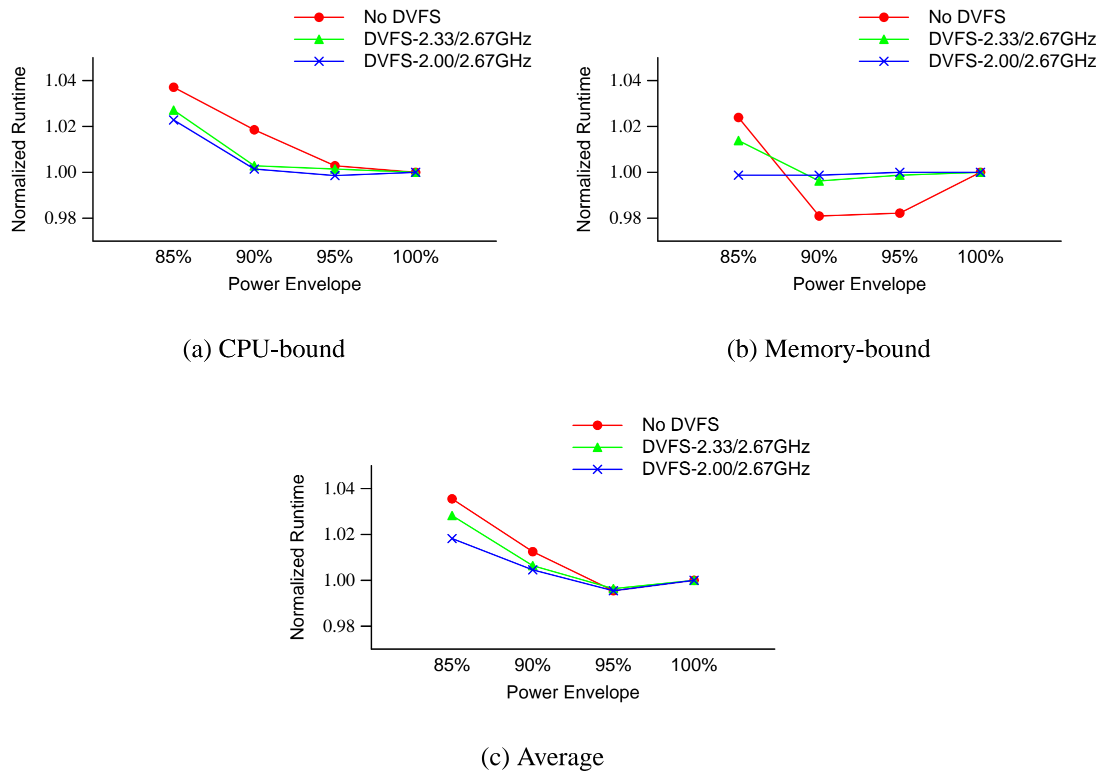


Figure 6.6: Cumulative Distribution Function (CDF) of Prediction Error for the Generalized Model

We test on the SPEC-OMP, SPEC 2006, and NAS benchmarks, and obtain median errors shown in Table 6.3. The *Per-Frequency* column gives median errors when we form a separate model for each frequency. The generalized model, based on data from 2.0 GHz and 2.67 GHz microbenchmark runs, gives median errors shown in the *Generalized* column. We lose some accuracy from increasing model generality. Error rates are higher, but still acceptable for applications needing power estimates. The CDF in Figure 6.6 gives us an idea of the coverage of our model. We deliver an overall median error of 4.2% (mean error of 4.5%). 61.5% of predictions show less than 5% error, and 94.6% of predictions show less than 10% error. The generalized power model approach would be more useful for systems with multiple levels of DVFS. It allows for faster model formation, and has the advantage of requiring only a single set of equations for all core frequencies.

To illustrate applicability of of the Echo framework in the absence of DVFS, as well as across multiple levels of DVFS, we repeat the experiments from Section 6.2 for two more scenarios. The first case uses only suspension and concurrency throttling to control power. The second case adds the DVFS option between two frequencies, 2.33 GHz and 2.67 GHz. The last case presents results from Section 6.2, where Echo can use



(a) CPU-bound

(b) Memory-bound

(c) Average

Figure 6.7: Runtimes for Mixture Workloads on the Intel 8-Core System with variation in DVFS (Normalized to No Power Envelope)

DVFS between two frequencies, 2.0 GHz and 2.67 GHz. Figure 6.7 shows the normalized runtimes for all three scenarios. For each case, we present the best performance achieved within the Echo framework. Therefore, each point on the curve is the minimum of the runtimes for all four policies. Absence of DVFS hurts performance, but not substantially. The different policies aid in bridging the performance gap due to absence of DVFS. Again, the best policy depends on the workload as well as the power envelope. For the memory-bound workload, absence of DVFS actually improves performance by about 2% for the 90% and 95% power envelopes. This effect is observed in our previous experiments, as well. Suspending processes in a memory-bound workload reduces resource contention, and the benchmarks finish faster as a result. This advantage diminishes as the power envelope is lowered.

Generally, we expect *DVFS-2.00/2.67GHz* to outperform the rest because it suspends processes for the least amount of time. As the power envelope is lowered, *DVFS-2.00/2.67GHz* can still maintain workload progress, while the other two have to suspend processes more often to remain within the envelope. The *DVFS-2.33/2.67GHz* set of experiments follows the performance of *DVFS-2.00/2.67GHz* quite closely for the 95% power envelope. For the lower envelopes, the processes must be suspended more often, and as a result the two performance curves diverge. The average workload (Figure 6.7(c)) presents a good general comparison. *DVFS-2.00/2.67GHz* performs the best, with *DVFS-2.33/2.67GHz* not too far behind. *No DVFS* lags behind the other two, but considering that it has no frequency scaling capabilities at all, it shows promise for use of the Echo framework even in the absence of DVFS. This shows the scalability of our approach to different machine configurations, ranging from no DVFS capabilities to multiple levels of DVFS.

CHAPTER 7

RELATED WORK

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

7.1 Power Prediction

Prior work falls under two different approaches. The first strategy estimates power consumption based on monitoring functional unit usage [32, 53]. The second strategy derives mathematical correlations between performance counters and power consumed, independent of the underlying functional units [14]. In our work, we combine the two approaches, using correlation and architectural knowledge to choose appropriate performance counters, and using analytic techniques to identify the relationship between performance counters and power consumption. We briefly outline related work.

Joseph and Martonosi [32] use performance counters to estimate power in a simulator and on real hardware. They perform experiments in SimpleScalar [7] and on a Pentium Pro. Their hardware supports two PMCs, but they require twelve. They perform multiple benchmark runs to collect the necessary data, and form their model offline. We use only four PMCs and *pfmon* to time-splice between two pairs of events. They could multiplex counters, but extending this to twelve PMCs would require that

program behavior be constant across the entire sampling interval. They are unable to estimate power for 24% of the chip, and thus assume peak power for those structures.

Contreras and Martonosi [14] estimate power for different frequencies using PMCs on an XScale system. Like Joseph and Martonosi, they gather data from multiple benchmark runs, since they require more PMCs than can be used simultaneously. They derive power weights for pairs of frequencies and voltages, and derive a parameterized linear model. They address an in-order single core. while we estimate power for out-of-order CMPs. Our platforms present increased complexity since they are high performance out-of-order multicore processors. This methodology, like that of Joseph and Martonosi, does not seem feasible at runtime, and cannot be used by an OS scheduler or application developer executing multiple programs in parallel.

Wu et al. [53] use microbenchmarks to estimate power consumption of functional units for the Pentium 4 architecture. Dynamic activity for all portions of the chip are not available, and if they were, the approach would exceed the limit of four CMPs for runtime power prediction.

Economou et al. [21] use performance counters to predict power of a blade server. They use custom micro-benchmarks to profile the system and estimate power for different hardware components (CPU, memory, and hard drive), achieving estimates with 10% average error. Like our work, their kernel benchmarks for deriving their model are independent of benchmarks used for testing.

Lee and Brooks [35] use statistical inference models to predict power consumption based on hardware design parameters. They build correlations based on hardware parameters, and use the most significant parameters for training a model to estimate power consumption. They randomly sample a large design space and estimate power

consumption based on previous power values for the same design space profiled a priori. Unfortunately, this methodology requires sampling the same applications for which they are trying to estimate power consumption. This makes the approach dependent on having already trained on the applications of interest. The method is not feasible when executing programs outside of the sampled design space. At runtime, scheduling is not known a priori, and the behavior of programs changes depending on their interactions with other processes sharing the same cache resources.

We estimate power consumption for an aggressively power efficient high performance platform. Our platform differs with previous literature in that: resources are shared across cores, performance counters do not individually exhibit high correlation with power, and significant power variation occurs across benchmarks (depending on workload). Additionally, since we desire real-time power estimation, we are limited to using only the number of performance counters available at runtime in a single run. This prevents us from making fair comparisons of this work with previous work. Our framework estimates power consumption on new programs that we have not run before, and evaluates results on a multi-core system, providing a practical method of estimating power consumption per-core.

7.2 Performance Prediction

Ipek et al. [28] use ANNs for performance prediction in the context of architectural space exploration. In this work, the authors reduce the number of points that must be simulated in evaluating design alternatives via thorough sensitivity studies. The values of various microarchitectural parameters are used to predict the resulting performance of a given application by sampling (simulating) a subset of points in the design space. Lee

and Brooks [35] perform a study with similar objectives using piecewise polynomial regression. Our contribution is an online performance prediction scheme based on a model derived from event rates observed during live execution. Once trained, our model can be applied to any application.

Lee et al. [36] compare the effectiveness of piecewise polynomial regression and ANNs for predicting performance in the context of varying input parameters. Their findings suggest that prediction accuracies between the two approaches are comparable, but each approach is advantageous in different contexts. However, they report that the training process is significantly simplified through the use of ANNs. While there is overlap between Lee et al. and our contribution, we evaluate performance predictions in a very different context. Specifically, we consider predictors for online use in the prediction of parallel application scalability. Further, the linear regression model we evaluate reduces the end-user burden during model specification while still delivering high accuracy.

Marin and Mellor-Crummey [38] semi-automatically measure and model program characteristics, using properties of the architecture, properties of the binary, and application inputs to predict application behavior. Their toolkit predefines a library of functions, and the user may add customized functions, if needed. They vary the input size in only one dimension (in contrast to our studies), and they cannot account for some important architectural parameters (e.g., cache associativity in their memory reuse modeling).

Carrington et al. [11] demonstrate a framework for predicting performance of scientific applications on LINPACK and an ocean modeling application. Their automated approach relies on a convolution method representing a computational mapping of an application signature onto a machine profile. Simple benchmark probes create machine profiles, and a separate tool generates application signatures. Extending the convolution

method allows them to model full-scale HPC applications [12]. They require generating several traces, but deliver predictions with error rates between 4.6% and 8.4%, depending on the sampling rates of the underlying traces. Using full traces obviously performs best, but such trace generation can slow application execution by almost three orders of magnitude. Some applications demonstrate better predictability than others, and for these trace reduction techniques work well: prediction errors range from 0.1% to 8.7% on different platforms. This work complements ours, and the two approaches may work well in combination. Their analytic models could provide bootstrap data, and our models could give them full application input parameter generality.

7.3 Power-Aware Scheduling

Merkel and Bellosa [39] use performance counters to estimate power per processor in an 8-way symmetric multiprocessing (SMP) system, and shuffle processes to reduce overheating of individual processors. Bautista et al. [9] schedule processes on a multicore system for real-time applications. They make assumptions about static power consumption for all applications. They leverage chip-wide DVFS to reduce energy when there is slack in the application.

Isci et al. [29] analyze power management policies for a given power budget, with experiments on the Turandot [41] simulator. They assume the presence of on-core current sensors to obtain per-core power, while we propose a technique to model per-core power without the need for additional hardware. They leverage per-core and domain-wide DVFS to retain performance while remaining within a power budget. While their work involves core adaptation policies, we explore both suspending threads as well as DVFS, to make our approach applicable to a wider variety of systems. Some of the

policies we propose are similar in nature to their work, and some of their policies might work well in our implementation. We do not present any policy comparisons, and leave them for future work.

Rangan et al. [42] also target power management for multicores. They propose migrating threads to cores with different frequencies as an alternative to performing DVFS. They attempt to mitigate the time cost of DVFS, while realizing its full benefits. They perform experiments using the CMP$im [30] cache simulator, augmented with timing and power models, for a 16-core CMP with in-order cores running at two different frequencies. The threads are moved to cores that are better suited to their needs. They find moving threads to be faster than DVFS, since their CMP has relatively simple cores with small amounts of architected state. They target SPEC 2006 single-threaded benchmarks and form workloads based on variability. Incorporating thread migration would benefit the Echo framework as well, and would make an interesting study on a real system like the Sun Rock [49].

7.4 Concurrency Throttling

Yang et al. [54] develop cross-platform performance translation based on relative performance among the target platforms, and they do so without program modeling, code analysis, or architectural simulation. Like ours, their method targets performance prediction for resource usage estimation. They observe relative performance through partial execution of two ASCI Purple applications [34]; the approach works well for iterative parallel codes that behave predictably (achieving prediction errors of 2% or lower) and enjoys low overhead costs. Prediction error varies much more widely (from 5% to 37%) for applications with variable overhead per timestep. Likewise, reusing partial execution

results for different problem sizes and degrees of parallelization renders their model less accurate. Li and Martínez [37] develop a heuristic search approach to improve concurrency using DVFS to optimize power consumption given a fixed performance requirement. The effectiveness of any search-based strategy is likely to decrease as the number of cores from which to choose grows. Search-based strategies require O(cores) samples of an application's execution phases, while prediction-based approaches require O(1) samples. Li and Martínez artificially lengthen their benchmarks to provide enough iterations to reach a decision (up to fifty), whereas our prediction-based approach succeeds on applications with as few as ten iterations. They require hardware modifications to gather input on runtime power consumption.

Sasaki et al. [44] implement a method for performance prediction at various degrees of DVFS. Their model is based on multiple linear regression on performance counters collected at runtime: they identify the level with the lowest power consumption that meets a specified performance requirement and use that for each phase. Prediction-based DVFS adaptation could be applied synergistically with prediction-based concurrency throttling to maintain low overhead while identifying an optimal execution point.

Curtis-Maury et al. [15] propose and evaluate a runtime scalability prediction model for adapting concurrency. They develop a multiple linear regression-based approach for mapping observed hardware event counters to performance estimates at varying levels of concurrency and different thread placements. Singh et al. [46] present a comparison of the two prediction strategies, linear regression and ANNs, and show that both methodologies achieve high accuracy and identify energy-efficient concurrency levels in multithreaded scientific applications. While the regression approach is successful, it requires fine-tuning a regression model with detailed architectural knowledge, whereas ANNs provide a non-linear model without user-provided domain knowledge. ANNs

maintain or improve performance for all benchmarks evaluated, relative to maximum concurrency, while regression slows down two benchmarks.

Compilers can be used to effect concurrency throttling on optimized, parallel codes [33]. Alternatively, runtime phase analysis via direct search algorithms or performance prediction across system and program configurations with varying degrees of concurrency may prove more effective. Compiler methods are effective for codes with simple memory access and thread execution patterns. However, they are constrained by limitations of compiler analysis and compiler-based performance prediction.

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

We derive an analytic, piece-wise linear power model that maps performance counters and temperature to power consumption accurately, independently of program behavior for the SPEC 2006, SPEC-OMP and NAS benchmark suites. We use custom microbenchmarks to generate data for creating our estimation model. The microbenchmarks stress the four PMC categories and we select counters based on their correlation with measured power consumption values. We achieve median error of 3.8% on an AMD quad-core CMP, 2.0% on an Intel quad-core CMP, and 2.8% on an Intel eight-core CMP. By employing three different CMPs with varying number of cores, we validate the portability and the scalability of our power modeling approach.

We leverage our power model to perform runtime, power-aware process scheduling. We suspend and resume processes based on power consumption, ensuring that a given power envelope is not exceeded. We propose and evaluate four scheduling policies and observe the resulting behavior. Our work is likely useful for consolidated data centers, where virtualization leads to multiple servers on a single CMP. Using our estimation methodology, we can accurately estimate power consumption at the core granularity, allowing for accurate billing of power usage and cooling costs. Estimating per-core power consumption is challenging, since some resources are shared across cores (such as caches, the DRAM memory controller, and off-chip memory). Additionally, our current machines only allow us to monitor a few performance counters per-core, depending on the platform (Intel and AMD).

We evaluate the scalability and energy efficiency of multithreaded scientific applications on an Intel quad-core and eight-core CMP. The number of cores per chip is continuing to increase, and so such a study is vital to understanding the future of both

power-aware and high-performance computing. We find that for a large portion of our evaluation suite, scalability is quite poor, and the resulting energy efficiency at high degrees of concurrency suffers as a result.

We improve the energy efficiency for many of our applications by predicting more efficient numbers and placements of threads at runtime, and improve the average ED^2 by 17.2% and 22.6% for the Intel quad-core and eight-core CMPs, respectively. We even manage a power savings of 1.9% for the eight-core CMP. The success of our approach is largely due to a performance prediction model based on applying ANNs to a set of performance counters collected online, which we show achieves high accuracy in terms of IPC prediction as well as identification of a more efficient thread configuration. A major advantage of our approach over existing work is that, through ANNs, we significantly reduce the end-user cost without sacrificing accuracy.

We implement a power efficiency management framework (Echo). We extend the power-aware scheduler to manage single-threaded and multithreaded workloads, and incorporate DVFS and adaptive concurrency throttling. We modify the scheduler policies to take advantage of all power saving approaches (DVFS, suspension, throttling). We discuss scalability to many-core and conclude that the Echo framework will continue to scale as more cores offer more opportunities for finer control over the power envelope. For systems supporting multiple levels of DVFS, we propose a generalized model that predicts power for all available frequencies. This infrastructure could manage energy efficiency, and/or schedule for power efficiency within a given power/thermal envelope.

For future work, we will investigate the possibility of better management policies to further reduce performance loss. Techniques to consider include thread migration, heuristics for interleaving suspension and DVFS, leveraging application phase behavior, and applying smaller sampling intervals. Experimenting with these will be interesting

with respect to effects on throughput. Our collaborators are currently utilizing the power modeling approach to efficiently co-schedule multithreaded applications. They schedule applications at thread counts that are the most energy efficient, using information gathered from the power model. We are attempting to adapt our power prediction approach to include inside a real operating system. Finally, we are investigating the possibility of exploiting our power estimates for continuous compilation systems that could modify binaries to use less power. As number of cores and power demands grow, process and thread management will be critical for efficient computing.

APPENDIX A
SPEEDING SIMULATIONS

Computer architects spend a significant amount of time and effort writing cycle-accurate simulators to have a basis for evaluating any improvements to the microarchitecture. Generally, they write the simulator to match a given real hardware configuration and validate performance based on this machine. In addition, when it comes to evaluating power and energy for the given architecture, they have to write more simulator code and incorporate a framework like Wattch [10] or Sim-Panalyzer [48]. We propose replacing such a framework with a power prediction model. Given a cycle-accurate performance simulator and the real hardware being validated against, one can form the power model from the real hardware and incorporate it into the simulator. This has two advantages. First, the predictive model can serve as a functioning power simulator while the detailed power model code is being written. Second, it can speed simulations by up to 37% when the predictive model is used in lieu of the full power framework code.

For illustrating the approach, we use XEEMU, an Intel XScale cycle-accurate simulator [24]. XEEMU simulates the runtime and power consumption of the XScale core as closely as possible. Herczeg et al. [24] form the power model to match measurements on an ADI 80200 EVB evaluation board. They validate the model using measurements on real hardware and demonstrate high accuracy for runtime, instantaneous power, and total energy consumption estimation. The average error is 3.0% and 1.6% for runtime and energy consumption, respectively.

A.1 Setup

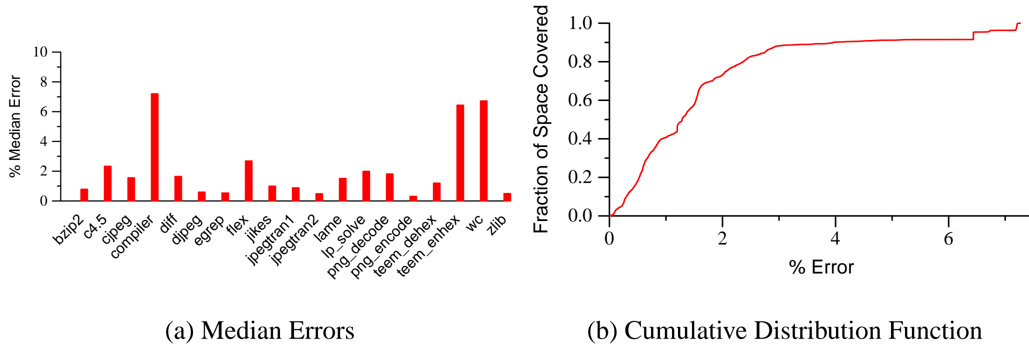
XEEMU adapts and fine-tunes the Sim-Panalyzer framework, a power estimation tool based on SimpleScalar. We rely on the high accuracy of the XEEMU power simulator to show the efficacy of our model approach. We assume power numbers produced by XEEMU to be as if they were from real hardware. We believe this to be an acceptable assumption, since the mean error when validated against real hardware is only 3%.

We compile our microbenchmarks for the XScale platform using the *gcc*-based *Wasabi* cross compiler tool chain [25]. We use the default XScale processor running at 600 MHz, with 32 KB each of data and instruction cache. We run our microbenchmarks and collect performance data from the simulator. We recall our four categories: *FP Units*, *Memory Traffic*, *Processor Stalls*, and *Instructions Retired*. We filter the performance statistics offered by the simulator and choose the following:

- dl1.misses : total number of misses
- sim_memory_dep : total number of stall cycles caused solely by the memory
- sim_num_insn : total number of instructions executed
- fpu access : number of times fpu is accessed

A.2 Evaluation

Herczeg et al. evaluate their XEEMU simulation accuracy on the CSiBE benchmark suite [19]. The suite contains various command line tools, such as data and image compressors, converters, and parsers. These programs not only test the overall accuracy of the simulator but exploit the special characteristics of the architecture, as well. The JPEG compressor (*cjpeg*) uses many shift operations. The hex encoder-decoder pair



(a) Median Errors (b) Cumulative Distribution Function

Figure A.1: XEEMU XScale Simulator Results

(*enhex, dehex*) executes many conditional block data transfer instructions. The VSL abstract machine (*vam*) rarely accesses the memory and fits in the cache. In contrast to *vam*, *minigzip* and the PNG encoder (*pnm2png*) are memory dominant programs that cause many data cache misses. All these programs are also compiled with the *Wasabi* tool chain to stand-alone binaries.

We evaluate the accuracy of our power model on the same benchmark suite. We run all benchmarks to completion and collect pertinent data. We show the error in Figure A.1(a). The prediction error ranges from 0.3% for *png_encode* to 7.2% for *compiler*. The overall median error is 1.3%, and the mean error is 1.7%. Figure A.1(b) shows the Cumulative Distribution Function for all benchmarks taken together. We find 90% of predictions to be under 4% error, 95% to be under 6.5% error, and 99.9% of all predictions to be under 7.3%. This shows excellent coverage for the model. The very small error on most predictions helps illustrate model fit. In Figure A.2, we see the speedup in simulation runtime when using the predictive model in lieu of the adapted Sim-Panalyzer framework. First, we run all benchmarks to completion on the original simulator. Next, we incorporate the power model and remove any calls to the Sim-Panalyzer framework. We graph runtimes for the modified simulator normalized to the original code. The speedup ranges from 25% for *png_encode* to 37% for *jpegtran1*.

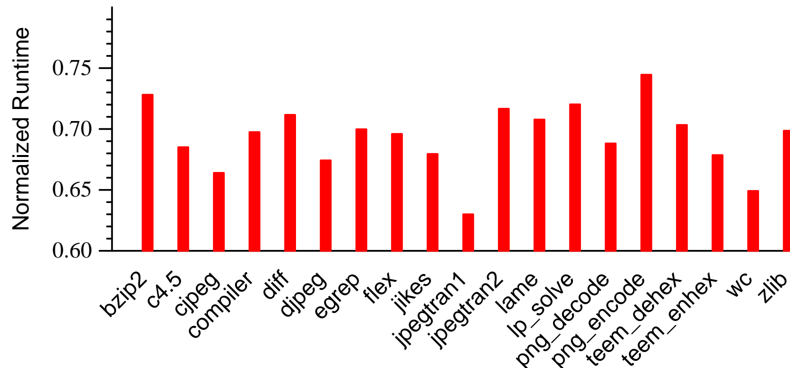


Figure A.2: Simulation Runtime for Modified Simulator Normalized to Original

A.3 Generalized Model for Multiple Frequency Levels

Table A.1: Median Errors for Per-Frequency and Generalized Power Models

Frequency (Voltage)	Per-Frequency	Generalized
200 MHz (1.0V)	1.9%	6.3%
266 MHz (1.0V)	5.6%	5.6%
333 MHz (1.0V)	4.9%	6.5%
400 MHz (1.1V)	2.9%	5.1%
466 MHz (1.215V)	4.1%	6.9%
533 MHz (1.215V)	1.5%	4.3%
600 MHz (1.32V)	1.3%	4.1%

For applications which utilize DVFS, forming a power model for every frequency supported can be very time consuming. We propose a generalized model that only needs a subset of the frequencies to be sampled. We form a piece-wise model based on microbenchmark data. We normalize each PMC, e_i, to the elapsed cycle count and get an *event rate*, r_i, for each counter. The prediction model uses these rates, frequency, F, and voltage, V, as input. We collect PMC values and temperature every 0.01s of actual execution time. For example, for a frequency of 200 MHz, we sample every 2M cycles. Since we are trying to predict power, which is an instantaneous metric, it is important to normalize the counters to the elapsed cycle count. This allows data from different frequencies to be used together to form the generalized model. We only use data from two extremes, 200 MHz and 600 MHz. We interpolate for the frequencies that fall between

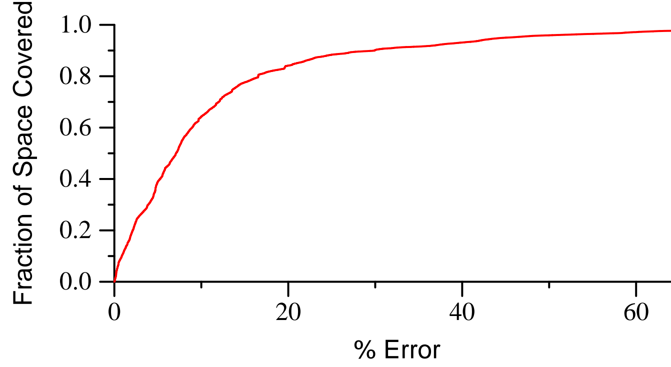


Figure A.3: Cumulative Distribution Function (CDF) of Prediction Error for the Generalized Model

these two. We model core power using our piece-wise model based on multiple linear regression. We produce the following function (Equation A.1), mapping frequency, F, voltage, V, and observed event rates r_i to core power P_{core}:

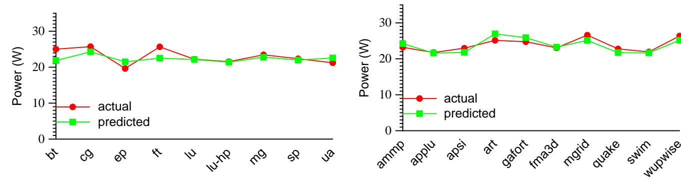
$$\hat{P}_{core} = \begin{cases} F_1(g_1(r_1), ..., g_n(r_n), F, V), & if \text{ condition} \\ F_2(g_1(r_1), ..., g_n(r_n), F, V), & else \end{cases} \tag{A.1}$$
$$\text{where} \quad r_i = e_i/(cycle\ count)$$

$$F_n = p_0 + p_1 * g_1(r_1) + ... + p_n * g_n(r_n) + p_{n+1} * F + p_{n+2} * V^2 * F \tag{A.2}$$

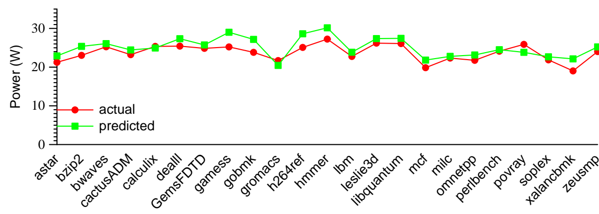
Table A.1 shows median errors. We include $V^2 * f$ as an input, since $P = 0.5 * C * V^2 * f$. This step significantly helps prediction accuracy. The *Per-Frequency* column gives median errors if we form a separate model for each frequency. The generalized model, based on data from 200 MHz and 600 MHz microbenchmark runs, gives median errors shown in the *Generalized* column. The error rates are higher, but still acceptable for applications needing power estimates. We get an estimate of the coverage of our model from the CDF in Figure A.3. We deliver an overall median error of 7.1%. Approximately 65% of predictions are under 10% error, and 85% of predictions are under 20% error.

APPENDIX B

POWER PREDICTOR RESULTS WITHOUT TEMPERATURE INPUT (AMD)

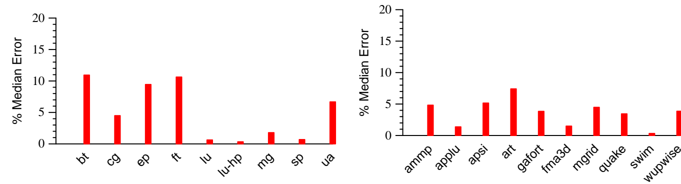


(a) NAS (b) SPEC-OMP

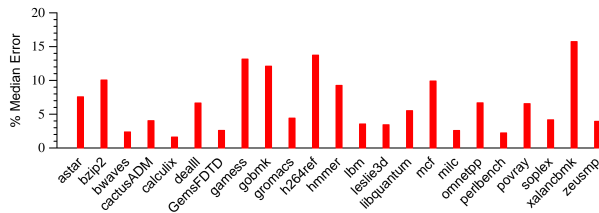


(c) SPEC 2006

Figure B.1: Measured vs. Predicted Power for AMD Phenom 9500



(a) NAS (b) SPEC-OMP

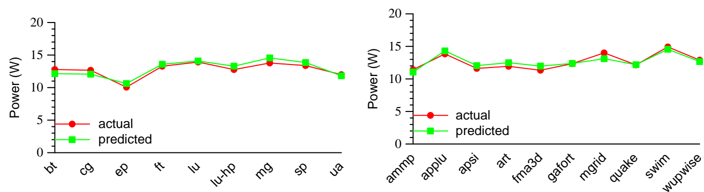


(c) SPEC 2006

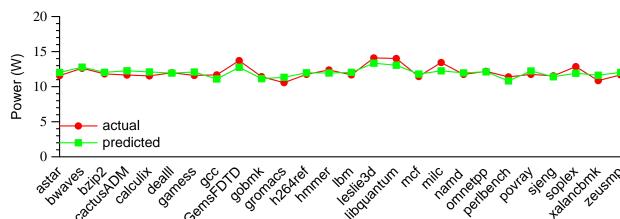
Figure B.2: Median Errors for AMD Phenom 9500

APPENDIX C

POWER PREDICTOR RESULTS WITH DVFS SCALING TO 1.1 GHz (AMD)

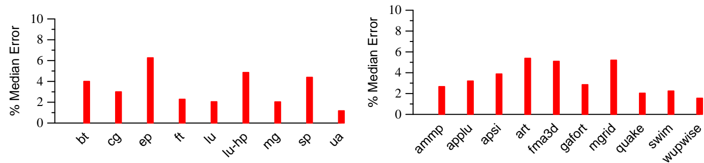


(a) NAS (b) SPEC-OMP

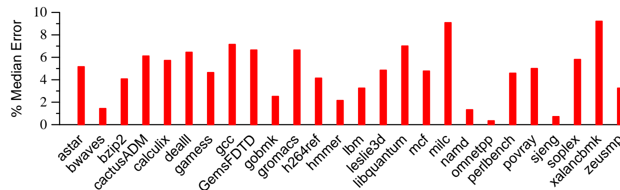


(c) SPEC 2006

Figure C.1: Measured vs. Predicted Power for AMD Phenom 9500



(a) NAS (b) SPEC-OMP

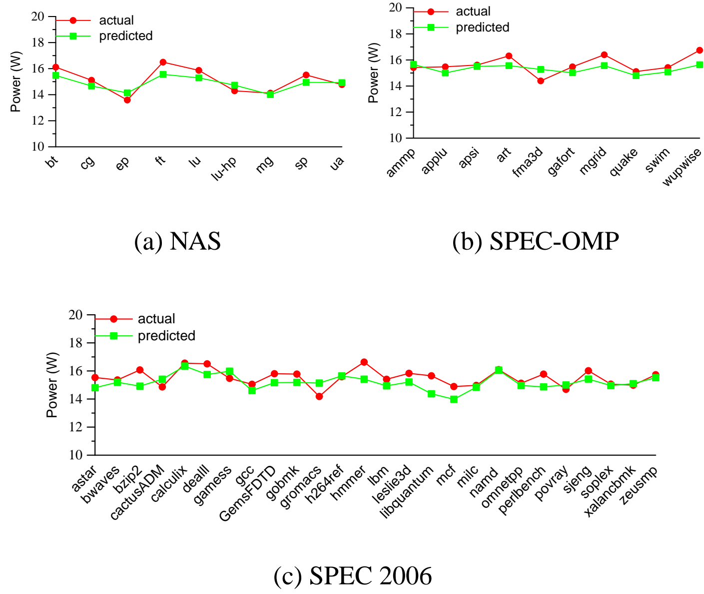


(c) SPEC 2006

Figure C.2: Median Errors for AMD Phenom 9500

APPENDIX D

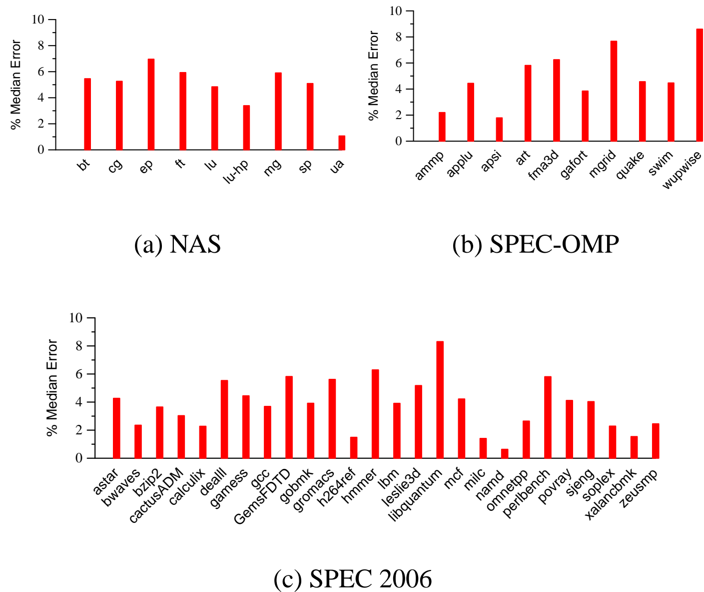
POWER PREDICTOR RESULTS WITH DVFS SCALING TO 2.0 GHz (INTEL)



(a) NAS (b) SPEC-OMP

(c) SPEC 2006

Figure D.1: Measured vs. Predicted Power for Dual Intel E5430 (8 Cores)



(a) NAS (b) SPEC-OMP

(c) SPEC 2006

Figure D.2: Median Errors for Dual Intel E5430 (8 Cores)

BIBLIOGRAPHY

[1] Intel Core 2 Quad Q6600 (Power consumption, temperature, overclocking). *http://www.behardware.com/articles/651-2/intel-core-2-quad-q6600.html*, January 2007.

[2] New Intel Core 2 Quad Q6600 CPU released. *http://www.gearfuse.com/new-intel-core-2-quad-q6600-cpu-released/*, January 2007.

[3] AMD Phenom(TM) Quad-Core Processor Die. *http://www.amd.com/us-en/Processors/ProductInformation/0,,30_118_15331_15332,00.html*, November 2008.

[4] Energy Consumption: The Processor and Cool'n'Quiet Mode. *http://www.tomshardware.com/reviews/amd-power-cpu,1925-7.html*, December 2008.

[5] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical report ucb/eecs-2006-183, EECS Department, University of California at Berkeley, December 2006.

[6] V. Aslot and R. Eigenmann. Performance characteristics of the SPEC OMP2001 benchmarks. In *Proc. of the European Workshop on OpenMP*, September 2001.

[7] T. Austin. SimpleScalar 4.0 release note. *http://www.simplescalar.com/*.

[8] D.H. Bailey, T. Harris, W.C. Saphir, R.F. Van der Wijngaart, A.C. Woo, and M. Yarrow. The NAS parallel benchmarks 2.0. Report NAS-95-020, NASA Ames Research Center, December 1995.

[9] D. Bautista, J. Sahuquillo, H. Hassan, S. Petit, and J. Duato. A simple power-aware scheduling for multicore systems when running real-time applications. In *Proc. 22nd IEEE/ACM International Parallel and Distributed Processing Symposium*, pages 1–7, April 2008.

[10] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proc. 27th IEEE/ACM International Symposium on Computer Architecture*, pages 83–94, June 2000.

[11] L. Carrington, A. Snavely, X. Gao, and N. Wolter. A performance prediction framework for scientific applications. In *International Conference on Computa-*

tional Science Workshop on Performance Modeling and Analysis, pages 926–935, June 2003.

[12] L. Carrington, N. Wolter, A. Snavely, and C.B. Lee. Applying an automatic framework to produce accurate blind performance predictions of full-scale HPC applications. In *Department of Defense Users Group Conference*, June 2004.

[13] R. Caruana, S. Lawrence, and C.L. Giles. Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping. In *Proc. Neural Information Processing Systems Conference*, pages 402–408, October 2000.

[14] G. Contreras and M. Martonosi. Power prediction for Intel XScale processors using performance monitoring unit events. In *Proc. IEEE/ACM International Symposium on Low Power Electronics and Design*, pages 221–226, August 2005.

[15] M. Curtis-Maury, F. Blagojevic, C. D. Antonopoulos, and D. S. Nikolopoulos. Prediction-based Power-Performance Adaptation of Multithreaded Scientific Codes. *Transactions on Parallel and Distributed Systems*, 19:1396–1410, October 2008.

[16] M. Curtis-Maury, J. Dzierwa, C. D. Antonopoulos, and D. S. Nikolopoulos. Online Power-Performance Adaptation of Multithreaded Programs using Hardware Event-Based Prediction. In *Proc. of the International Conference on Supercomputing*, pages 157–166, June 2006.

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

[18] P. de Langen and B. Juurlink. Leakage-Aware Multiprocessor Scheduling for Low Power. In *Proc. of the 20th International Parallel and Distributed Processing Symposium*, pages 8–15, Rhodes, Greece, April 2006.

[19] University of Szeged Department of Software Engineering. GCC code-size benchmark environment (csibe). *http://www.csibe.org/*.

[20] C. Dubach, J. Cavazos, B. Franke, G. Fursin, M. O'Boyle, and O. Temam. Fast compiler optimisation evaluation using code-feature based performance prediction. In *Proc. 4th International Conference on Computing Frontiers*, pages 131–142, New York, NY, USA, 2007. ACM Press.

[21] D. Economou, S. Rivoire, C. Kozyrakis, and P. Ranganathan. Full-system power analysis and modeling for server environments. In *Proc. Workshop on Modeling, Benchmarking, and Simulation*, June 2006.

[22] Electronic Educational Devices. Watts Up PRO. *http://www.wattsupmeters.com/*, May 2009.

[23] S. Eranian. Perfmon2: a flexible performance monitoring interface for Linux. In *Proc. 2006 Ottawa Linux Symposium*, pages 269–288, July 2006.

[24] Z. Herczeg, Á. Kiss, D. Schmidt, N. Wehn, and T. Gyimóthy. Xeemu: An improved xscale power simulator. In *International Workshop on Power And Timing Modeling, Optimization and Simulation*, pages 300–309, 2007.

[25] Intel Corporation. Wasabi systems gnu tools version 031121 for intel xscale microarchitecture. *http://www.intel.com/design/intelxscale/dev_tools/031121/wasabi_031121.htm*.

[26] Intel Corporation. Intel Core(TM) i7 Processor. *http://www.intel.com/products/processor/corei7/*, December 2008.

[27] E. İpek, S.A. McKee, B.R. de Supinski, M. Schulz, and R. Caruana. Efficiently exploring architectural design spaces via predictive modeling. In *Proc. 12th ACM Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 195–206, October 2006.

[28] E. İpek, S.A. McKee, K. Singh, R. Caruana, B.R. de Supinski, and M. Schulz. Efficient architectural design space exploration via predictive modeling. *ACM Transactions on Architecture and Code Optimization*, 4(4), January 2008.

[29] C. Isci, A. Buyuktosunoglu, C-Y Cher, P. Bose, and M. Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *Proc. IEEE/ACM 40th Annual International Symposium on Microarchitecture*, pages 347–358, December 2006.

[30] A. Jaleel, R.S. Cohn, and C. Luk. CMP$im: Using pin to characterize memory behavior of emerging workloads on cmps. In *Intel Design, Test, and Technologies Conference*, August 2006.

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

[32] R. Joseph and M. Martonosi. Run-time power estimation in high-performance microprocessors. In *Proc. IEEE/ACM International Symposium on Low Power Electronics and Design*, pages 135–140, August 2001.

[33] I. Kadayif, M. Kandemir, N. Vijaykrishnan, M.J. Irwin, and I. Kolcu. Exploiting processor workload heterogeneity for reducing energy consumption in chip multiprocessors. In *Proc. ACM/IEEE Design, Automation and Test in Europe Conference and Exposition*, volume 2, pages 1158–1163, February 2004.

[34] Lawrence Livermore National Laboratory. The ASCI PURPLE benchmark codes. *http://www.llnl.gov/asci/purple/benchmarks/*, October 2002.

[35] B. Lee and D. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *Proc. 12th ACM Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 185–194, October 2006.

[36] B. Lee, D. Brooks, B.R. de Supinski, M. Schulz, K. Singh, and S.A. McKee. Methods of inference and learning for performance modeling of parallel applications. In *Proc. ACM Symposium on the Principles and Practice of Parallel Programming*, pages 249–258, March 2007.

[37] J. Li and J.F. Martínez. Dynamic Power-Performance Adaptation of Parallel Computation on Chip Multiprocessors. In *Proc. of the 12th International Symposium on High-Performance Computer Architecture*, pages 77–87, Austin, TX, February 2006.

[38] G. Marin and J. Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. In *Proc.ACM International Conference on Measurement and Modeling of Computer Systems*, pages 2–13, June 2004.

[39] A. Merkel and F. Bellosa. Balancing power consumption in multiprocessor systems. In *Proc. of the ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 403–414, April 2006.

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

[41] M. Moudgill, P. Bose, and J. Moreno. Validation of Turandot, a fast processor model for microarchitecture exploration. In *Proc. International Performance, Computing, and Communications Conference*, pages 452–457, February 1999.

[42] K. Rangan, G. Wei, and D. Brooks. Thread motion: Fine-grained power management for multi-core systems. In *Proc. 36th IEEE/ACM International Symposium on Computer Architecture*, pages 302–313, June 2009.

[43] B. Saha, A. Adl-Tabatabai, A. Ghuloum, M. Rajagopalan, R. Hudson, L. Petersen, V. Menon, B. Murphy, T. Shpeisman, E. Sprangle, A. Rohillah, D. Carmean, and J. Fang. Enabling Scalability and Performance in a Large Scale CMP Environment. In *Proc. of the European Conference on Computer Systems*, pages 73–86, March 2007.

[44] H. Sasaki, Y. Ikeda, M. Kondo, and H. Nakamura. An Intra-Task DVFS Technique based on Statistical Analysis of Hardware Events. In *Proc. of the International Conference on Computing Frontiers*, pages 123–130, Ischia, Italy, May 2007.

[45] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proc. 10th ACM Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, October 2002.

[46] K. Singh, M. Curtis-Maury, A. Shah, S.A. McKee, F. Blagojevic, D.S. Nikolopoulos, B.R. de Supinski, and M. Schulz. Comparing scalability prediction strategies on an smp of cmps. Technical Report CSL-TR-2009-1054, Cornell Computer Systems Lab, May 2009.

[47] Standard Performance Evaluation Corporation. SPEC CPU benchmark suite. *http://www.specbench.org/osg/cpu2006/*, 2006.

[48] D. Grunwald T. Mudge, T. Austin. The simplescalar-arm power modeling project. *http://www.eecs.umich.edu/ panalyzer/.*

[49] M. Tremblay and S. Chaudhry. A third-generation 65nm 16-core 32-thread plus 32-scout-thread sparc processor. In *International Solid-State Circuits Conference, Digest of Technical Papers*, pages 82–83, February 2008.

[50] P.A. Castillo Valdivieso, J.J. Merelo Guervós, M. Moretó, F.J. Cazorla, M. Valero, A.M. Mora, J.L. Jiménez Laredo, and S.A. McKee. Evolutionary system for prediction and optimization of hardware architecture performance. In *Proc. IEEE Congress on Evolutionary Computation*, pages 1941–1948, May 2008.

[51] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain S. Venkataraman, Y. Hoskote, and N. Borkar. An 80-tile 1.28TFLOPS Network-on-Chip in 65nm CMOS. In *Proc. of the International Solid State Circuits Conference*, pages 98–99, February 2007.

[52] V.M. Weaver and S.A. McKee. Can hardware performance counters be trusted? In *Proc. IEEE International Symposium on Workload Characterization*, pages 141–150, September 2008.

[53] W. Wu, L. Jin, and J. Yang. A systematic method for functional unit power estimation in microprocessors. In *Proc. 43rd ACM/IEEE Design Automation Conference*, pages 554–557, July 2006.

[54] T. Yang, X. Ma, and F. Mueller. Cross-platform performance prediction of parallel applications using partial execution. In *Proc. IEEE/ACM Supercomputing International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 40–50, November 2005.

[55] R.M. Yoo, H. Lee, K. Chow, and H.S. Lee. Constructing a non-linear model with neural networks for workload characterization. In *Proc. IEEE International Symposium on Workload Characterization*, pages 150–159, October 2006.