Workload Adaptive Power Management with Live Phase Monitoring and Prediction

Canturk Isci

A Dissertation

Presented to the Faculty

of Princeton University

in Candidacy for the Degree

of Doctor of Philosophy

Recommended for Acceptance

by the Program in

Electrical Engineering

September 2007

Abstract

In current computer systems, power dissipation is widely recognized as one of the primary critical constraints. Improving the power efficiency of current and emerging systems has therefore become a pressing challenge and an active research area over recent years. Dynamic, on-the-fly management techniques aim to address this challenge by adaptively responding to the changes in application execution. These application patterns, commonly referred to as "phases", expose distinct, dynamically-varying and often repetitive characteristics of workloads. Dynamic management techniques, guided by workload phase information, can effectively tune system resources to varying workload demands for improved power-efficiency.

This thesis researches new methods to characterize and predict application behavior for a dynamic power management endgoal. Specifically, this work has two major thrusts. First, it explores different approaches to characterize and predict dynamically varying workload power behavior. Second, it discusses runtime management techniques for real systems that can proactively adapt processor execution to varying application execution characteristics.

This work develops a runtime, real-system power model that provides processor power consumption details in terms of the component powers of different architectural units. We show that similarity analysis methods applied to these component powers help expose power phase behavior of applications. A small set of "power signatures" can represent overall workload power characteristics within 5% of the actual behavior. We develop a "transition-guided" phase detection framework that can identify repetitive application phase patterns despite system-induced variability effects. This detection strategy can identify recurrent phase signatures with less than 5% false alarms on running systems. Last, we propose a workload-adaptive dynamic power management framework guided by runtime phase predictions. This predictive power management approach is shown to improve the energy-delay product of a deployed platform by 7% when compared to existing reactive techniques and by 27% over the baseline unmanaged system.

Overall, this thesis shows a roadmap to effective on-the-fly phase detection and prediction on real-systems for application to workload-adaptive dynamic power management. With the increasing focus on adaptive and autonomous system management, this research offers practical techniques that can serve as integral components for current and emerging power-aware systems.

Acknowledgements

This dissertation would not have existed without the guidance and support of my advisor, Margaret Martonosi. She has been an invaluable inspiration to me as a researcher, a mentor and a writer throughout my graduate study. I am deeply thankful for her extensive advice on all aspects of my research from developing research ideas to presenting outcomes. Her patient and positive approach has been my primary source of motivation during challenging periods. I believe, working with and learning from Margaret has made me a better academic, and has been one of the greatest privileges of my life at Princeton.

I would like to thank my dissertation committee, Doug Clark, Li-Shiuan Peh, Sharad Malik and Ricardo Bianchini, for their insightful comments on this work. Their feedback has been very valuable for me to improve this thesis. I am grateful to Doug and Li-Shiuan for their timely feedback and many excellent suggestions on the previous drafts of this dissertation.

I owe many thanks to my great lab mates in the “mrmgroup”, including Zhigang Hu, Russ Joseph, Philo Juang, Fen Xie, Ting Liu, Qiang Wu, Yong Wang, Hide Oki, Gilberto Contreras, James Donald, Chris Sadler, Pei Zhang, Eric Chi, Abhishek Bhattacharjee, Vincent Lenders, Maria Kazandjieva, Carole Wu and Manos Koukoumidis. I am grateful for their support in countless paper revisions, practice talks and research discussions. I have learnt a great deal from them over the years, and their friendship and unique personalities have made my lab life a lot more enjoyable.

I was also fortunate to collaborate with some great researchers in industry. I thank Pradip Bose, Alper Buyuktosunoglu, Eugene Gorbatov and Rick Forand for their mentorship and encouragement. I gained tremendous knowledge and experience working with them during my graduate study.

I would also like to thank the members of our department staff that I have known over the years, including Karen Williams, Sarah Griffin, Tamara Thatcher, Anna Gerwel, Stacey Weber, Sarah Braude, Meredith Weaver and Roelie Abdi-Stoffers. Their friendly person-

alities have always brought cheer to my daily departmental routine. I am grateful for their patience and help with my countless inquiries during my study at Princeton.

The years I have spent at Princeton have been particularly memorable thanks to my friends who had shared the many ups and downs of my life over the years. I thank Mehmet Ekmekci, Mert Rory Sabuncu, Fatih and Aysen Unlu, Filiz Garip, Oguzhan Karakas, Sinan Gezici, Murat Fiskiran, Vassos Soteriou, Erhan Bayraktar and Alp Atici for their endless support.

This research benefited from generous financial support from the National Science Foundation, the Semiconductor Research Corporation, the New Jersey Council of Science and Technology, Intel Corporation, and IBM Research. In addition, I would like to acknowledge the support of the Gigascale Systems Research Focus Center, one of five research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program.

Last but not least, I would especially like to thank my parents, Sevim and Coşkun, my sister Dilem, and my greater family, including my grandparents, aunts, uncles and cousins for their constant love and support. I am deeply grateful for their guidance, humor and encouragement. They will always be my main inspirations in every aspect of my life. Finally, I would like to thank my Belma, who has always been the calming voice in my mind, for seeing the best in everything and for being beside me all these years.

To my family.

Contents

Chapter 1

Introduction

1.1 Background and Motivation

Computing systems have experienced a tremendous sustained growth in performance and complexity for more than two decades. Exponentially increasing transistor integration enables more devices to be packed within single chips, which in turn provides more functionality and state with each generation of processors. Figure 1.1 illustrates this for a range of processor families [12, 36, 53, 56, 141, 144]. Moreover, reduced process dimensions enable faster switching transistors, driving higher operating frequencies with each generation. Coupled with technology advances, new architectural and compiler techniques have pushed the performance bar even higher with deeper pipelines, high speculation, out-of-order and superscalar microarchitectures, and increasing instruction-level parallelism. In addition, new simultaneously multithreaded and multicore systems enable thread-level parallelism [66, 134, 156, 157, 170]. All of these advances translate into more computations per unit time with each new computer generation.

From a historical perspective, these have been tremendous forward progress in computing performance. By leveraging both technological and architectural advances, microprocessor designers have been able to actually surpass the performance trends indicated by Moore's Law [129, 133]. For example, when we look at the reported performance results with the SPEC CPU2000 benchmarks between 2000 and 2006, we see more than 10-fold

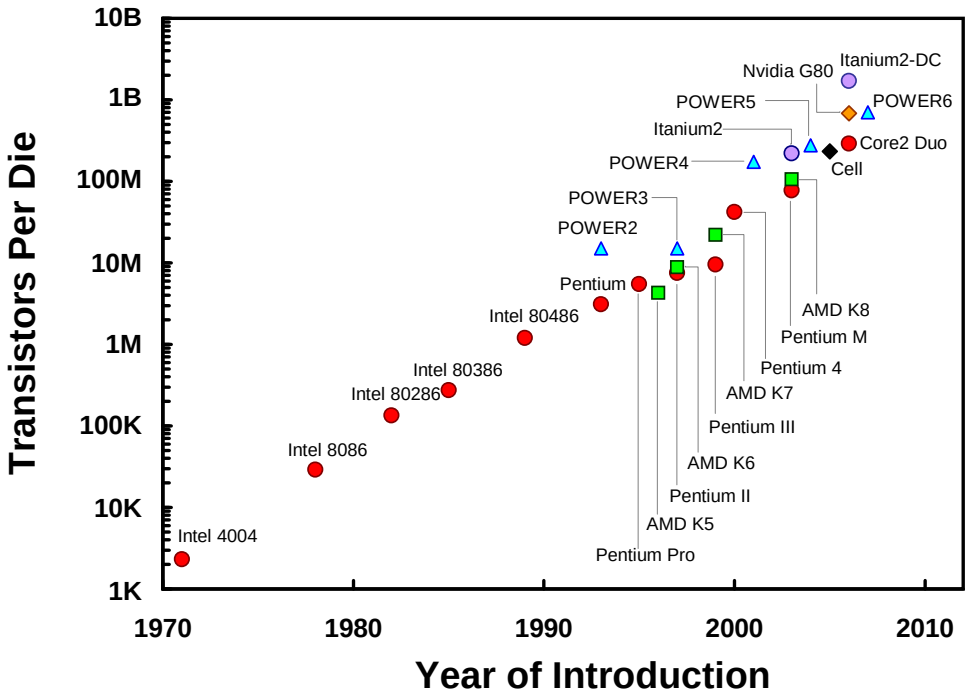


Figure 1.1: Number of transistors within a die over time.

increase in integer performance and 14-fold increase in floating point performance for Intel family processors [165]. This unabated push towards higher performance and reduced form factors has provided currently emerging mobile devices with computing capability that was previously confined to mainframe systems.

Nonetheless, this forward progress in performance has not come for free. Together with increasing clock rates and performance capabilities, the power dissipation of computing systems has also accelerated rapidly. Figure 1.2 illustrates this for Figure 1.1's processor families over the same time period [12, 31, 53, 57, 141]. As this figure demonstrates, processor generations also experienced an exponential increase in power density. This increase in power density has recently become one of the primary constraints in microprocessor design. First, stemming from both increased power dissipation and widespread adoption of personal computers, the overall energy impact of computing systems has become an important issue. Once again looking from a historical perspective, the total worldwide processor power dissipation of personal computers increased by more than 50 times over the last decade [173]. Second, increasing power density has also directly influenced thermal

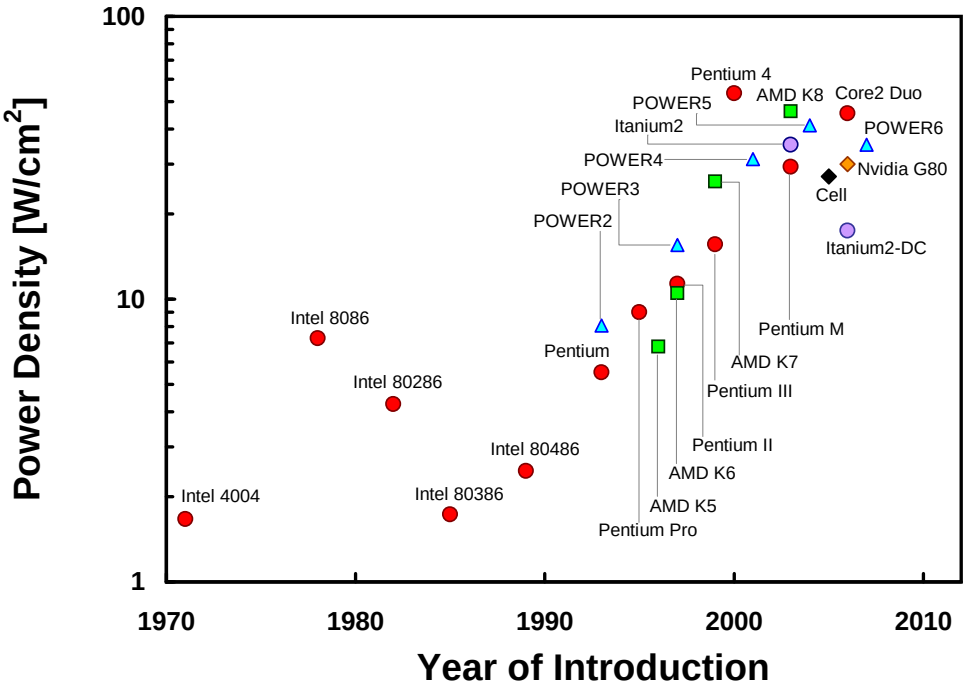


Figure 1.2: Processor power density over time.

limitations of processors, requiring advanced cooling and thermal management strategies [58, 155]. Third, increasing power demand, as well as the temporal and spatial power variations within microprocessors have produced significant strain on effective and reliable power delivery [92, 141]. Last and more recently, the financial and environmental impacts of computing system power dissipation has also been widely acknowledged. Especially in large-scale data centers, the current annual cost of power delivery and cooling has reached to the order of millions of dollars. If the current trend—that advances in computing performance are accompanied with rising power demand—continues in the next generation systems, the ongoing costs of power and cooling can soon surpass the initial cost of the underlying computing hardware by a growing margin [12]. To address the impacts of computer power dissipation, the Environmental Protection Agency has recently announced new specifications for computer power-efficiency [171]. Based on the projections of these specifications, improving the energy-efficiency of computing systems can potentially achieve $1.8 billion of total energy cost savings over the next five years. Moreover, such emphasis on computing power can eliminate greenhouse gas emissions equivalent to the annual

emissions of 2.7 million cars.

Interestingly, this is not the first time the computing industry has faced the power challenge. Early mainframe systems that relied on bipolar devices had experienced a similar exponential growth in power until the early 1990s, at which point the mainframe industry had to move towards CMOS devices that enabled an order of magnitude improvements in power densities [146]. Less than two decades later, we have once again approached the limits of power density. As CMOS technology continues to be the viable design option for microprocessors, there is a growing necessity to devise and employ effective power management techniques in all levels of computing systems, from circuits and architectures to systems and software. Indeed, recent years have unveiled numerous research efforts that aim to address power-efficiency at all levels of abstractions.

These different power-management strategies can be categorized as either static and dynamic management approaches. Static, or offline, techniques involve design-time decisions, profile-based optimizations and compiler-driven management responses. These approaches are employed at various design stages and abstraction layers. These include circuit-level techniques such as transistor reordering and dual-threshold circuits [104, 118, 161], architectural mechanisms such as profiling-based adaptations at subroutine granularities or execution checkpoints [7, 75], systems- and application-level approaches such as task partitioning and stretching, deadline-based scheduling, software transformations and remote execution [43, 102, 114, 164], and compiler-driven management techniques that involve profiling and instrumentation of applications with power management hints or state keeping instructions [1, 65, 71, 122, 154, 180].

Dynamic, or online, power management techniques involve runtime control mechanisms in hardware or software; they tune the configurable computing resources during execution. There is a large variety of dynamic management techniques across the whole spectrum of computing systems hierarchy, spanning from circuit level techniques to application and compiler level power management. Circuit-level adaptations include techniques such

as adaptive body biasing and multi-threshold CMOS circuits (power gating) [4, 97, 98]. Architectural power management techniques involve pipeline reconfigurations [3, 8, 26, 90, 139, 153], adaptive cache scaling and decay [41, 48, 96, 140], pipeline-delay-based supply voltage tuning [47], speculation control [23, 123], multiple clock domain architectures [147, 178] and management techniques for chip multiprocessors [94, 103, 115]. At the system-level many power-aware adaptations exist that target at dynamic management of the system operation and the underlying platform components. One of the most widely used dynamic power management techniques at the system level is workload-dependent dynamic frequency and voltage scaling [33, 176]. Some other employed dynamic power management techniques are adaptive disk control [60], energy-efficient I/O and memory management [110, 162, 136, 143, 177, 186], task-level energy budgeting [5, 20, 119] and power-aware scheduling [67, 127]. In addition to system-level management approaches, there are also some power-aware dynamic compilation techniques [73, 172, 179].

Static approaches generally have the broad view of the entire application, and lead to simpler control. However, they lack the actual dynamic execution information of applications. Many software-level static management approaches also require prior profiling of applications or recompilations to incorporate compiler directives. In contrast, dynamic techniques are directly exposed to the dynamic execution behavior and can guide management responses on-the-fly. However, the major drawback of these online techniques lies in their limited view of application execution as they cannot know *a priori* the whole application structure. In general, dynamic management also necessitates more elaborate monitoring and control schemes to track execution characteristics and to apply management responses. Nonetheless, as the need for aggressive power management continues to increase, such control mechanisms become more attractive in emerging systems despite the design effort they require. In particular, as current workloads exhibit highly variable and nondeterministic characteristics, and as the pool of legacy applications grows, static techniques bring limited benefits. Dynamic management techniques offer significant additional

improvements in overall system power efficiency.

My research particularly aims to leverage the broad view of application execution at runtime by monitoring architectural characteristics of applications and inferring dynamically-varying workload behavior. I use observed runtime workload characteristics to detect and predict repetitive application execution and this repetitive behavior information guides dynamic management techniques. One of the primary drivers of dynamic power management is the inherent variability in both the running workload demands and the underlying computing systems. Efficiently matching the underlying resources to the dynamically varying application demands by adaptively configuring these computing structures is a powerful enabler for power-efficient computation. My dissertation research focuses on two important research challenges for such workload-adaptive and dynamically-controlled execution:

(i) Developing accurate and practical characterizations of dynamically varying workload demands and correctly projecting future behavior.

(ii) Efficiently managing the dynamic configurations of the underlying computing resources based on projected workload demand.

One primary focus of my dissertation research is to bring real-system experimentation and validation with real measurements into architecture research. In the following chapters of this dissertation, I provide an overview of the different research aspects and the accomplishments of my research along these two thrusts.

1.2 Research Overview

My dissertation research explores architectural and real-system techniques to characterize and predict wide-scale power behavior of programs and develops autonomous methods that track and predict dynamically-varying workload characteristics to guide runtime, workload-adaptive power management techniques. Many of the presented studies aim to explore and leverage the *phase behavior* of workloads. This phase behavior represents the temporal variations in workload behavior that are commonly observed during execution.

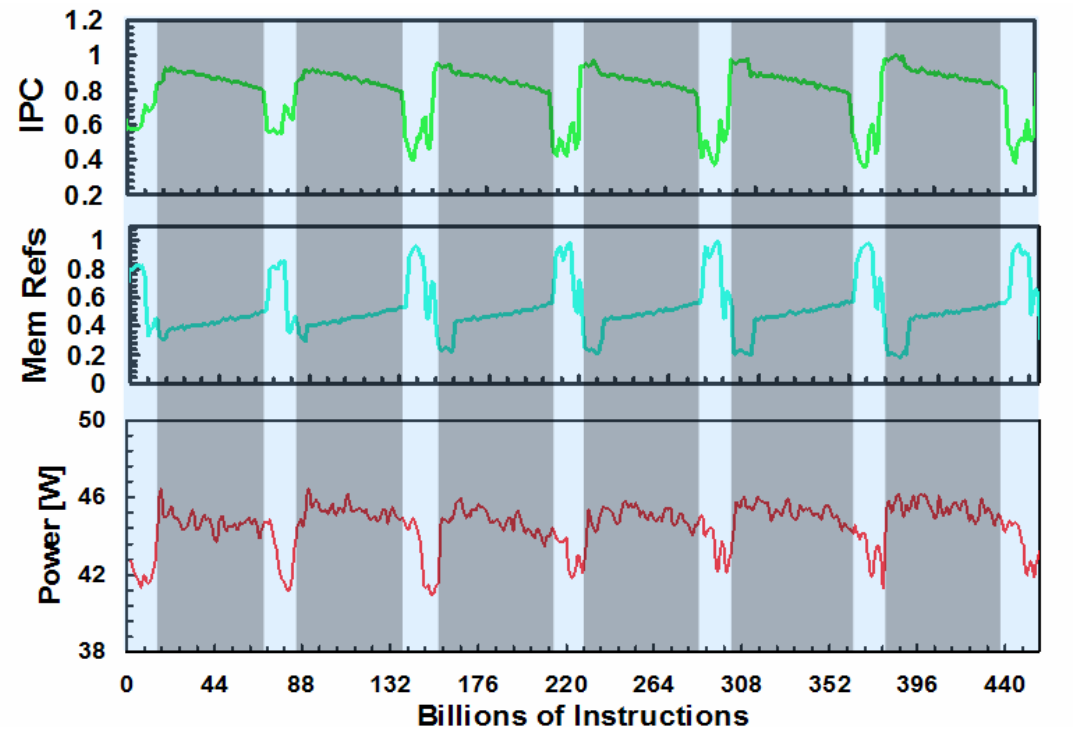


Figure 1.3: Phase behavior as observed from the measured performance metrics and power for the `vortex` benchmark. This execution snapshot can be roughly separated into two phases that repeat throughout benchmark execution.

These workload phases are known to exhibit repetitive patterns due to the iterative nature of dynamic execution and can be observed in various forms such as performance characteristics, power consumption and traversed execution address space. Moreover, different phase patterns can be observed at different phase granularities from a few hundred of instructions to billions of instructions. Figure 1.3 shows an example of this phase behavior with an execution snapshot from the SPEC CPU2000 `vortex` benchmark when its execution characteristics are classified into two major phases. In this example, the three charts show the phase behavior for `vortex` for two performance metrics as well as with the actual measured power behavior.

From a high-level perspective, my thesis research contributes to existing literature in four related research areas:

- First, it presents one of the first real-system frameworks for modeling microarchitecture-level power consumption of modern processors at runtime.

- Second, I describe workload phase analysis methodologies that target characterizing the dynamically-varying power behavior of applications.
- Third, my research is one of the first real-system phase analysis studies that tackles the problem of identifying repetitive execution characteristics despite the system-induced variability effects. In this direction, I propose novel phase characterizations and present effective techniques that mitigate the sampling and variability effects.
- Fourth, my work demonstrates a complete predictive dynamic management system that predicts application characteristics at runtime and performs autonomous system adaptations to improve power efficiency based on projected workload demand.

Moreover, in this dissertation I describe three different real-system infrastructures that I developed for experimentation and evaluations. These infrastructures are deployed in running systems for remote power monitoring and estimation, phase analysis with dynamic instrumentation and real-measurement feedback, and phase-prediction-driven dynamic power management. Below, I provide an overview of each of these four major aspects of my dissertation research, which are detailed in the subsequent chapters of this thesis.

1.2.1 Live, Runtime Power Estimation

The ability to measure or model processor power dissipation lies at the heart of power-oriented computing research. At the architecture level, much of this is performed via simulator infrastructures. These either perform analytical power derivations for architecture components based on technology parameters [24] or use empirical power model macros derived from lower-level production simulators [21]. Regardless of the taken approach, the architectural power modeling principle remains similar, where the derived maximum component powers are scaled with component utilization rates and architectural parameters to form component-level power estimates. Together with holding or idle power at zero utilization, these power estimates can then approximate the processor power dissipation. While

such simulation-oriented techniques provide extensive detail, they are generally prone to limited absolute accuracy, they are impractical for long-timescale simulations and they often consider applications in an isolated environment, thus lacking the effects of underlying system events. Real system measurements can remedy these shortcomings [51, 142, 168]. However, they generally lack the architectural detail provided by simulations and focus only on total power dissipation.

This line of my research explores an alternative approach to modeling processor power consumption that aims to leverage the advantages of both domains. I propose a real-system power measurement and estimation approach that can also provide microarchitecture-level detail. Fundamentally, this power modeling approach is similar to the simulation approach, where we consider maximum component powers scaled with activity factors. However, instead of cycle-level accounting, my technique relies on hardware performance monitoring events to track component activity. Moreover, I develop this as a runtime power estimation strategy that operates at native application execution speed. I use real power measurement feedback to calibrate power estimators, to incorporate nonlinear power behavior of processor components due to baseline power management techniques and to provide a validated absolute estimation accuracy. While there are prior studies that also investigate event-counter-based power estimations [13, 93, 95], these studies do not focus on the distribution of power to the architectural components. Furthermore, they only consider processors with small power variation. My work provides both validated total power estimates and their decomposition into architectural components. These estimates are evaluated on a high-end system with aggressive speculation and baseline power saving techniques, where the observed power at different execution regions can vary by as much as 600%. This runtime power estimation framework can approximate processor power behavior within 5% of actual power consumption, as validated with simultaneous real measurements.

1.2.2 Phase Analysis for Power

In recent years, there has been a growing interest in application phase behavior. Part of this interest focuses on identifying workload phases for characterization purposes and summarizing execution, while others explore methods to detect phases at runtime to guide dynamic adaptations [6, 41, 72, 90, 152, 153]. With such phase-based adaptations, computing hardware and software can be tuned at runtime to the demands of different program phases. Prior research has considered a range of possible phase analysis techniques, but has focused almost exclusively on performance-oriented phases. Moreover, the bulk of phase-analysis studies have focused on simulation-based evaluations. However, effective and practical analysis of application phase behavior on real-systems is essential to employing these phase-based adaptations on running systems. In addition, there is generally a missing link between phase characterizations and their ability to represent power behavior. Such power characterization is very important especially for dynamic power and thermal management, providing a direct relation between dynamic workload execution and its impact on processor power consumption.

In this thesis I describe a phase analysis methodology that is targeted directly towards characterizing workload power behavior. This approach uses the temporal similarity among estimated component power dissipations to discern the phase patterns in workload power behavior. The power phase characterizations acquired with this method capture the power variations during workload execution within 5% of actual measurements using a small set of representative phases. These phases generally summarize overall execution with less than 1% of the complete execution information. I develop a novel real-system framework for power-oriented phase analysis that coordinates performance monitoring, power estimations, dynamic instrumentation and real power measurements. With this evaluation infrastructure I demonstrate the comparative benefits of different phase characterization techniques that utilize control-flow or event-counter features of applications. This part of my work shows that while both features reveal significant insights to power phase behav-

ior, event counter features further provide 33% improvements in the characterization of workload power variations.

1.2.3 Mitigating System Induced Variability Effects on Real-System Phase Detection

One primary requirement for the application of phase-based dynamic adaptations is the ability to discern repetitive execution. Detecting repetitive phases in application execution helps apply dynamic management responses proactively, thus improving their overall effectiveness. Real system experiments bring additional challenges to the detection of such repetitive behavior due to system induced variations. Therefore, it is essential to understand how these indeterministic system events alter workload phases from phase to phase and from run to run. Consequently, for a phase detection technique to be effective on real systems, it should be resilient to these variability effects.

This part of my work examines the phase behavior of applications running on real systems to reliably discern and recover phase behavior in the face of application variability stemming from real-system and time sampling effects. I discuss and classify the extent and type of the alterations application phases experience with real-system experiments. I propose a set of new, "transition-based" phase detection techniques. These techniques can detect repetitive workload phase information from time-varying, real-system measurements with less than 5% false alarm probabilities. In comparison to previous detection methods, my transition-based techniques achieve on average 6-fold improvements in phase detection efficiency by mitigating the system induced variability effects.

1.2.4 Runtime Phase Tracking and Phase-Driven Dynamic Power Management

One of the primary motivations for doing power management dynamically is the highly variable phase behavior within applications at different execution regions. Dynamic management techniques highly benefit from this application phase behavior, which can help identify workload execution regions with different characteristics, and thus can dictate different dynamic management responses. Most existing dynamic management techniques

respond to these phase changes *reactively*. When they observe a noticeable deviation from previous application characteristics, these techniques adjust the underlying system configurations dynamically, assuming this recent behavior will persist in future execution [33, 41, 90, 162, 176, 186]. These approaches have difficulty however, when applications change characteristics at a high rate. In such cases recognizing and predicting phases on-the-fly provides better adaptation of the applied dynamic configurations. Therefore, it is important to develop methods to identify and predict repetitive phases, to *proactively* apply dynamic management responses.

My work develops online phase prediction methods that can be applied in running systems and demonstrates how these runtime phase predictors can effectively guide dynamic, on-the-fly processor power management. I describe a general-purpose phase prediction framework that can be configured for different power-performance trade-offs and can be utilized to track various application characteristics for the desired management actions. This phase predictor operates at runtime with negligible overheads and autonomously tracks and predicts application phases. These phase predictions can be employed to guide various management techniques. In my real-system experiments I demonstrate their benefits with dynamic voltage and frequency scaling (DVFS) as an example technique. I implement this complete runtime phase prediction and phase-driven dynamic adaptation infrastructure on a mobile laptop platform. Compared to existing reactive and statistical approaches, our phase predictor significantly improves the accuracy of the predicted workload behavior, reducing the misprediction rates by 2.4X for applications with variable behavior. My experiments demonstrate that DVFS-based dynamic management improves the energy-delay product of the experimental system by 27% on average, when guided by my runtime phase predictor. Compared to prior reactive approaches, these dynamic adaptations improve the energy-delay product of applications by 7%, while incurring less performance degradation.

1.3 Literature Review

This section gives a general overview of existing work related to my thesis research. Each of the following chapters provides more detailed discussions of prior work specific to each of the presented studies. Here, I discuss related literature along the main areas of contribution discussed above. These are categorized under three areas: processor power modeling, workload characterization and phase analysis, and workload-adaptive power management.

1.3.1 Processor Power Modeling

Earlier work on processor power modeling involves power measurement feedback for software and instruction-level power models. These include instruction energy tables and inter-instruction effects for processor and memory [113, 126, 168]. Software power models aim to map energy consumption to program structure [51, 142]. In general, these techniques are employed in simpler or embedded processors with minimal clock gating and power management that exhibit low temporal variations. In these cases, the power behavior largely depends on the operating frequency and voltage [28] and simple table-based approaches provide good approximations to processor power behavior.

Architectural and functional module-level power modeling has also been prevalent in power-aware computing studies. These have focused mostly on high-level abstractions of processor components. These abstractions encompass energy consumption models driven by functional unit complexity, profiled averages or switching activities particular to different units [105]. Starting from simple average-case estimates [145], these power estimators evolved into activity and lookup based power models [106, 107] that can also incorporate inter-module interactions [125]. As more capable and detailed execution- or trace-driven architectural simulation tools became available, accompanying cycle-accurate power modeling tools have also been developed.

Among different power estimation frameworks, here I mention several of the most commonly used models. *Wattch* is a processor power modeling infrastructure that relies

on parameterized power models for different processor building blocks such as array and associative memory structures, logic, interconnect and clock tree [24]. *SimplePower* is another cycle-accurate energy estimation tool that uses energy models together with switch capacitance tables for each microarchitectural unit [175]. These approaches use analytical energy models that rely on circuit capacitance parameters. In contrast, *PowerTimer* uses an empirical energy estimation model based on circuit-level energy models derived from low-level simulations [21]. Last, *SoftWatt* provides a full-system power model, including the processor and the complete memory hierarchy [59].

More recently, there has been growing interest in runtime architectural power modeling on real-systems. These approaches enable power estimations for the long timescales that are required for system-level and thermal adaptations. Since these approaches lack extensive simulation-style detail, they rely on supporting hardware or software functionality such as performance counters to drive power estimations. Prior work demonstrates that several performance monitoring events correlate highly with processor power dissipation [13]. These events can be configured to track and estimate processor power behavior and can be used to infer the distribution of power to microarchitectural components [93, 95, 176]. This runtime information is used in conjunction with analytical models for detailed component-level power estimates [18, 19, 34, 111]. Simple runtime models are also employed to track the operating system's contribution to power consumption [116]. While the above approaches consider fixed, static power models, adaptive, feedback-driven power estimation models have also recently been explored [61]. As power dissipation and thermal limitations become pressing issues in large-scale systems, such runtime models are also emerging in the server and cluster domains to enable efficient monitoring and dynamic management of large-scale systems [45, 63].

In runtime power modeling, my work is one of the first studies that provides microarchitecture-level power estimations on real systems for a high-end, highly speculative processor. I develop power estimation models that track the power consumption of microar-

chitectural units in all execution regions with high or low processor utilization. Moreover, my work presents a complete power modeling and validation framework including remote runtime monitoring and real-time power measurement feedback.

1.3.2 Workload Characterization and Phase Analysis

There is a large body of existing work related to workload characterization and the analysis of application phase behavior. These studies can be classified under various themes such as online and offline approaches, simulation-based and real-system characterization, characterizations with different workload features and for different endgoals.

One set of existing research employs different characterization techniques to summarize execution with representative regions or phases. Some of these techniques use simulations to classify workload execution based on programmatical information (such as executed instruction addresses and visited basic blocks) [32, 40, 72, 151, 152] or performance characteristics [35, 46, 101]. Another line of phase characterization research focuses on real-system studies that track hardware events or dynamic program flow [6, 29, 108, 128, 131, 132, 169]. Several of these studies employ a wide range of similarity measures and clustering methods such as k-means, regression trees, principal or independent component analysis for online or offline classification of execution into self similar regions.

A major area of research focuses on monitoring and detecting workload phase behavior for dynamic adaptations [68]. These studies use various workload features and evaluation techniques in their analyses. Part of these studies focus on different indicators of dynamic program flow to monitor varying workload characteristics such as branch counts [90], working set signatures [41], traversed basic blocks [109, 153] and visited subroutines [75]. These approaches track patterns in execution flow to trigger suited dynamic management responses that employ various architectural reconfigurations. In addition to the above simulation-oriented studies, some real-system studies consider detecting specific application behavior for dynamic responses. These works track application phases to control management schemes readily available in current systems such as voltage and frequency

scaling [176, 179], to detect changes in execution space and to drive dynamic optimization strategies in runtime systems [38, 100, 120].

Application phase monitoring and detection guides dynamic adaptations to react to the changes in observed characteristics. Once the new behavior is detected, corresponding responses in tune with the demands of the new phase can be activated. However, *predicting* this change in application characteristics can provide additional benefits by initiating management proactively. This is especially important in the case of quickly varying application behavior, where the fundamental frequency at which the application phases change is close to the sampling rate of the tracked characteristics. Existing research has employed different strategies to predict varying workload characteristics. Compiler- and application-level techniques develop static, analytical models based on program structure to predict changes in workload characteristics such as memory access patterns [52, 54]. Several prediction schemes that dynamically update their decisions during workload runtime have been proposed at the systems and architecture levels. At the system level, both statistical and table-based approaches that predict specific workload characteristics based on previous history have been proposed [44]. In addition, memory related runtime phase predictors based on memory reuse distance patterns [150], as well as dynamic code region based phase predictions [99] have been studied in prior related work. In architectural studies, the ability to propose hardware support has led to more elaborate phase prediction mechanisms. Run-length and control-flow based phase predictors have been developed with hardware support to predict phases in the dynamic execution space of applications [153]. In addition to predictors of future workload phases, alternative schemes that predict phase changes and durations have also been employed in architectural implementations [109]. Overall, these works demonstrate effective prediction techniques across a wide range of granularities, with variety of workload features spanning both hardware and software mechanisms.

My research contributes to the existing body of phase analysis work in characterization, detection and prediction of application phases with a primary focus on real-system

phase analysis methods. While most of the existing phase characterization work focuses on performance behavior of workloads, my thesis presents new techniques to identify power phase behavior of applications using hardware performance monitoring features. It develops novel strategies to detect repetitive application phases on real systems in spite of the system-induced perturbations on workload characteristics. Last, my work demonstrates a fully-autonomous, real-system phase prediction infrastructure that predicts future phase behavior of applications at runtime by leveraging the pattern behavior in execution phases.

1.3.3 Workload-Adaptive Power Management

Earlier in this chapter, I have discussed the extensive range of research broadly in the area of dynamic management, spanning from circuits to systems and applications. Here I review some of these approaches that particularly aim to tune system execution to the dynamic changes in the workload characteristics. I discuss related work in workload-adaptive power management under three abstractions: compiler- and application-level techniques, system-level management and architectural adaptations.

High-level workload adaptations involving compilers and applications give high-level software more responsibility for power management. Typically, these approaches can operate in two opposite directions. First, part of the existing work has developed strategies to adapt the workloads themselves for varying power constraints by providing different degrees of quality of service. These adaptations include application features with different qualities or optional application steps that are activated only at high energy settings. Some techniques also involve choosing between local and remote program or data components based on their power-performance trade-offs [50, 102, 143]. This first direction deliberately induces changes in workload characteristics to respond to energy constraints, and can be referred to as power-driven workload adaptations.

In the second direction, several techniques have considered employing special directives within applications to guide lower-level power management. Such directives are introduced via compiler support or specialized application programming interfaces to perform

bookkeeping operations about application characteristics [1, 7], to insert offline profiling information for code regions at different power management states [71, 122, 154] and to inform the underlying system layers about different application operations such as I/O intensive regions [65, 177].

System-level power management techniques are applied in two different manners. First, some studies have considered performing operating system tasks such as scheduling and memory management in a power-aware manner. Second, additional studies make use of the operating system to assist lower-level management functionalities in their management decisions. In these applications, the operating system is extended with monitoring and control interfaces that track workload characteristics and provide control directives to the underlying management schemes such as frequency scaling and disk power management. In the first direction, prior studies have considered energy-aware scheduling of workloads with different characteristics to balance power consumption, to reduce power density and to control energy dissipation rate in both single and multiprocessor systems [14, 67, 127, 184]. Other workload-adaptive system research has discussed power-aware memory management [135, 186] and page allocation [110]. Some recent studies have also presented methods for power-efficient distribution of parallel, multithreaded applications into multiple homogeneous or heterogeneous processing components [5, 37]. In the second direction, previous studies have discussed system-level adaptations for disk power management [60], controlling network interfaces and managing other input/output devices [174]. In addition, there has been a growing body of work in system-level management for dynamic voltage and frequency scaling [33, 49, 176]. More recently, there has also been interest in machine learning techniques for power management across multiple platform components [167], as well as dynamic compilation support for workload-adaptive power management [73, 172, 179].

At the architecture level, existing work has proposed several strategies that track varying workload characteristics to perform architectural adaptations. Tracking methods differ significantly in their approaches. These can be simple occupancy or usage based models

[3, 139], metrics that characterize varying workload performance [8, 26] access frequency monitoring [48, 96], inconsistency checks [47] or more detailed hardware structures that aim to discern varying application phases [41, 90, 153]. In general, architectural management approaches focus on modulating the effective size or speed of different hardware units. Among different architectural components, memory hierarchy is one of the most investigated structures. Different studies have proposed adaptively disabling or reducing supply voltages for different cache ways and unused blocks [48, 96, 140]. Some work has proposed dynamically configurable caches based on varying working set size information and changes in control flow [9, 41, 153]. Architectural management schemes for higher levels of memory hierarchy, including main memory and disks have also been explored [117, 186]. Besides the memory hierarchy, several studies have focused on other architectural adaptations, such as adaptive issue queues [8, 26, 139]. These approaches have considered monitoring changes in application performance (i.e. rate of executed instructions) and changes in the occupancy of queue structures to tune their configurations to the changes in workload characteristics. Other management schemes have also been proposed for adaptive pipeline scaling and dynamic configurations of other architectural components such as reorder buffers and register files [3, 90]. These techniques have also employed some amount of architectural support (for example, the branch behavior buffer and power profiling units) to track dynamically-varying workload demands and to effectively match the dynamic configurations to different application phases.

My thesis in particular discusses workload-adaptive power management techniques that operate at the architecture and system boundary. It leverages architectural execution information to guide system-level adaptations. Most of the existing system adaptations either function reactively by responding to recent execution behavior or rely on prior profiling information. My work, however, describes a predictive and completely on-the-fly adaptation strategy that utilizes runtime phase predictions to manage dynamic adaptations, without effecting the execution or the structure of workloads.

1.4 Thesis Contributions

My thesis makes four main contributions to the existing literature. First, I describe a generic approach to microarchitecture-level power modeling using processor hardware performance monitoring features. I demonstrate a detailed, yet practical runtime power monitoring and estimation approach with simultaneous measurement support for runtime validation feedback. Overall, this framework paves the way for many following runtime power and thermal management studies that can benefit from insight on live processor power dissipation.

Second, I provide two important contributions to the general body of workload characterization and phase analysis research. I demonstrate practical real-system methods for identifying application phases at runtime. These techniques can be readily employed in system-level dynamic power and thermal management studies. Moreover, my work defines phases targeted directly to discern varying power characteristics of workloads, using event-counter-based power estimations at the basis of its similarity analysis.

Third, this thesis presents a complete flow of methods that mitigate the negative impacts of system-induced variability and sampling effects on the detection of repetitive application behavior. My work describes a taxonomy of phase transformations due to variability and sampling effects. I introduce a new, transition-based phase characterization, which is shown to be more resilient for repetitive phase detection under the influence of these transformations. This work provides a quantitative evaluation of phase detection techniques and quantifies their effectiveness in recognizing recurrent execution.

Last, in this thesis I demonstrate a complete real-system framework for runtime phase prediction and its application to workload-adaptive power management. I describe a configurable runtime phase prediction methodology that seamlessly operates on a real mobile system with negligible overheads. I depict the immediate benefits of runtime phase prediction for on-the-fly, phase-driven dynamic power management. Although the examples shown in this thesis use certain phase definitions for specific power management techniques, the

developed approaches represent a general-purpose phase monitoring and prediction framework. My infrastructure can be employed for monitoring and predicting different workload characteristics that can guide a range of dynamic management techniques.

1.5 Thesis Outline

The following chapters of this dissertation present the main accomplishments of my research in more detail. I present this in a progressive manner, starting with the experimentation basics and the power analysis framework, followed by phase analysis basics, phase detection and prediction methods and finally their application to dynamic power management. In particular, Chapter 2 presents the fundamentals of my real-system experimentation framework and develops runtime processor power monitoring and estimation techniques. Chapter 3 discusses different phase analysis strategies and demonstrates their effective application for power-oriented workload phase characterization. Chapter 4 focuses on the interesting challenges of phase detection in real-system experiments and develops an effective phase detection framework, which is resilient to system-induced variations in observed workload characteristics. Chapter 5 introduces an efficient real-system phase prediction method and outlines a complete infrastructure that is driven by runtime phase predictions for workload-adaptive power management. This chapter meshes the different aspects of my research together and demonstrates the concrete benefits of phase-based dynamic power management for power-aware computing systems. Last, Chapter 6 presents the final remarks and discusses avenues of future research.

Chapter 2

Power and Performance Measurement on Real Systems: Methods and Basics

With power dissipation becoming an increasingly vexing problem across many classes of computer systems, measuring power dissipation of real, running systems has become crucial for hardware and software system research and design. Live power measurements are imperative for studies requiring execution times too long for simulation, such as thermal analysis [14, 112, 155]. Furthermore, researchers often need the ability to measure live, running systems and to correlate measured results with overall system hardware and software behavior. Live measurements allow a complete view of operating system effects, I/O, and many other aspects of "real-world" behavior, often omitted from simulation.

To enable such complete view of system behavior, many processors provide hardware performance counters that help give unit-by-unit views of processor events [16, 25, 77, 159, 160]. While these event counters are designed to reflect performance, they can also be used to derive energy estimates for the underlying processor components [18, 85, 93, 95, 116]. Most of the research described in this thesis is based on real-system experimentation and real-measurement-based validations. We develop a runtime performance monitoring framework and devise event-counter-based power estimations. We use real power measurements to validate power estimations as well as to evaluate our characterization and dynamic management techniques in the chapters that follow.

This chapter presents an overview of our performance monitoring and counter-based power estimation framework. This framework lays out the general experimental principals used in the subsequent chapters, while each of the latter studies have unique experimental features that we discuss in the corresponding chapters. The primary contributions of this chapter are threefold. First, it presents a complete real-system experimentation framework for power oriented systems research, including performance monitoring, real power measurement and estimation, and runtime validation. Second, it describes a detailed methodology for gathering live, per-unit power estimates based on hardware performance counters in complicated and aggressively-clock-gated microprocessors. Third, it presents architecture-level power characterizations for several SPEC and other common desktop applications, which are validated with real measurements.

2.1 Experimental Setup Overview

Figure 2.1 shows a high-level overview of the real-system experimentation flow that is used in various studies in this thesis. This figure also summarizes some of the primary functionalities of the different experimental framework components. In general, the overall experimentation framework consists of the experimental computer system, external power measurement components and a monitoring system that performs data collection and additional analyses. The experimental system includes the applications that are tested, software monitoring and control mechanisms that are implemented in the operating system (OS), and hardware structures within the processor that perform performance monitoring and that configure processor operating modes. The paragraphs below elaborate on the main features of these components.

Experimental System Hardware: Most of today's processors include some dedicated hardware performance counters for debugging and measurement. In general, performance counter hardware includes event signals generated by CPU functional units, event detectors detecting these signals and triggering the counters, and hardware counters configured

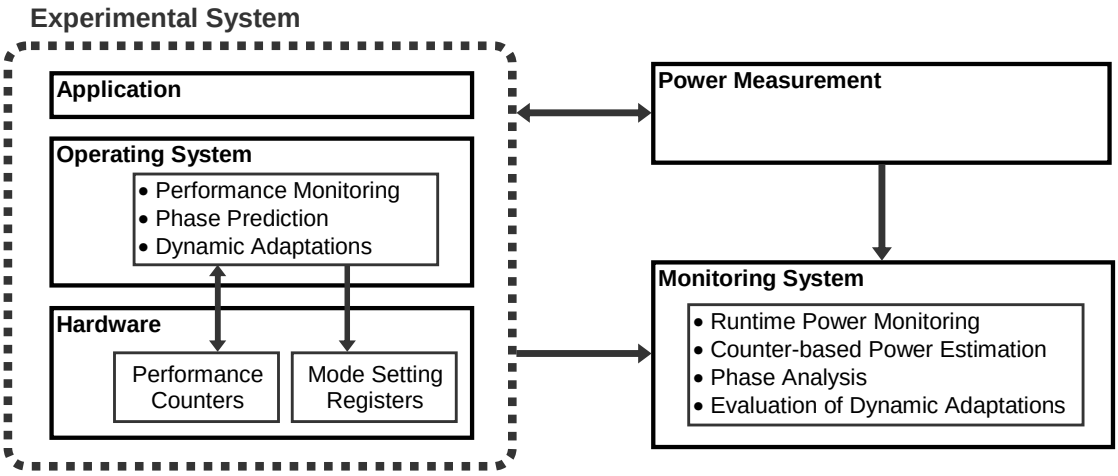


Figure 2.1: High-level view of general real-system experimentation framework.

to increment according to the triggers [79, 81, 159]. We rely on hardware performance counters to track architectural characteristics. In dynamic adaptation studies we also utilize other specialized registers within the processor to tune system execution to the workload demands, such as dynamically adjusting the voltage and frequency settings.

Experimental System OS: The operating system provides the necessary interface between the processor hardware and the monitoring and control mechanisms used in this research. We implement several functionalities inside the operating system as kernel modules or interrupt handlers for multiple purposes in different studies. These include (i) runtime performance monitoring, (ii) dynamic phase predictions and (iii) dynamic management actions.

Power Measurement: An important aspect of real-system experimentation for power oriented research is real power measurements for modeling and evaluation. Processor power dissipation can be measured in various ways. These include using serial sense resistors [93, 148, 168] or current probes [5, 19, 85] over the processor power lines, which are then fed into a digital multimeter, oscilloscope or a data acquisition system [34, 70, 72, 83, 179]. More recently, as the benefits of such power dissipation information are appreciated we also see designs for high-end systems emerging with on-chip power monitoring features

[124, 138], as well as new standards for platform-level power monitoring, such as the Power Supply Management Interface (PSMI) [149].

In this thesis research, real-system power measurements provide runtime processor power dissipation feedback. This information is used to validate power estimations and to evaluate phase characterization and dynamic power management techniques in different studies. We track processor power consumption with power measurements via current probes or data acquisition systems. These power measurements are either fed back to the experimental machine or are sent to a separate monitoring machine.

Monitoring System: The monitoring machine also performs a variety of tasks in different studies, such as: (i) monitoring the runtime power behavior of applications, (ii) estimating runtime power dissipation of processor units based on monitored performance counters, (iii) characterizing power phase behavior of applications and detecting repetitive application execution, and (iv) evaluating the benefits of employed dynamic power management techniques. In some of our studies, the monitoring system also communicates with the experimental machine to track performance behavior at runtime.

Overall, Figure 2.1 depicts the governing experimental flow that is employed in this thesis. The subsequent chapters discuss the specific implementation details of the individual studies and expand upon the above mentioned functionalities.

2.2 Using Performance Counters for Power Estimation

This section discusses a specific aspect of our thesis research related to real-system power monitoring: runtime power estimation using hardware performance counters. While total power measurements for long-running programs are already useful, it is also important to estimate how power subdivides among different hardware units within a processor. For this purpose, this work uses power estimates based on performance counter readings to produce per-unit power breakdowns of total processor power dissipation. From a Pentium 4 die layout, we break the processor into sub-units such as L1 cache, branch prediction

hardware, and others. For each component, we develop a power estimation model based on combinations of events available to Pentium 4 hardware counters as well as heuristics that translate event counts into approximate access rates for each component. We use real power measurements obtained from a current probe to provide a runtime comparison between the measured and estimated total power measures.

The machine used in these experiments is a 1.4GHz Pentium 4 (P4) processor, 0.18μ Willamette core. The CPU operating voltage is 1.7V and published typical and maximum power values are 52W and 71W, respectively [80]. The NetBurst microarchitecture of P4 is based on a 20-stage misprediction pipeline with a trace cache to remove instruction decoding from the main pipeline. In addition to a front-end branch prediction unit (BPU), a second smaller BPU is used to predict branches for Uops (micro-ops) within traces. It has two double-pumped ALUs for simple integer operations. The L1 cache is accessed in 2 cycles for integer loads, while the L2 cache is accessed in 7 cycles [69]. The processor implements extremely aggressive power management, clock gating, and thermal monitoring. Almost every processor unit is involved in power reduction and almost every functional block contains clock gating logic, summing up to 350 unique clock gating conditions. This aggressive clock gating provides up to approximately 20W power savings on typical applications and produces high amounts of power variation within and across workloads [15].

Prior work has developed counter-based or profile-based estimates for much simpler processors [13, 93, 113, 168]. In our approach, we estimate physical component powers using counter-based measures, and also generate accurate total power estimates. This modeling technique is distinct from prior work in the following ways. We estimate power for a much more complicated modern processor, with extremely aggressive clock gating and high power variability. Second, we consider strictly physical components from the die layout. Finally, we estimate power for all levels of processor utilization for arbitrarily long periods of time, rather than restricting our technique only to power variations at high processor utilization. The latter two are particularly important for thermal studies as ther-

mal variations show significant spatial distributions among physical processor components and are observed in long timescales on the order of seconds [155]. The remainder of this section describes our power estimation methodology and discusses the particulars of the underlying experimental framework.

2.2.1 Defining Components for Power Breakdowns

There are two primary factors that drive the way we determine the processor components for which the power breakdowns are generated. First, we desire microarchitecture-level granularity in the power decompositions. Second, we choose components that provide a direct mapping to the physical layout. Both of these decisions also help enable future studies for architecture-level processor thermal modeling and hotspot analysis [112].

Based on an annotated P4 die photo we define 22 physical components: Bus control, L1 cache, L2 cache, L1 BPU, L2 BPU, instruction TLB & fetch, memory order buffer, memory control, data TLB, integer execution, floating point execution, integer register file, floating point register file, instruction decoder, trace cache, microcode ROM, allocation, rename, instruction queue1, instruction queue2, schedule, and retirement logic.

2.2.2 Selecting Performance Monitoring Events for Power Estimation

For each of the 22 components, we need a performance counter event or a combination of events that can approximate the access count of that component. The finalized set of heuristics that define these access counts involve 24 event metrics composed in various ways for the 22 defined processor components [86]. As an example for the access rate heuristics, the access rate for the trace cache component can be approximated by configuring the “Uop queue writes” event to count all speculative Uops written to the small in-order Uop queue in front of the out-of-order engine. These come from either trace cache build mode, trace cache deliver mode or microcode ROM.

As another example, the access rates for the bus control logic component are obtained by counting the allocations into the I/O queue (via IOQ Allocations) and by tracking the

activity on the front side bus (via FSB Data Activity). IOQ Allocations count all bus transactions (all reads, writes and prefetches) that are allocated in the IO Queue (between the L2 cache and bus sequence queue). FSB Data Activity is configured to track the events that occur on the front side bus when processor or other agents drive, read or reserve the bus. The bus ratio (3.5 in our implementation) is the ratio of processor clock (1400MHz) to bus clock (400MHz), and converts the counts in reference to processor clock cycles. Equation 2.1 shows the resulting access rate relation for the memory controller unit.

$$Access\ Rate(Bus\ Control) = \frac{IOQ\ Allocation}{\Delta Cycles} + \frac{Bus\ Ratio \cdot FSB\ Data\ Activity}{\Delta Cycles} \quad (2.1)$$

To account for all component accesses, we use 15 counters with 4 rotations. The P4 events and counter assignments minimize the counter switches required to measure all the metrics needed. At least four rotations are unavoidable. This is because floating point metrics involve 8 different events, of which only two at a time can be counted due to the limitations of P4 counter configurations.

2.2.3 Counter-based Component Power Estimation

We use the component access rates—either given directly by a performance counter or approximated indirectly by one or more performance counters—to weight component power numbers. In particular, we use the access rates as weighting factors to multiply against each component's maximum power value. This product is further multiplied with a scaling factor that is based on microarchitectural and structural properties. In general, all the component power estimations are based on Equation 2.2, where maximum power and conditional clock power are estimated empirically during implementation. The C_i in the equation represent the 22 hardware components.

$$Power(C_i) = AccessRate(C_i) \cdot ArchitecturalScaling(C_i) \cdot MaxPower(C_i) + NonGatedClockPower(C_i) \quad (2.2)$$

As an example of our overall technique, consider the trace cache component. It delivers three Uops/cycle when a trace is executed and builds one Uop/cycle when instructions are decoded into a trace. Therefore, the access rate approximation in deliver mode is scaled by 1/3, while the access rate from instruction decoder is scaled with 1. These rates are then used as the weighting factors for the estimated maximum trace cache power.

Equation 2.3 constructs the total power as the sum of 22 component powers calculated as above, along with a fixed idle power of $8W$ obtained from actual power measurements. Hence, this fixed $8W$ base includes some portion of globally non-gated clock power, whereas the conditionally-gated portion of clock power is distributed into component power estimations.

$$Total\ Power = \sum_{i=1}^{22} Power(C_i) + Idle\ Power \tag{2.3}$$

For initial estimates of each component's "maxpower" value, $MaxPower(C_i)$ in Equation 2.2, we used physical areas on the die. In many cases, these areas serve as good proportional estimates. To further tune these maximum power estimates, we developed a small set of training benchmarks that exercise the CPU in particular ways. By measuring total power with a multimeter, we could compare true total power over time to the total power estimated by summing our component estimates. After several experiments with the training benchmarks, we arrived at a final set of maxpower and non-gated clock power values for each of the components. These are hard-coded as the P4 specific weighting factors in the final implementation of our power estimation setup.

While this section describes our overall event-counter-based power estimation strategy for a specific platform, there are certain design aspects that could help further improve the accuracy of power estimations with future performance monitoring hardware implementations [82]. First and foremost, as power and thermal estimations correspond directly to physical units located on die, counters that individually track accesses to each unit separately are extremely desirable. This set of counters should provide high parallelism in

concurrent counting to minimize the need for counter rotations. Such information, together with the documentation of maximum utilizations and maximum power per unit allows for easier and more accurate tracking of component activity. Second, additional structures to track bitline activity and one/zero population counts are imperative for good power estimates. Third, depending on circuit implementation, certain queues at the in-order processor front-end and out-of-order engine schedulers dissipate power in proportion to their occupancy. Specifically for these units, metrics that gauge the occupancy ratio lead to better power approximations. Furthermore, in lower-power embedded processors power consumption of support logic outside the core can significantly contribute to processor power consumption. Adding counter support for external core components (off-chip memory accesses, DMA unit activity, etc.) can increase the fidelity of counter-based power estimation, providing greater opportunities for power behavior monitoring and control.

2.3 Implementation Details for Counter-based Power Estimation

The complete event-counter-based power estimation framework is conceptually based on the generic experimental flow described in Section 2.1. Runtime performance monitoring lies at the center of this power estimation methodology. Direct validation of the power estimations is performed with runtime real power measurements. The overall experimental setup combines these two components into the final infrastructure that performs runtime power estimation with real measurement feedback. This section first discusses the implementation details for performance monitoring and power measurement. Afterwards, it presents the complete power estimation infrastructure.

2.3.1 Hardware Performance Monitoring

To access the hardware performance counters, there are a number of pre-written counter libraries available [16, 78, 158, 25]. For efficiency and ease-of-use, we have written our own Linux loadable kernel modules (LKMs) to access the counters. Our LKM-based implementation offers a mechanism with sufficient flexibility and portability, while incurring

negligible power and performance overhead so that we can continuously collect counter information at runtime and generate runtime power statistics. In order to use the performance counters, we implement two LKMs. The first LKM, *CPUinfo*, is simply used to read information about the processor chip installed in the system being measured. This helps the tool identify architecture specifications and discern the availability of performance monitoring features. The second LKM, *PerformanceReader*, implements six system calls to specify the events to be monitored, and to read and manipulate counters. The system calls are:

(i) select events: Updates the event selection control register (ESCR) and counter configuration control register (CCCR) fields as specified by the user to define the events, masks, and counting schemes.

(ii) reset event counter: Resets specified counters.

(iii) start event counter: Enables specified counter's control register to begin counting

(iv) stop event counter: Disables specified counter's control register to end counting

(v) get event counts: Copies the current counter values and time stamp to user space

(vi) set replay MSRs: updates special model specific registers (MSRs) required for "replay tagging" [79].

With this simple and lightweight interface, we can completely control and update counters easily from within any application.

2.3.2 Real Power Measurements

In this work, real power measurements provide the time-varying processor power dissipation information to validate the counter-based power estimations. We use a current probe with a digital multimeter to track current flow in the Pentium 4 desktop platform.

Figure 2.2 shows the details of our power measurement setup. Here CPU power is measured with a clamp ammeter (current probe). The main power lines for the CPU operate at 12V, and then are fed to the voltage regulator module, which converts this voltage to the

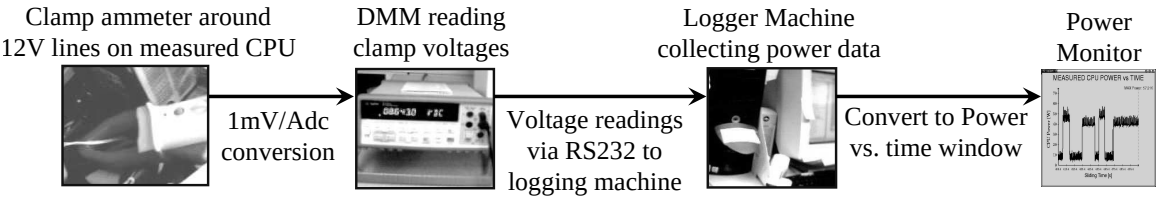


Figure 2.2: Processor power measurement setup.

actual processor operating voltage and provides tight control on voltage variations [185]. Therefore, we use our current probe to measure the total current through the 12V lines. We use a Fluke i410 current probe connected to an Agilent 34401 digital multimeter (DMM). The current probe converts current readings to voltages with a $1mV/A$ conversion rate. The DMM sends the voltage readings to a second logging machine via the serial port. The logger machine converts these values into processor power dissipation with the power relation: $P = V \cdot I = 12 \cdot (VoltageSample[V]) \cdot 1000$. It displays the measured runtime power in our developed *power monitor* with a sliding time window, while also logging time vs. voltage information.

The DMM samples 1000 current readings per second with $4\frac{1}{2}$ digit resolution, which corresponds to 0.12W power resolution. However, it can transfer around 55 samples per second over RS232, so we collect the data in the logger machine at 20ms intervals, while finer granularity sampling is possible with a General Purpose Interface Bus (GPIB). The logging machine then computes a moving average within a longer second sampling period that is used to update the on-screen power monitor and the data log. These coarser granularity samples are used in validating the counter-based power estimations.

2.3.3 Overall Implementation

In the final implementation, the performance reader provides the system with the required counter information. The monitoring machine collects all the counter and measurement information to generate the runtime component power estimations. We verify power estimates against total power measurements by measuring actual power and by feeding this information simultaneously to the monitoring machine. Figure 2.3 depicts the overall ex-

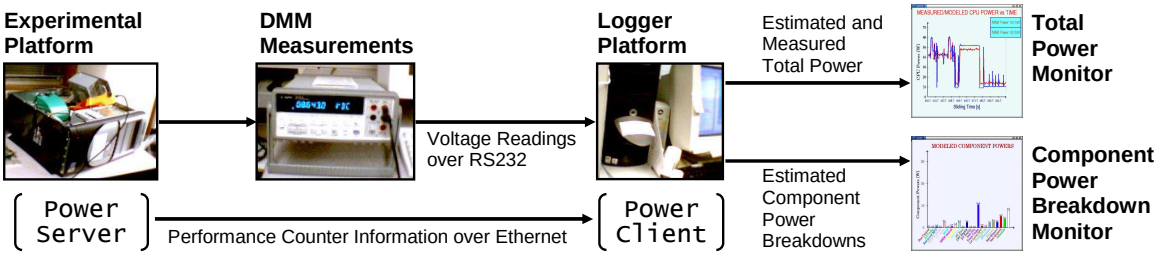


Figure 2.3: Overall runtime power measurement and performance-counter-based power estimation setup.

perimental setup for this final implementation.

Measured processor current is sent by the DMM to the logger machine via RS232 and the logger machine converts the current information to power. On the experimental machine, the *PowerServer* collects counter information every 100 ms, for the P4 events chosen to approximate component access rates. Every 400ms, the PowerServer sends collected information to the logging machine over the Ethernet. While this perturbs system behavior slightly, it is done infrequently to minimize the disturbance. On the logger machine, the *PowerClient* collects measured ammeter data from the serial port, and raw counter information from Ethernet. Combining the two, it applies the access rate and power model heuristics, and generates component power estimates for the defined components. After synchronizing the modeled and measured power over a 100 second time window, the PowerClient generates a runtime *component power breakdown monitor* as well as runtime *total power monitor* for both measured and counter estimated power.

2.4 Power Estimation Results

This section provides the results for our power estimation framework for some microbenchmarks with well-defined characteristics, for the full runtimes of SPEC benchmarks, and for some common desktop applications. The benchmarks are compiled using gcc-2.96 and with compiler flags of "-O3 -fomit-frame-pointer". For SPEC workloads, we use the reference inputs with a single iteration of run. In order to demonstrate our ability to model power closely even at low CPU utilizations, we also experimented with practical desktop tools:

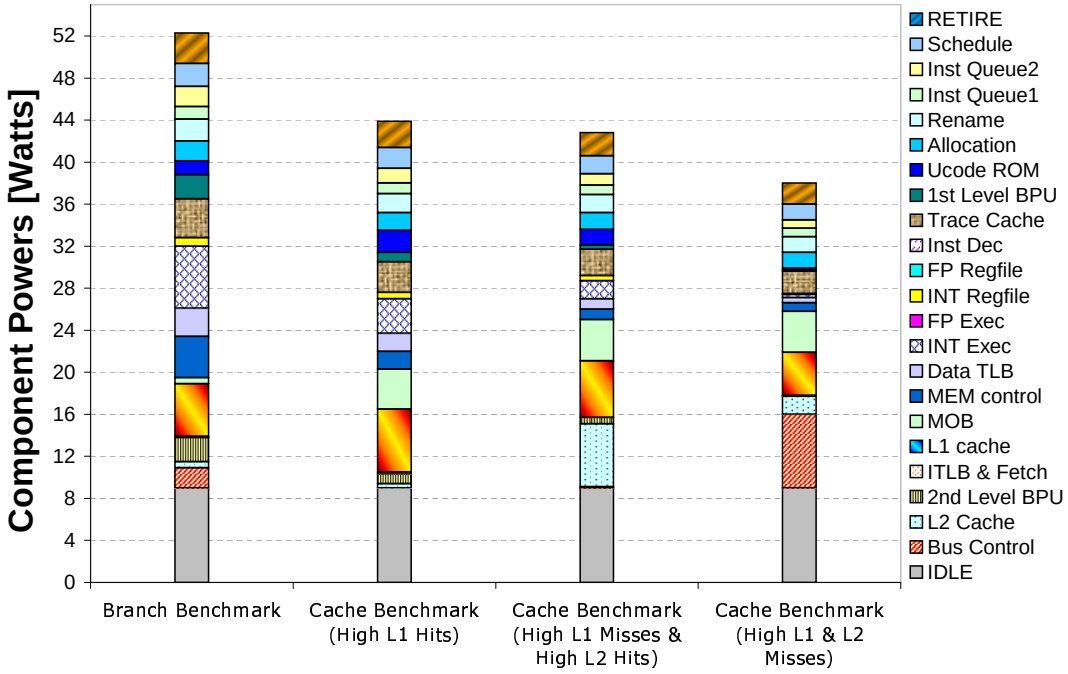


Figure 2.4: Power breakdowns for branch and cache benchmarks.

AbiWord for text editing, *Mozilla* for web browsing and *Gnumeric* for numerical analysis. All these benchmarks share the common property of producing low CPU utilization with only intermittent power bursts.

2.4.1 Microbenchmark Results

Figure 2.4 shows component power breakdowns for two microbenchmarks. The `branch` benchmark creates different branch misprediction rates and different ratios of taken branches. This is a very small program that is expected to reside mostly in the trace cache and that is mostly L1 bound. This microbenchmark is a high Uops per cycle (UPC), high-power integer program. The `cache` benchmark creates variable L1 and L2 cache hit rates by performing a linked list traversal in a pseudorandom sequence.

The leftmost bar of Figure 2.4 shows the estimated power breakdowns for our branch exercise microbenchmark. The breakdowns show high issue, execution and branch prediction logic power. In contrast, because the application dataset mainly fits in the L1 cache, the L2 cache and bus for main memory dissipate lower power.

The second bar of Figure 2.4 shows breakdowns for cache exercise microbenchmark

with an almost perfect L1 hit rate. Once again, the component breakdowns track our intuition well. The breakdowns show high L1 power consumption and relatively high issue and execution power as we do not stall due to L1 miss and memory ordering/replay issues. Both L2 and bus power are relatively low.

In the third bar of Figure 2.4, we configure the cache microbenchmark to generate high L1 misses, while hitting almost perfectly in L2. The power distribution of L2 cache is seen to increase significantly, while execution and issue cores slow down due to replay stalls. Moreover memory order buffer power shows a slight increase due to increasing memory load and store port replay issues.

Finally, in the rightmost bar of Figure 2.4 the workload also generates high L2 misses and therefore bus power climbs up, while the execution core slows down even further due to higher main memory penalties. Although total L2 accesses actually increase, due to significantly longer program duration, access rates related to L2 drop and aggregate L2 power decreases.

Overall, this sequence of microbenchmarks, while simple, builds confidence that the counter-based power estimates show meaningful insights to architecture-level power dissipation and do not violate intuition in their estimates. The sections that follow present more large-scale, long-running experiments on SPEC and desktop applications.

2.4.2 SPEC Results

Figure 2.5 first shows our power estimation results for the SPEC `gcc` benchmark to demonstrate the capability of our power estimation framework. In this figure, we show the total estimated and measured power behavior of `gcc` for its complete execution time over its five data sets. `Gcc` is one of the most highly varying benchmarks in the SPEC CPU2000 suite, as observed with the measured power timeline. Our power estimations closely track actual `gcc` power behavior, at all regions of execution. This shows our event-counter-based power estimations provide a very good proxy to application power behavior, regardless of the range of power consumption.

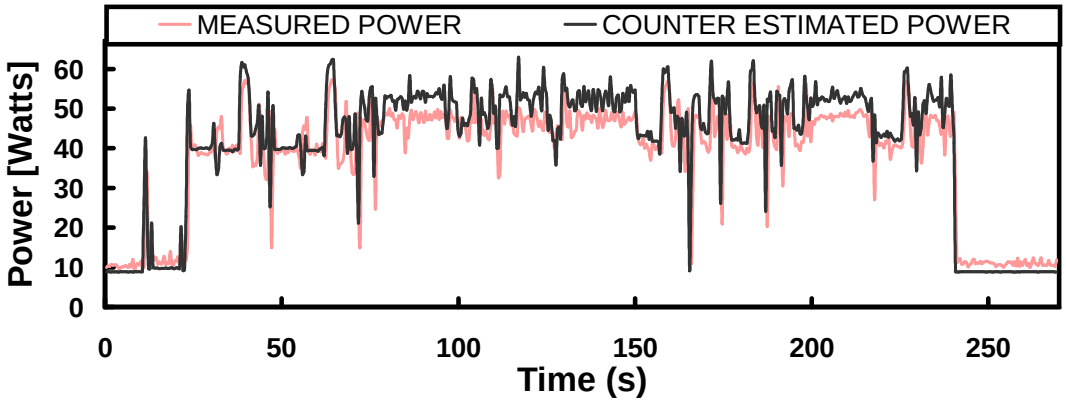


Figure 2.5: Measured and estimated power behavior for the `gcc` benchmark with a 400ms sample granularity.

In addition to `gcc`, Figures 2.6–2.9 also show the power estimations and detailed component breakdowns for the SPECint application `vpr`, and `equake` from SPECfp. For reference inputs, the `vpr` benchmark actually consists of two separate program runs. The first run uses architecture and netlist descriptions to generate a placement file, while the second run uses the newly-generated placement file to generate a routing descriptor file [166]. Although the total average power for the two runs is quite similar, Figure 2.6 shows a noticeable phase change at around 300s when the second run begins. Figure 2.7 demonstrates even more clearly how distinct the power behavior in the second phase is. Although the first run, the placement algorithm, dissipates very stable power, the second phase's routing algorithm has a much more variable and periodic power behavior. As discussed in prior work [101], the initial placement phase produces higher miss rates than the routing part. This is because routing benefits from the fact that placement brings much of the dataset into memory. The per-component power breakdowns corroborate this with the increased L2 power in second phase.

As an example of floating point benchmarks, Figures 2.8 and 2.9 show the `equake` benchmark. `Equake` models ground motion by numerical wave propagation equation solutions [10]. The algorithm consists of mesh generation and partitioning for the initialization, and mesh computation phases. In Figure 2.8, we can already clearly identify the initialization phase and computation phase. Figure 2.9 demonstrates the high microcode ROM

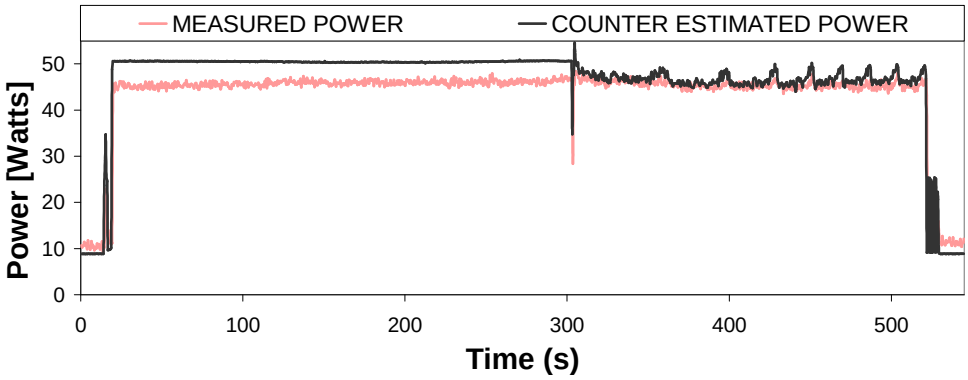


Figure 2.6: Total measured and modeled runtime power for `vpr`.

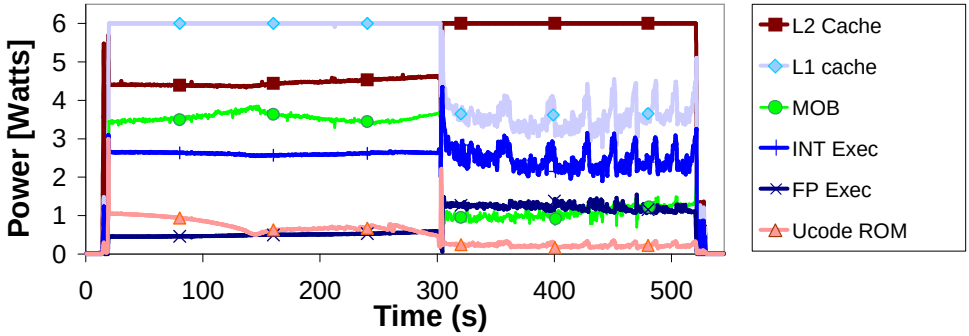


Figure 2.7: Estimated power breakdowns for `vpr`.

power as the initialization phase uses complex IA32 instructions extensively. The mesh computation phase, then exhibits the floating point intensive computations.

In addition to `gcc`, `vpr` and `equake`, we have generated similar power traces for several other SPEC2000 benchmarks. Figures 2.10 (a) and (b), present statistical measures that confirm the accuracy of our modeling framework, for the larger set of the SPEC2000 benchmarks.

Figure 2.10 shows the average power computed from real power measurements and counter estimated total power, for both the whole runtime of the benchmarks and for the actual execution phases, excluding idle periods. Hence, the idle-inclusive measures cannot be considered as standard results, as the idle periods vary in each experiment. They are of value, however, for comparing counter-based totals to measured totals, because one of our aims is to be able to characterize low utilization power with reasonable accuracy as well. For the estimated average power, the average difference between estimated and

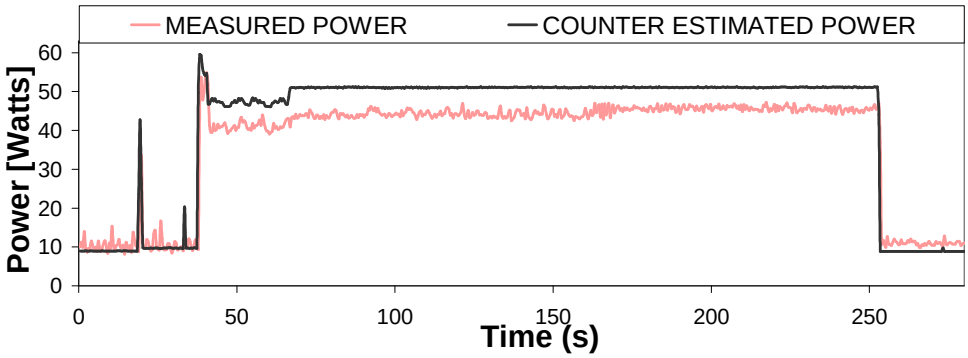


Figure 2.8: Total measured and modeled runtime power for equake.

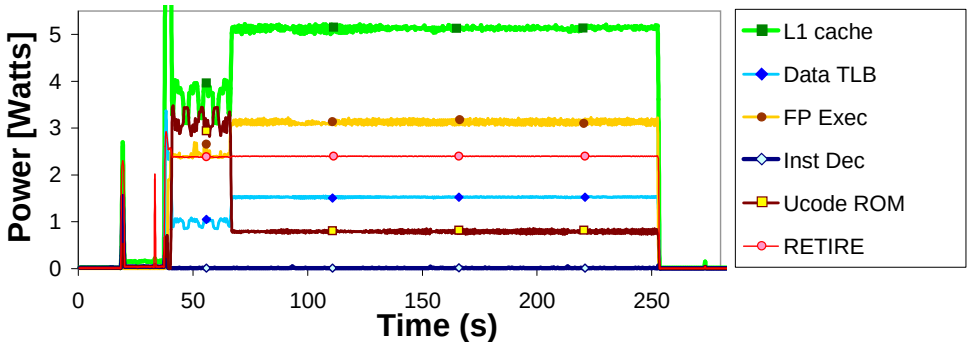


Figure 2.9: Estimated power breakdowns for equake.

measured power is around 3 Watts, with the worst case being equake (Figure 2.8), with a 5.8W difference. For the standard deviation, the average difference between estimated and measured power is around 2 Watts, with the worst case being vortex, with a standard deviation difference of 3.5W.

2.4.3 Desktop Applications

In addition to SPEC applications, we investigated three Linux desktop applications as well. These help demonstrate our power model's ability to estimate power behavior of practical desktop programs. Because of their interactive nature, these applications typically present periods of low power punctuated by intermittent bursts of higher power. The three applications, shown in Figure 2.11, are AbiWord for text editing, Mozilla for web browsing and Gnumeric for numerical analysis.

In the web browsing experiment in Figure 2.11(a), the power traces represent opening the browser, connecting to a web page, downloading a streaming video and closing the

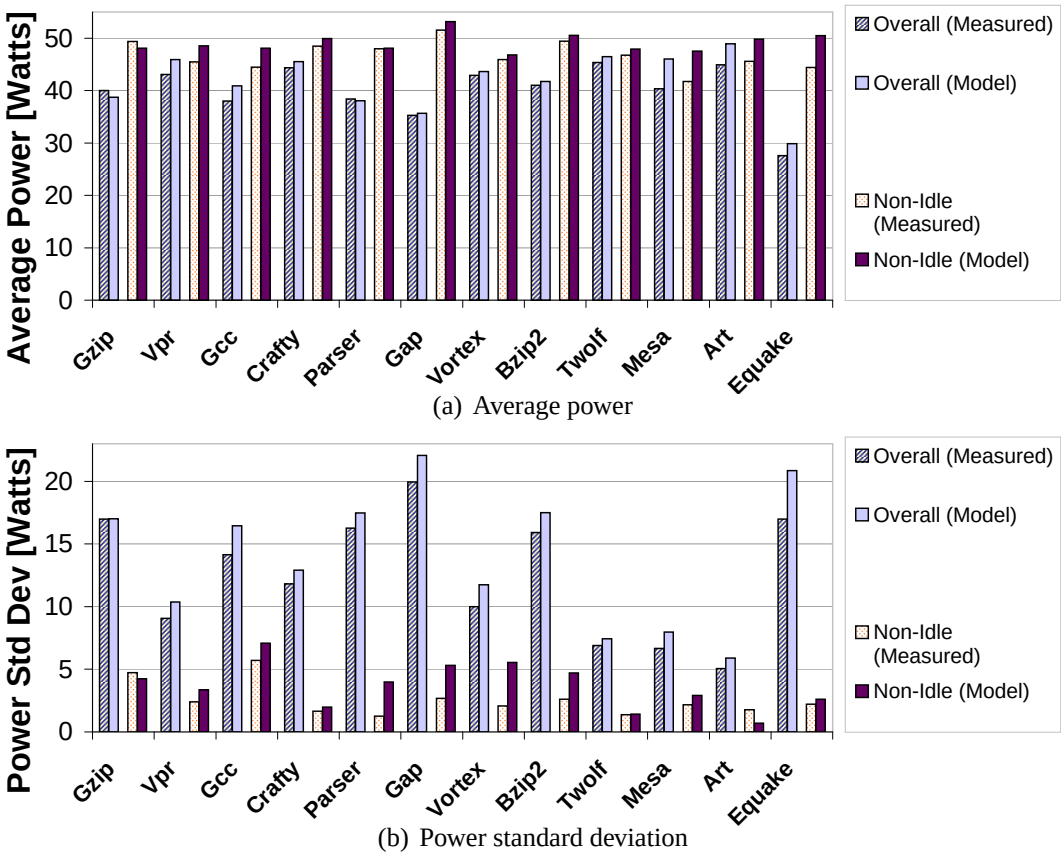


Figure 2.10: Average (top) and standard deviation (bottom) of measured and counter estimated power for the SPEC2000 benchmarks. For each benchmark, the first set of power values represents averaging and standard deviation over the whole runtime of the program. The second set represents averaging and standard deviation only over non-idle periods.

browser. In the text editing experiment in Figure 2.11(b), the power traces represent opening the editor, writing a short text, saving the file and closing the editor. In the Gnumeric example in Figure 2.11(c), the power traces represent opening the program, importing a large data set, performing statistics on the data, saving the file and closing the program. The power traces reveal the bursty nature of the total power timeline for these benchmarks. Overall, the long idle periods mean that the benchmarks have low average power dissipation. The power traces for the desktop applications also reveal that our counter-based power model follows even very low power trends with reasonable accuracy. Together with the SPEC results, this demonstrates that our counter-based power estimates can perform reasonably accurate estimations independent of the range of power variations produced by

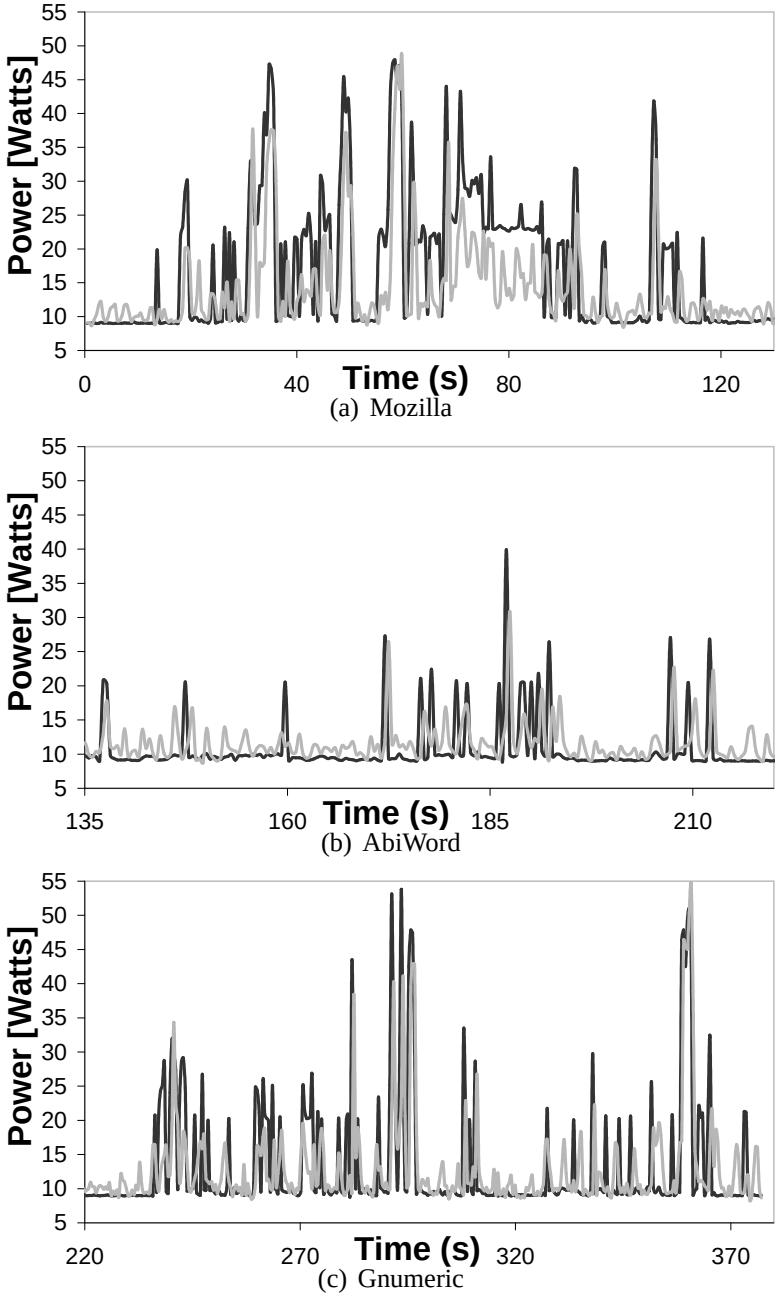


Figure 2.11: Total measured (light) and counter estimated (dark) runtime power for 3 desktop applications.

different applications, without any realistic bounds on the observed timescale.

2.5 Related Work

While there has been significant work on processor power estimations, much has been based only on simulations. Our approach, in contrast, uses live performance counter mea-

surements as the foundation for an estimation technique.

One category of prior work involves live measurements of total power. While these are numerous, we touch on a few key examples here. In early work, Tiwari et al. developed software power models for an Intel 486DX2 processor and DRAM and verified total power with real measurements [168]. This work developed instruction energy cost tables and demonstrated inter-instruction effects. Russell and Jacome presented a software power model for i960 embedded processors, validated using current measurements [142]. Flinn et al. developed the PowerScope tool, which maps consumed energy to program structure at procedural level [51]. More recently, Lee et al. used energy measurements based on charge transfer to derive instruction energy consumption models for a RISC ARM7TDMI processor [113]. This study used linear regressions to fit the model equations to measured energy at each clock cycle. These techniques are aimed at very simple processors with almost no clock gating, and therefore need to track and model only minimal temporal power variations.

As a first example of Pentium 4 power measurement studies, Seng and Tullsen investigated the effect of compiler optimizations on average program power [148]. They use real measurements to track the processor power. However, they do not present total and component-level power estimations.

Another category of prior work is on performance counters and power metrics. For example, Bellosa uses performance counters to identify correlations between processor events for an Intel Pentium II processor [13]. This counter-based energy accounting scheme is used as a feedback mechanism for OS directed power management such as thread time extension and clock throttling. Likewise, the Castle tool, developed by Joseph and Martonosi [93], uses performance counters to model component powers for a Pentium Pro processor. It provides comparisons between estimated and measured total processor power. Our work makes significant extensions in both infrastructure and approach in order to apply counter-based techniques to a processor as complex as the P4. Kadayif et al. use performance

counter information to estimate memory system energy consumption for an UltraSPARC processor [95]. They collect memory related event statistics which are applied to an analytical memory energy model. Last, Haid et al. [62], propose a coprocessor for runtime energy estimation for system-on-a-chip designs. Essentially, the goal of this work is to design a set of event counters specifically for power measurement.

2.6 Summary

This chapter presented an overview of our real-system power and performance monitoring methods. It has described a complete view of our real-system experimentation framework for our power-oriented research. It has discussed the general experimentation principals of runtime performance monitoring, power measurement and estimation. As such, this chapter provides the high-level view of our general research and evaluation methods, applied throughout this thesis.

Following from the overview of the experimentation basics, this chapter has also presented our runtime power estimation methodology based on hardware performance counters for a modern processor with highly variable power behavior. This framework performs power estimations for both the overall processor power consumption and for the individual architectural units within the processor. The resulting power estimations are validated against real power measurements at runtime. The experimental results showed that our power estimations track even very fine trends in program power behavior closely, and can accurately estimate processor power consumption at all levels of CPU utilization.

There are several key contributions of this chapter. First, it lays out a general experimentation approach carried out in the following chapters of this thesis. The measurement and estimation framework offers an alternative to purely simulation-oriented power research. Our runtime power estimations demonstrate a promising and practical methodology for tracking architectural power behavior in real-system power and thermal management techniques. The component power breakdowns offer architecture-level detail to runtime

workload power characteristics, which is useful for both characterization and adaptation purposes. Starting with the next chapter, we build upon this monitoring and estimation framework for several research goals. We use the generated component power estimates to characterize power phase behavior of applications and to identify repetitive application execution. We use real-system power and performance monitoring to accurately predict future application behavior and to guide workload-adaptive dynamic power management.

Chapter 3

Power Oriented Phase Analysis

Most workloads exhibit considerable variations in execution behavior during their lifetimes. These different execution characteristics that are observed during workload execution are commonly referred to as *phases* of a workload. Due to large loops at program scale, and procedure-based execution nature, these phases also generally show certain repetitive patterns. While this workload phase behavior has long been observed [39], in recent years application phase behavior has seen growing interest with two main goals. Some seek to identify program phases in order to select representative points within a run to study or simulate [6, 72, 109, 137, 151, 152]. Others seek to recognize phase shifts on-the-fly in order to perform optimizations such as dynamic adaptations in cache organization, voltage/frequency scaling, thermal management, or even dynamic compiler optimizations of hotcode regions [11, 14, 41, 76, 90, 155].

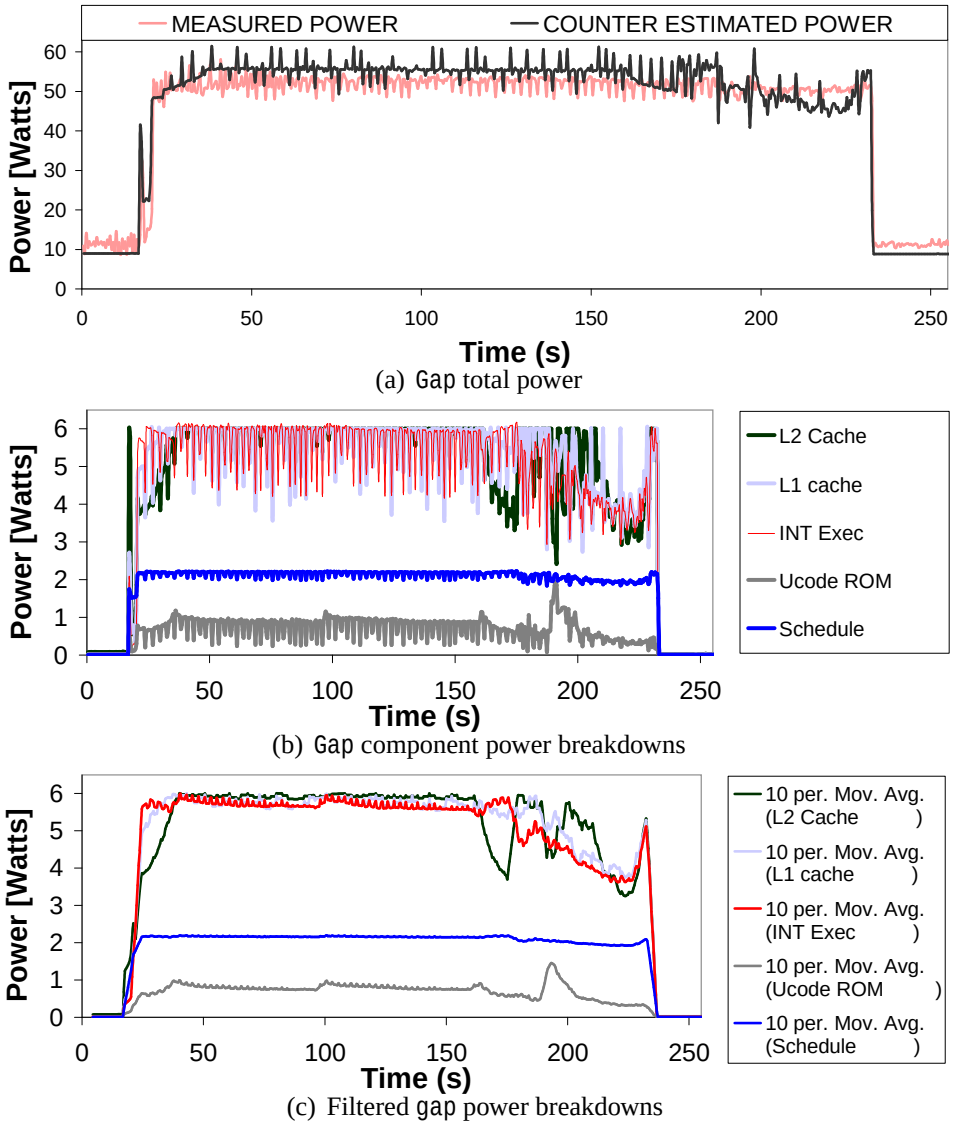
While most of the prior phase analysis studies focus on performance characteristics of applications, runtime power behavior also exhibits significant phase behavior. Moreover, this power phase behavior gets emphasized with emerging processor generations due to increasingly aggressive power management techniques [15, 185]. Different programs with similar average power can show significantly different power variations. Likewise, a single program with stable total power can have distinctively different power behavior—in terms of the decomposition of power to the architectural units—in different execution phases.

This chapter demonstrates a phase analysis method which relates directly to power. Our analyses use estimated power dissipations of processor components—such as cache and integer ALU—to identify the phases a program goes through during its execution. We consider these component power estimates, or *power vectors*, as characteristic features of application power behavior. After describing our phase characterization methodology, this work also evaluates how these features perform in comparison to features used in prior studies for power phase characterization [41, 90, 152]. The most important aspect of this work is that it uses power signatures of programs and therefore, presents a direct and accessible way to analyze power phase behavior. The power vectors used in our analyses are acquired at runtime. Therefore, they are directly applicable to runtime, phase-driven dynamic adaptation techniques.

The power phase analysis described in this chapter offers three primary contributions to the existing research. First, this work demonstrates the advantage of using event-counter-based power vectors for power phase characterization. Second, representative power vectors, generated as part of similarity analysis, can be used as program power signatures in power oriented studies. Third, as this analysis is based on a real system, it can directly be utilized in power aware research for runtime phase identification. With the ability to identify recurring phases over large scales of execution, our technique can also be used for system-level dynamic management [17, 30, 63, 64, 74, 130, 176, 184].

3.1 Characterizing Workload Power Behavior with Power Vectors

Applications exhibit phases at various time granularities and with different characteristics. For example, benchmarks can exhibit different phases with different datasets even though the observed total power may remain similar. On the other hand, within a single dataset a benchmark may go through different phases such as initialization, computation and reporting [6, 85, 88, 108, 151, 169]. Figures 3.1 and 3.2 show two benchmarks, SPEC2000 `gap` and `gzip`, where `gap` shows distinct phases for a single dataset and `gzip` shows periodic



(a) `Gap` total power

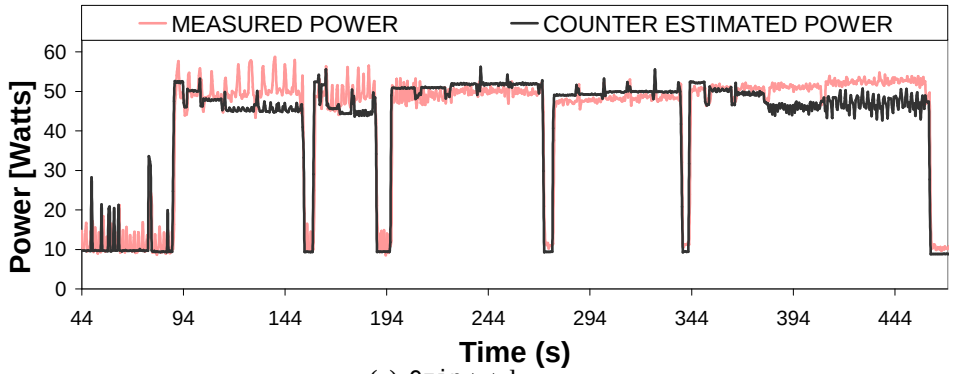
(b) `Gap` component power breakdowns

(c) Filtered `gap` power breakdowns

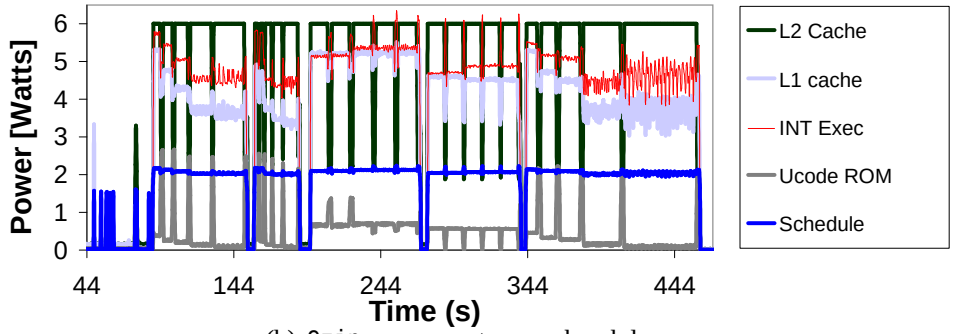
Figure 3.1: Total and component-wise power traces for `gap`.

phases within a dataset as well as recurring phases across its 5 datasets. The sampling period used in these measurements is 400ms. The figures also include plots for power breakdown traces filtered with a 10 point moving average so that we filter down higher frequency phase components and look at distinct phases at the larger whole execution scale.

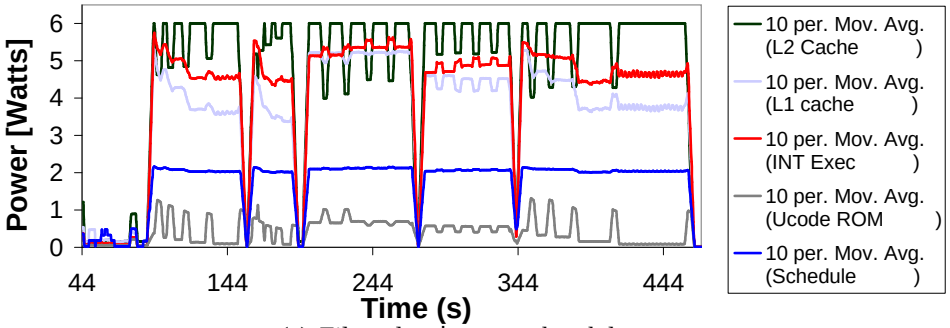
The power traces shown in Figures 3.1 and 3.2 demonstrate two important observations. First, program power behavior exhibits phase behavior, similar to performance metrics such as IPC and miss rates. Additionally these phases may not be visible in *total* power observa-



(a) `Gzip` total power



(b) `Gzip` component power breakdowns



(c) Filtered `gzip` power breakdowns

Figure 3.2: Total and component-wise power traces for `gzip`.

tions, but can be hidden in the variations of power vectors. Second, the runtime technique observes large scale phase behavior in the order of tens of seconds. For most workloads, executing the first few billions of instructions, which correspond to a few seconds of actual execution, can produce a misleading view of program power behavior [35]. Thus, these two observations set the fundamental principals of our power phase analysis research: to focus on large-scale power behavior of whole programs and to identify regions that can accurately represent program power behavior.

Our power phase analysis considers power vectors as points in the positive quadrant of

the power space spanned by the 22 dimensions of these vectors. As each power vector corresponds to a specific execution time sample in the program trace, we evaluate the power behavior similarity of execution regions by measuring the spatial closeness of the points specified by the corresponding power vectors. We use the Manhattan—L1—distance between two vectors as our measure of closeness; it is defined as the absolute difference of vector elements summed over all vector components.

We record the Manhattan distances for all vector pairs in an upper diagonal *similarity matrix* in execution order. Matrix entry (r,c) shows the Manhattan distance between the power vectors corresponding to r^{th} and c^{th} execution time samples. Only the upper diagonal needs to be constructed, as distance from the r^{th} vector to the c^{th} is identical to distance from the c^{th} vector to the r^{th}. The matrix entries are nonnegative real numbers. A "0" at entry (r,c) represents a perfect similarity between execution samples r and c, while higher values represent higher dissimilarity. The execution time flow is along the matrix diagonal. For an execution point r, entries in the upper column $(r_i < r, r)$ represent its similarity with respect to previous samples, while the entries in the right row $(r, c_i > r)$ represent similarity with respect to samples in the forward path.

We visually demonstrate the power similarity matrices in terms of *matrix plots* that are aligned with the execution timeline along the diagonal, where the top left corner represents the start of the timeline and the lower right corner represents the end of the timeline. The matrix entries are presented as greyscale pixels, where the shading is scaled from white, for maximum dissimilarity, to black, for perfect similarity. Figure 3.3 shows a simplified example of our similarity analysis. In this figure we consider four time samples, each as five dimensional vectors. The diagonal-symmetric, 4 by 4 similarity matrix stores the Manhattan distances between vector pairs, which is then represented in the matrix plot.

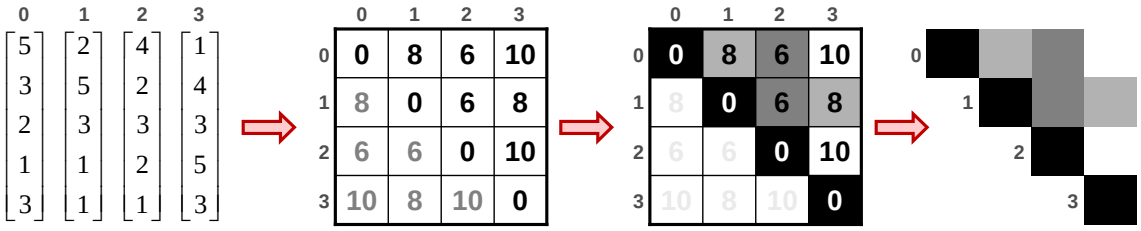


Figure 3.3: Similarity analysis example with four time samples represented as five dimensional vectors.

3.2 Similarity Metrics

While the initial similarity measure provides satisfactory results for vectors with similar norms, it inherently considers vectors with small magnitudes similar regardless of the distribution of power. The initial absolute vectors distinguish regions with high power well. However, they cannot discern low-power regions with different power distributions as their absolute similarity distance remains relatively small. On the other hand, considering normalized vectors helps distinguish small-magnitude vectors with different compositions easily as normalized vectors focus primarily on how power is distributed among vector components. However, normalized vectors cannot differentiate between vectors with different norms and similar component ratios. In refining our similarity analysis, we use a more restrictive approach in order to distinguish cases where vector magnitudes or component ratios are different, even for vectors of smaller magnitudes. Therefore, we use normalized vector distances in conjunction with the original absolute vectors.

We first construct the similarity matrix based on the absolute power vectors from Manhattan distances of all vector combination pairs. A single matrix entry, $AM(r,c)$ for this absolute metric, is computed as shown in Equation 3.1, where $PV_{r,c}$ represent the sample power vectors and $i \in \{1,2,\ldots,22\}$ correspond to vector component indices.

$$AM(r,c) = \sum_{i=1}^{22} |PV_r(i) - PV_c(i)| \tag{3.1}$$

In a similar fashion, we construct the similarity matrix based on only normalized power

vectors from the Manhattan distances of all the combination pairs of normalized power vectors. We compute a single matrix entry $NM(r,c)$ as shown in Equation 3.2, where $NPV_{r,c}$ represent the sample normalized power vectors.

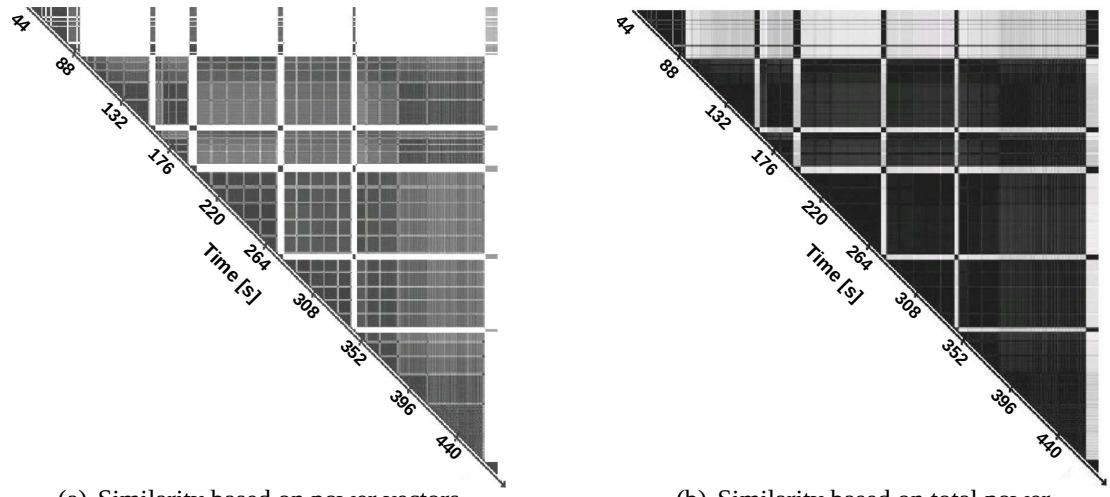
$$NM(r,c) = \sum_{i=1}^{22} |NPV_r(i) - NPV_c(i)| \tag{3.2}$$

The reason behind normalization is to emphasize the differences between the distribution of power into the vector components. In other words, the similarity metric demonstrated here is based on the relative ratios of component powers independent of vector magnitudes. Consequently, the similarity matrix discriminates small magnitude power vectors better than the original approach.

Both normalized and non-normalized techniques tend to disregard certain types of dissimilarities. Therefore, in order to restrict ourselves to similarities that satisfy both cases, we developed an intersection of the above two matrices so that two vectors are considered similar only if they are similar under both measures. We perform this by adding the two matrices after normalizing each to unity in order to weight both measures equally. We then limit the resultant matrix elements to a maximum value of 1. That is, 1 is representative of maximum dissimilarity and 0 corresponds to perfect similarity. We perform a limiting operation, rather than normalization, after adding the two matrices in order to achieve a final similarity metric which emphasizes dissimilarities. In other words, we want a similarity and a dissimilarity to result in dissimilarity. Consequently, the final similarity matrix is constructed from the two previous similarity matrices as shown in Equation 3.3, where SM, AM and NM represent final, original and normalized similarity matrices respectively.

$$SM(r,c) = \min\left(\frac{AM(r,c)}{\max\limits_{r',c'}\left(AM(r',c')\right)} + \frac{NM(r,c)}{\max\limits_{r',c'}\left(NM(r',c')\right)}, 1\right) \tag{3.3}$$

The matrix plot representing this final similarity metric is shown in Figure 3.4(a) for the `gzip` benchmark. This final plot identifies both ratio based and magnitude based dis-



(a) Similarity based on power vectors (b) Similarity based on total power

Figure 3.4: Similarity matrices based on power vectors and total power.

similarities relatively well. Moreover, the emphasis on dissimilar regions also provides much sharper distinction between the degrees of similarities. To provide a comparison, we also show the observed similarity information by considering solely total power in Figure 3.4(b). Here, the similarity information is calculated by considering total power as a single dimensional power vector. Therefore, the similarity matrix directly corresponds to the variation of absolute total power difference among execution points. Each matrix entry (r,c) is computed as $|P_r - P_c|$, where $P_{r,c}$ represent the total power samples at execution points r and c. In comparison to Figure 3.4(b), the final similarity matrix plot reveals significantly higher information regarding program power phases, both at lower power and higher power execution regions.

This final similarity metric demonstrates that power vector based phase analysis provides detailed insight into workload power behavior, which cannot be directly observed from total power. It identifies several regions with distinct power characteristics, which are considered to have similar behavior from total power observations. In the rest of this chapter, we utilize this similarity metric to identify program phases and to characterize program power behavior.

3.3 Representing Execution with Signature Vectors

One primary aim of power phase analysis is to achieve a small characteristic set of execution phases that capture most of the workload's power behavior. This set of phases are useful for representative evaluations of workload characteristics by monitoring a small portion of the overall application. Moreover, the sequence of the observed phases can guide phase-driven, runtime adaptations. Our methodology is best described as "representative sampling" [169], where we identify a small set of *execution points* that are representative of the overall power traces of programs. Second, we also define a set of *representative power vectors*, which are not directly associated with execution points. Instead, they define a program "signature" based on their component powers and their ordering in the timeline. These signature vectors can be used in program identification and phase prediction.

For both of these problems, we use various clustering algorithms to group execution points. Here, we demonstrate our results with a simple runtime *thresholding* or *first pivot* method. Later we consider more elaborate clustering methods. In the thresholding method, we specify a threshold as a percentage of maximum dissimilarity between sample pairs. Then, as the execution moves forward in time, we identify samples that lie within the threshold criterion. The thresholding algorithm performs this similarity grouping to generate a *grouping matrix*. Similar to the initial similarity matrix, the grouping matrix illustrates which other points are similar to each execution point for a given threshold. To divide execution into sets of phases, we consider each new execution sample r that does not fall into a prior phase category as a *pivot*. In the forward execution path, we identify the points $(r, c_i > r)$ that lie within a threshold distance of r and tag the corresponding execution points c_i as the same group.

There are three primary components of our workload power behavior characterization technique: (i) generating representative vectors, (ii) selecting execution points, and (iii) reconstructing power traces using these representative samples. We describe the basics of each of these steps here.

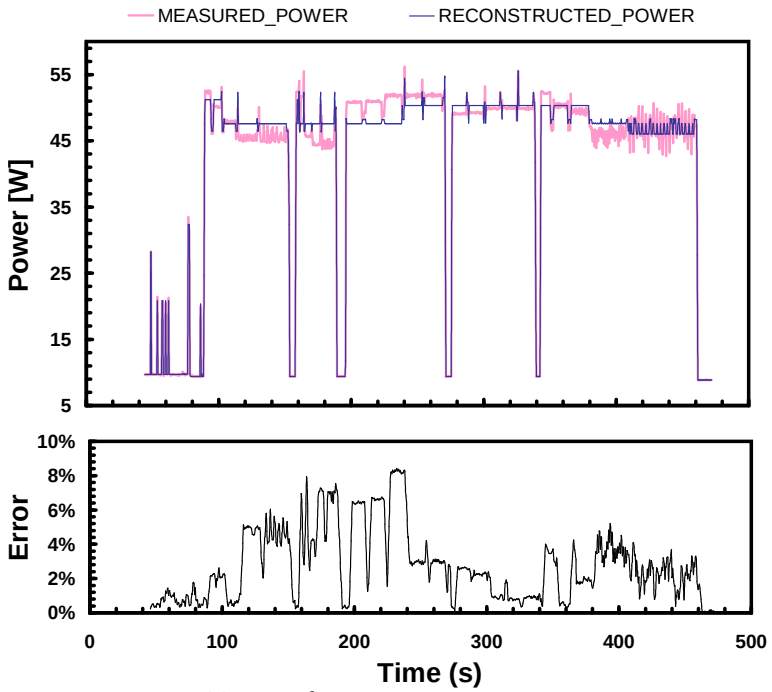
Generating Representative Vectors: To represent workload power behavior with a small set of signature vectors, we define a representative vector for each phase group, which is the set of all vector instances belonging to that phase. Consequently, the number of phases is equal to the number of representative vectors for a given trace. Each representative vector is the component-wise arithmetic average of all the vectors belonging to the corresponding group.

Selecting Execution Points: Unlike representative vectors, the execution points refer to actual regions in workload execution. These points identify characteristic regions with specific power behavior that can be used for more detailed exploration of power behavior, in a similar fashion as proposed in simulation studies [27]. In our approach, we choose the earliest occurring member of each group—the pivot—as the selected execution point for that group. Thus, as the distance between the startpoint of a group and all other members of the group is always bounded by the given threshold, we can always formally specify an upper bound on the amount of difference between the originally estimated power and our power approximation based on the selected set of power vectors.

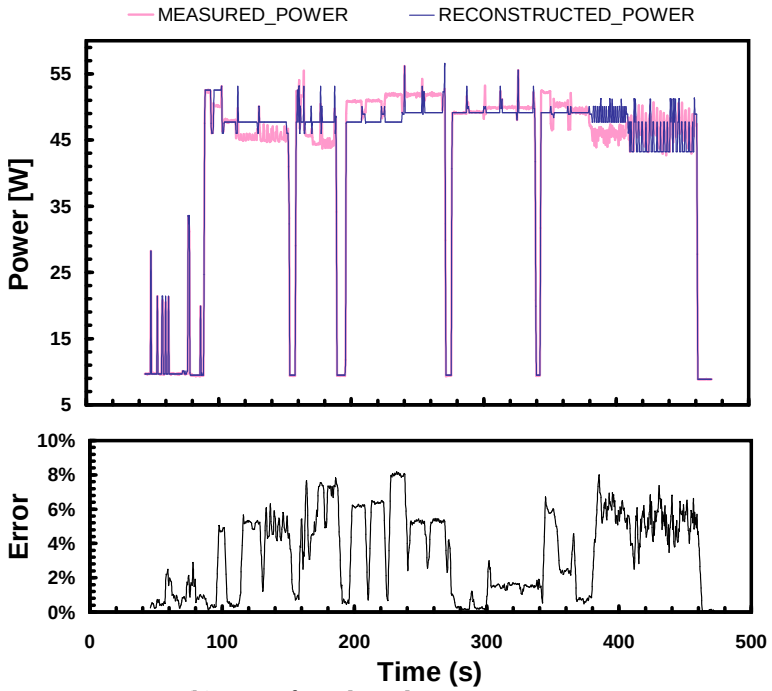
Reconstructing Power Traces: The definitions of representative vectors and selected execution points can characterize overall execution with a small set of vectors. For each sample, the representative vector for the corresponding phase the sample belongs to is that sample's power vector. Thus, we reconstruct the whole power trace with only the representative vectors. Similarly, with the selected execution points, we identify the corresponding power vectors and construct the power trace based on the selected execution point vectors. These approaches closely approximate original application power behavior with minimal information.

3.3.1 Representation Accuracy with Power Phases

Here, we quantify our approximation error with respect to the actual application power. Figure 3.5 shows the reconstructed power traces together with the actual power behavior



(a) Error for representative vectors



(b) Error for selected execution points

Figure 3.5: Reconstructed power behavior and absolute error in total power estimates for `gzip` reconstructed from representative vectors and selected execution points.

for both representative vectors and selected execution points for the `gzip` benchmark. In the lower plots, the figure also shows the absolute error in reconstructed total power for

both cases. In this example, we classify `gzip` execution into approximately 7 phases per dataset with the first pivot method using a 10% similarity threshold. Thus, the reconstructed power traces rely on around 1% of the actual execution information to capture `gzip`'s power behavior.

The errors for representative vectors and vectors based on selected execution points differ in one major aspect. Since the startpoints of groups are the selected execution points, the sum of absolute errors for components is always within the specified threshold for selected execution points while the errors for representative vectors are not necessarily bounded with the same threshold. However, as the representative vectors are the averages of each group, they have a lower average error over the whole timeline. For representative vectors, the RMS error is 2.31W (4.9%), while for execution points, the RMS error is 3.08W (6.6%).

Finally, Figure 3.6 shows a summary of the experiments with different applications. This figure shows the average variation between actual and reconstructed power for different numbers of phases. In addition, it shows the maximum and minimum observed variation among the tested applications. For a specific phase number, the whole power behavior is characterized with the same number of representative vectors. The variations are averaged over all applications. As the number of phases increase, the characterization accuracy improves and reconstructed power behavior converges to the actual power with increasing number of phases.

The important observation in Figure 3.6 is that a small number of power phases can capture the most of the application power behavior. For most applications around 10 different phases can represent overall power variation of applications within 5% of the actual power. This typically corresponds to less than 1% of the whole execution.

Overall, these results demonstrate that power phase analysis with event-counter-based power vectors and the composite similarity metric can effectively capture varying workload power behavior. The following sections evaluate our phase characterization in more detail,

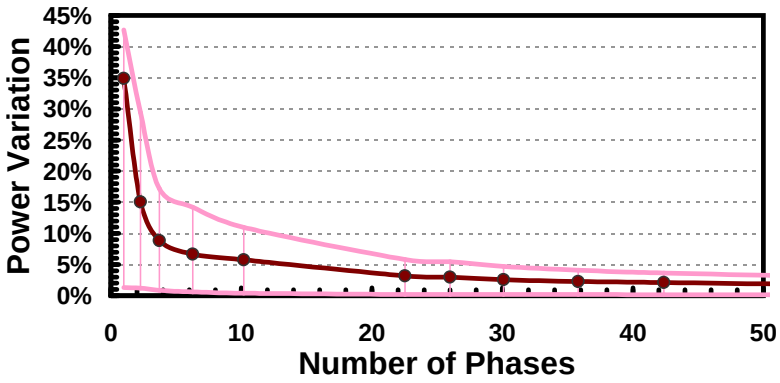


Figure 3.6: Average variation from actual power for the SPEC applications with different number of phases together with the maximum and minimum variation bands for different applications.

employing comparative studies with other existing representation features.

3.4 Comparing Event-Counter-Based Phases to Control-Flow-Based Phases

While the previous sections have focused on event-counter-based program characterizations with power vectors, various prior studies have demonstrated that phase behavior can be observed via different features of applications. The rest of this chapter compares how our phase characterizations perform relative to the existing phase analysis approaches. Most of these approaches fall into two main categories: In the first category, application phases are determined from the control flow of the applications or the program counter (PC) signatures of the executed instructions [41, 72, 90, 109, 137, 151, 152, 153]. In the second category, phases are determined based on the performance characteristics of the applications [14, 35, 44, 84, 169, 176].

Although there have been some previous efforts to compare or evaluate phase characterization techniques [6, 40, 108], they do not perform a direct comparison of the two main approaches, namely control-flow-based and event-counter-based phases. Moreover, there is generally a missing link between phase characterizations and the ability to use them to represent power behavior, especially on real systems. Such power characterization is very

important for real systems, as a primary goal of phase characterization is dynamic power management of running systems.

This study primarily evaluates these techniques for accurate power behavior characterization on a real system. We compare these with respect to the actual, measured runtime power dissipation behavior of applications. Specifically, we look at phase analysis based on basic block vector (BBV) features of an application [152] to determine regions of similar power behavior. We compare this to phases determined by a particular set of performance monitoring counter (PMC) events that are chosen to reflect power dissipation [85]. We test the power characterization accuracy of these methods on 21 benchmarks from SPEC2000 suite and 9 other benchmarks derived from commonly used desktop and multimedia applications. In general, tracking performance metrics performs better than tracking control flow in identifying power phase behavior of applications. Additionally, specific examples from real applications demonstrate cases where power phase behavior cannot be deduced from code signatures.

3.5 Dynamic Instrumentation Framework

To collect synchronous control flow, event counter and power information during an application's execution, we use dynamic instrumentation via Pin [121]. Pin provides several flexible methods to dynamically instrument the binary at different granularities. This first step, *instrumentation*, simply decides where in the native code the additional procedures to analyze the application behavior should be inserted. Afterwards, whenever one of these instrumentation checkpoints is reached, Pin gains control of the application and injects corresponding analysis routines. During execution, each time the instrumented locations are visited, their injected analysis routines also execute, providing the dynamic application information. This second phase of operation is called *analysis*. Pin utilizes a single executable, *Pintool*, to perform instrumentation and analysis on an application.

Figure 3.7 presents an overview of the experimental setup for power phase analysis with

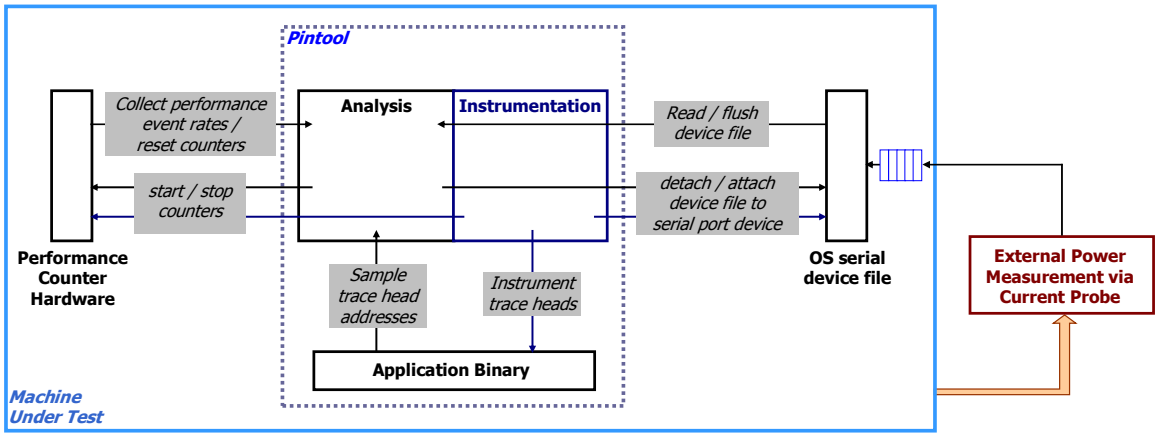


Figure 3.7: Experimental setup for power phase analysis with Pin.

Pin. The Pintool uses trace-level instrumentation to keep track of executed code traces. The analysis routine consists of three levels of hierarchy. The first level simply provides an account of executed instructions. This is implemented as an inlined conditional at the trace level to improve performance and to avoid perturbing power behavior. The instrumented traces include multiple basic blocks consisting of around 50 instructions. The second level samples one PC address approximately every 1 million instructions. The highest level analysis is invoked every 100 million instructions. This routine generates one BBV from the 100 PC samples, reads performance statistics from PMCs and logs the measured power history from the serial device file. These three sources of data collection are shown with the three incoming arrows to the analysis routine of the Pintool.

It is important to isolate application behavior from Pin operation. Pin provides control flow information about the application on its own. However, performance monitoring and power measurements are out of Pin's control. Therefore, Pin routines disable data logging for power and performance at routine entries, and reenable data logging at routine exits. Under Pin execution, instrumentation and analysis are temporally intermixed. Therefore, we use these handles during both instrumentation and analysis.

External, live power measurements provide real power information to compare with the power phase characterizations. Power measurements are performed by measuring the current flow into the processor with a current probe. This measurement information is then

fed back to the measurement system over the serial port interface.

To isolate the application power behavior from Pin analysis and instrumentation, we use certain controls within the instrumentation and analysis routines of our Pintool. These handles detach/attach the serial device driver from the device file at routine entries/exits via `termios` flags. This approach preserves previous application power history, while preventing further logging while inside an instrumentation or analysis routine. At the end of a 100 million instruction sampling period, the highest level analysis routine halts logging and reads the logged power history for the past sampling period. This history is then averaged and is assigned as the observed power for the past sampling quantum. Afterwards, the buffer is flushed and reenabled for logging at the start of the next sampling interval.

Similar to the power measurement method, several handles control PMC monitoring from within the Pintool. Pintool initialization first configures the events to be monitored. This is the most heavyweight operation, but it only occurs once, before the application execution commences. We selectively halt/start performance monitoring at instrumentation and analysis routine entries/exits. This is used to avoid polluting the PMC information with Pin execution. Although we provide the start/stop handles to all routines, after our initial experiments we do not invoke them for instrumentation and the second level analysis routines, as their costs are comparable. This trade-off only affects PMC information without any effect on control flow information and power measurements. The highest level analysis routine reads the past PMC statistics and resets the counters for the following sampling period.

3.5.1 Program Counter Sampling and BBV Generation

To track control-flow-based application phases, we use the BBV approach [152]. BBVs summarize application execution by tracking both which basic blocks of the application are touched and how many times each basic block is visited during a sampling interval. BBVs represent application execution behavior by providing both working set information and execution frequencies for different basic blocks [40]. BBVs are constructed from exe-

cution flow by mapping executed PC addresses to the basic blocks of an application binary. Originally, each component of a BBV is a specific basic block, and the magnitude of the component represents how often the corresponding basic block has been executed for a past sampling period. For practical purposes, BBVs are generally mapped into smaller dimensional vectors via random projection/hashing, component analysis, or the elimination of the least significant dimensions [6, 46, 152, 153, 182, 183].

Our implementation uses Pin to sample the PC addresses at trace heads. As each trace head is also a basic block start address, each sampled PC actually corresponds to a specific basic block. For sampling periods, we sample one PC every 1 million instructions similar to prior work [6] and construct a BBV at every 100 million instructions. Thus, each BBV has an $L1$-norm—sum of vector components—of 100. To apply dimension reduction, we choose 32 buckets based on previous work [153]. We use a variation of Jenkins' 32 bit integer hash function [91] to reduce the large and variable BBV dimensions into common 32 dimensional vectors. As has been discussed in previous studies [108], sampling always incurs some amount of information loss. Therefore, we compare full-blown BBVs, constructed from complete PC information, to our sampled BBVs with similarity matrices [152]. Both methods reflect the major phase content in terms of execution flow similarity and lead to similar phases for small numbers of target phase clusters.

3.5.2 Using Performance Counters to Generate PMC Vectors

The original power vectors are 22-dimensional vectors that require four counter rotations to collect. To use this information in the dynamic instrumentation framework without incurring too much reconfiguration overhead, we reduce this original event counter set to a final set of 15 PMC events that can be monitored simultaneously without conflicts. Therefore, no PMC configuration is required except at the initial Pintool startup. Factor analysis [42] is used to choose the reduced set of event counters. This process works by eliminating highly correlated dimensions. We call these reduced dimension vectors *PMC vectors*. While the original 22-dimensional vectors were developed to provide a one-to-one mapping between

the physical processor components and the estimated power breakdowns, such a mapping is not directly required for tracking power phases. The reduced set of PMC vectors perform almost identically to the original power vectors in identifying phases. The complete list of chosen performance counters are shown in Table 3.1 together with the applied mask configurations that define the particular event subsets we choose to track.

PMC Event	Mask	Description
IOQ_allocation	0x0EFE1	*I/O Queue and Bus Sequence Queue allocations from all agents*
BSQ_cache_ref	0x0507	*L2 cache read and write accesses*
FSB_data_activity	0x03F	*Front Side Bus utilization for reading, driving or reserving the bus.*
ITLB_reference	0x07	*ITLB translations performed*
uop_queue_writes	0x07	*All µops written to the µop queue*
TC_deliver_mode	0x038	*Number of cycles the processor is buiding traces from instruction decode*
uop_queue_writes	0x04	*µops written to the µop queue by microcode ROM*
x87_FP_uop	0x08000	*All x87 floating point µops executed*
LD_port_replay	0x02	*Number of replays at the load port*
x87_SIMD_moves	0x018	*Executed x87, MMX, SSE and SSE2 load, store and register move µops*
ST_port_replay	0x02	*Number of replays at the store port*
branch_retired	0x0F	*All branches retired*
uops_retired	0x03	*Number of µops retired*
front_end_event	0x03	*Number of loads and stores retired*
uop_type	0x06	*Tags load and stores (Does not count)*

Table 3.1: The set of chosen performance counter events and mask configurations.

Every 100 million instructions, we collect the performance event counts and cycle count for the past sampling period. We then convert these event counts into per-cycle rates. These 15 event rates are then used to construct the 15 dimensional *PMC vector*, which gauges the similarity of execution samples in a similar manner as BBVs.

3.6 Phase Classification

We cluster BBV and PMC vector samples into phases with multiple clustering algorithms. First, a runtime *First Pivot Clustering* method assigns samples to phases as they are observed. We also experiment with a more detailed method, namely *Agglomerative Clustering* [42]. There are two variations of this method: *complete linkage* and *average linkage*.

The original first pivot method provides an upper bound to the distance within each

phase, but it does not guarantee a fixed number of phases. We change this to an iterative process, where the threshold is changed dynamically based on the acquired number of phases. With this modification, we classify both BBVs and PMC vectors into 5 final phases after a few iterations.

Agglomerative clustering is a tedious bottom-up approach for clustering samples into phases. In this approach, the clustering algorithm starts with an initial solution of N clusters, where N is the number of samples. At each iteration, the algorithm compares all pairwise combinations of the current set of clusters and finds the best candidate pair of clusters to combine into a single cluster. The pairs are compared based on a *linkage* criterion, which determines the best candidates. We experiment with two types of linkages, complete and average linkage. Average linkage compares the average distance between all sample pairs belonging to two different clusters. For two clusters with i and j samples respectively, it computes the distance between all the $i \cdot j$ pairs and finds the average distance between the clusters. It chooses to combine two clusters with the minimum average distance. This leads to clusters with similar ranges in all dimensions. Complete linkage compares the maximum pairwise sample-distance among clusters. It combines the clusters with the least maximum distance among all their pairs. Consequently, the final set of clusters have similar ranges among most of their samples, although the range across each dimension can be different.

3.6.1 Evaluating Phase Classifications

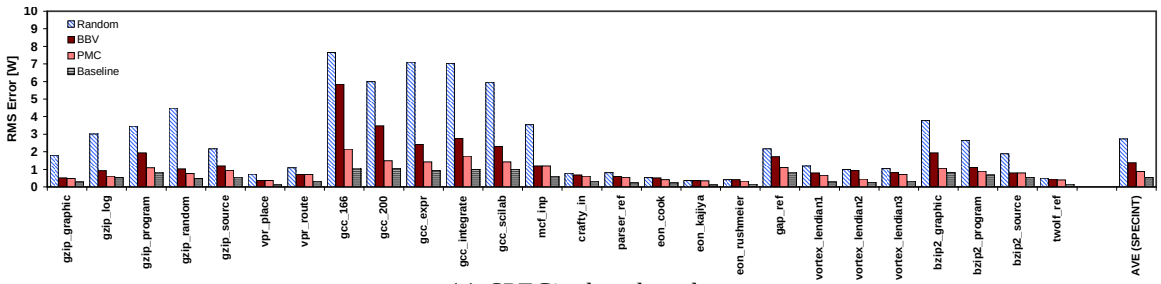
We evaluate the quality of generated phase clusters by comparing the measured power at each sample to the aggregate power for the whole cluster the sample belongs to. For a benchmark with N samples, each sample i $(i = 1, \ldots, N)$ is an element of one of the final phase sets P_j $(j = 1, \ldots, 5)$. Each sample has a corresponding set of data $[bbv_i, pmc_i, pwr_i]$, where bbv_i and pmc_i are the corresponding BBV and PMC vectors used during phase clusterings, and pwr_i is the measured power value during sample i's execution. For each phase P_j, we compute a "representative power", R_j, as the arithmetic average of the power values for the total N_j samples belonging to that phase. Then, for each sample i, we compute the

squared difference between the sample's actual power value pwr_i and the representative power R_j for its owner phase P_j. We denote R_j values corresponding to each sample i with RS_i. For example, for a sample k that belongs to phase P_2, $RS_k = R_2$. The rooted average of the squared differences over all samples is the final RMS error figure E_{RMS}. Equation 3.4 summarizes this error computation.

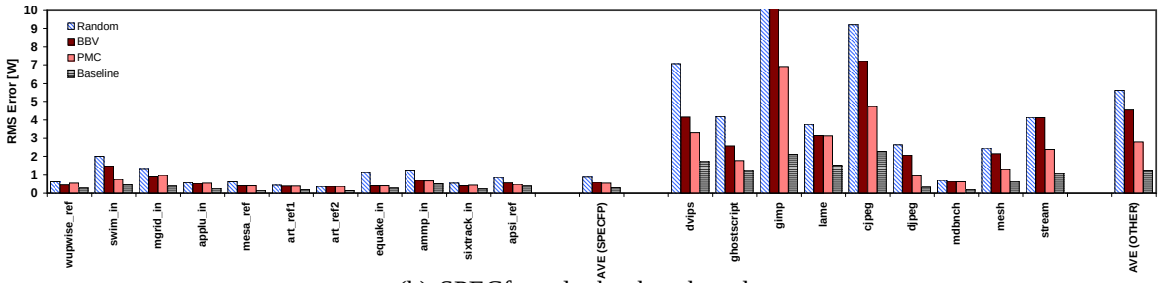
$$R_j = \frac{\sum_{i \in P_j} pwr_i}{N_j} \qquad (j = 1, \ldots, 5)$$
$$E_{RMS} = \sqrt{\frac{\sum_{i=1}^{N} (pwr_i - RS_i)^2}{N}} \tag{3.4}$$

To gauge the ability of the phase classification techniques to discern application power behavior, we also provide the error boundaries that can be achieved with perfect knowledge of power information—a lower bound—as well as without any knowledge of application behavior—an upper bound. To compute lower error bounds, we look directly at the measured power. We apply all three clustering algorithms to each benchmark's power information and for each case choose the smallest error value achieved. This "gold standard" measure is the *baseline error* in our results. For the upper error bounds, a separate clustering method assigns each sample to any of the final target phases randomly, without using any application behavior information. We refer to the results of this "uninformed" phase characterization as *random error*.

Our experiments use 11 SPECint benchmarks—all except `perlbmk` due to compilation problems—and 10 SPECfp benchmarks—excluded are F90 benchmarks. We experiment with all reference datasets for the 21 SPEC benchmarks leading to a total of 37 different experiments. In addition to SPEC, we also use 9 other benchmarks from previous studies and derived from well-known applications. These benchmarks are `ghostscript`, `dvips`, `gimp`, `lame`, `cjpeg`, `djpeg`, `mesh`, `stream` and `mdbnch`. For some cases, we alter the dataset



(a) SPECint benchmarks



(b) SPECfp and other benchmarks

Figure 3.8: Power characterization errors for BBV and PMC phases with agglomerative clustering and complete linkage.

or iterations for the benchmarks to achieve longer execution times [88].

3.7 Phase Characterization Results

Although we perform our experiments for all clustering approaches, the observed results are consistent regardless of clustering approach [88]. Therefore here Figure 3.8 shows the overall results for only agglomerative clustering with complete linkage. Figure 3.8(a) shows the results for the SPECint applications and Figure 3.8(b) shows the results for SPECfp and other applications. The figures show the upper (*random*) and lower (*baseline*) error bounds for each application and the achieved errors with BBV and PMC based approaches. They also show the average accuracies for the SPECint, SPECfp and other experimented benchmarks.

Comparing among the three sets of applications, SPECfp applications lead to relatively low errors even with random phase clustering for some cases. This is due to the generic flat power behavior of these benchmarks (`applu`, `art`, `sixtrack`, `wupwise`). In some other cases, benchmarks go through specific initialization (i.e. `equake`) or periodic phases (i.e.

ammp) with significant changes in all control flow, performance and power features. In these cases, both BBVs and PMCs achieve very good power characterizations approaching baseline errors.

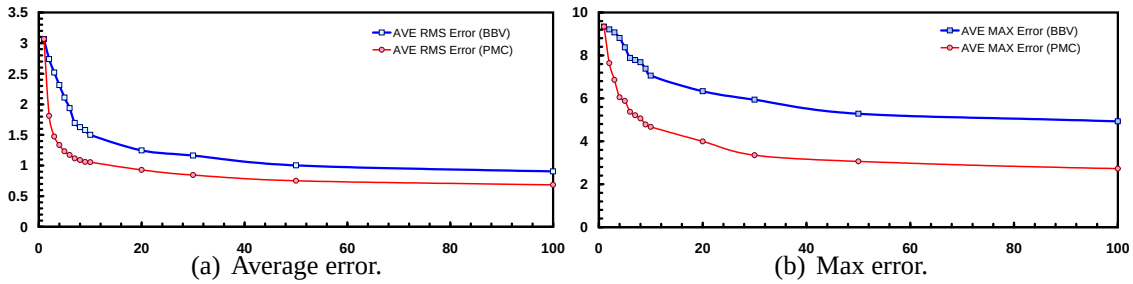
SPECint shows significantly higher errors for all approaches due to higher variations in behavior. In many cases, BBVs and PMCs have significant improvement over random clustering. This shows the benefit of phase tracking for power behavior characterization.

Most of the other experimented benchmarks show significantly higher error ranges due to their high power variability based on input data characteristics and functional behavior. In these cases, applying phase analysis, especially with PMCs, proves to be very useful in identifying similar power behavior.

Overall, for the three benchmark sets, BBVs achieve errors that are on average 48% less than random clustering errors for benchmarks with non-flat power behavior. PMC phases lead to 66% less error than random clustering. For the PMC based approach, power characterization accuracies are 2-6X better than random clustering. Performing the same comparisons with respect to baseline errors, BBVs achieve 2.9X higher error on average compared to baseline, while PMC error is 1.8X of the baseline figure. These comparisons show that BBV and PMC phase analyses have significant benefit in characterizing power behavior. However, there still exist opportunities to improve power phase behavior characterization of applications.

As the above measures also indicate, in almost all experimented cases, our PMC based phase analysis represents power behavior better than a BBV based approach. PMCs lead to 2.2% and 1.4% errors for SPECint and SPECfp, while BBVs achieve 3.4% and 1.5% errors. For the other benchmarks, PMCs and BBVs have 7.1% and 14.7% average errors respectively. For most of the benchmarks PMCs achieve 30-40% less errors than BBVs with an average of 33%. Thus, although both techniques provide useful features to identify power phase behavior, in general PMCs perform better.

Thus far, all analyses have used a fixed target number of 5 phases to enable meaningful



(a) Average error. (b) Max error.

Figure 3.9: Variation of errors with respect to number of final phases.

averaging across benchmarks. However, we have also experimented with different numbers of target phases to verify the reliability of our results. Figure 3.9 shows the achieved errors as both RMS and maximum observed values. For each benchmark, we compute the RMS and maximum error figure for each target phase count. Afterwards, we average these values over all benchmarks to reach a single error figure for each target phase count.

Intuitively, for a single final phase, both BBVs and PMCs will reach the same error, equivalent to the standard deviation of all the power samples of the benchmark. As the number of phases increases, errors for both methods will decrease with different slopes. As number of target final phases grows towards infinity, both error curves will converge to 0, i.e. where each phase is a singleton sample. Figure 3.9 shows the behavior up to 100 phases. As phase counts grow beyond 100, both curves approach 0. PMC based phases perform consistently better, independent of the number of final phase clusters.

3.8 What Control Flow Information Does Not Show

There are multiple aspects of application behavior that can cause the control flow and performance based approaches to reach different phase characterization conclusions. *Dynamic change in data locality* during an application's execution can cause the power behavior to significantly change. While this change can be easily recovered from memory performance metrics, code signatures cannot reflect this as execution footprints are not altered. *Effectively same execution* represents the converse of the above effect. In various applications, multiple procedures or code segments perform similar processes, leading to practically

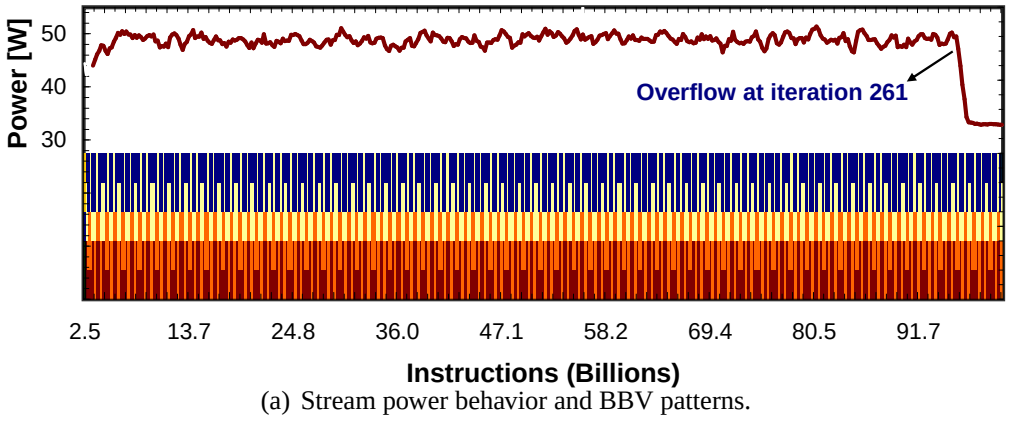
identical power behavior. These are considered as fairly different phases in terms of control flow, which may result in many different phase clusters that do not reflect actual changes in program power. Typical examples for these are scientific or other iterative processing applications performing different tasks on an input with similar power/performance implications [68]. *Operand dependent behavior* may result in similar effects as the first case, where power and latency of a unit depends on the input operands, despite the same control flow. Typical cases for these are overflow handling and scaling of execution based on the input operand values or widths [22].

This section demonstrates two of these effects, operand dependent behavior and effectively same execution based on observations from actual applications.

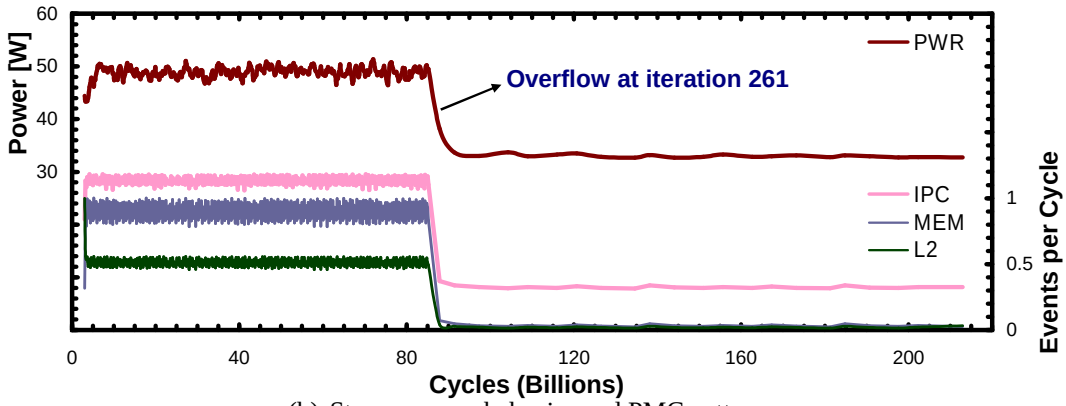
3.8.1 Operand Dependent Behavior

The `stream` benchmark shows a simple example of operand dependent behavior and its implications on power. `Stream` performs four repetitive operations with simple vector kernels. It operates on three vectors, `a`, `b` and `c`. The four operations are *copy* ($c[j] = a[j]$), *scale* ($b[j] = scalar * c[j]$), *add* ($c[j] = a[j] + b[j]$) and *triad* ($a[j] = b[j] + scalar * c[j]$). It targets at measuring sustainable memory bandwidth with vectors larger than cache sizes and by avoiding data reuse. There exists a positive feedback between each iteration of the four described operations. This causes the the FP operations to overflow at iteration 261, where the first vector `a` overflows at *triad*. This is then propagated to vectors `b` and `c` in the next iteration. This overflow causes the three FP kernels to experience a slowdown larger than 10X, while the *copy* operation is not significantly effected. Consequently, power dissipation experiences a drastic phase change, while the execution path is still conserved.

Figure 3.10, shows the resulting behavior in terms of power, BBV signatures and PMC signatures. Figure 3.10(a) shows the power (top) and BBV signatures (bottom) with respect to executed instructions. It shows the BBV signatures as stacked vector sample bars, where the magnitude of each vector component adds on top of the stack. Here, we see the repetitive BBV vector patterns throughout the execution, corresponding to the 4 different



(a) Stream power behavior and BBV patterns.



(b) Stream power behavior and PMC patterns.

Figure 3.10: Power phase change at overflow condition for the `stream` benchmark. Upper plot (a) shows BBV signatures, unable to detect the phase change. Lower plot (b) shows PMCs detecting the change. Lower plot is drawn with respect to elapsed cycles to show the actual time behavior.

operations repeated 275 times. As the control flow is repetitive, the sudden power drop goes undetected with BBVs. Figure 3.10(b) shows the same execution with power (top) and some of the PMC vector samples (bottom). Shown PMC metrics represent instructions per cycle (IPC), L2 cache access rates (L2) and memory access rates (MEM). This figure shows the execution with respect to cycles to emphasize the actual effect of overflow on elapsed time in different power phases. While the lower power phase occupies less than 6% of the executed instructions, the time spent in this phase is more than half of the total execution. Tracking PMCs easily identifies this power phase change, resulting from operand dependent behavior of `stream`.

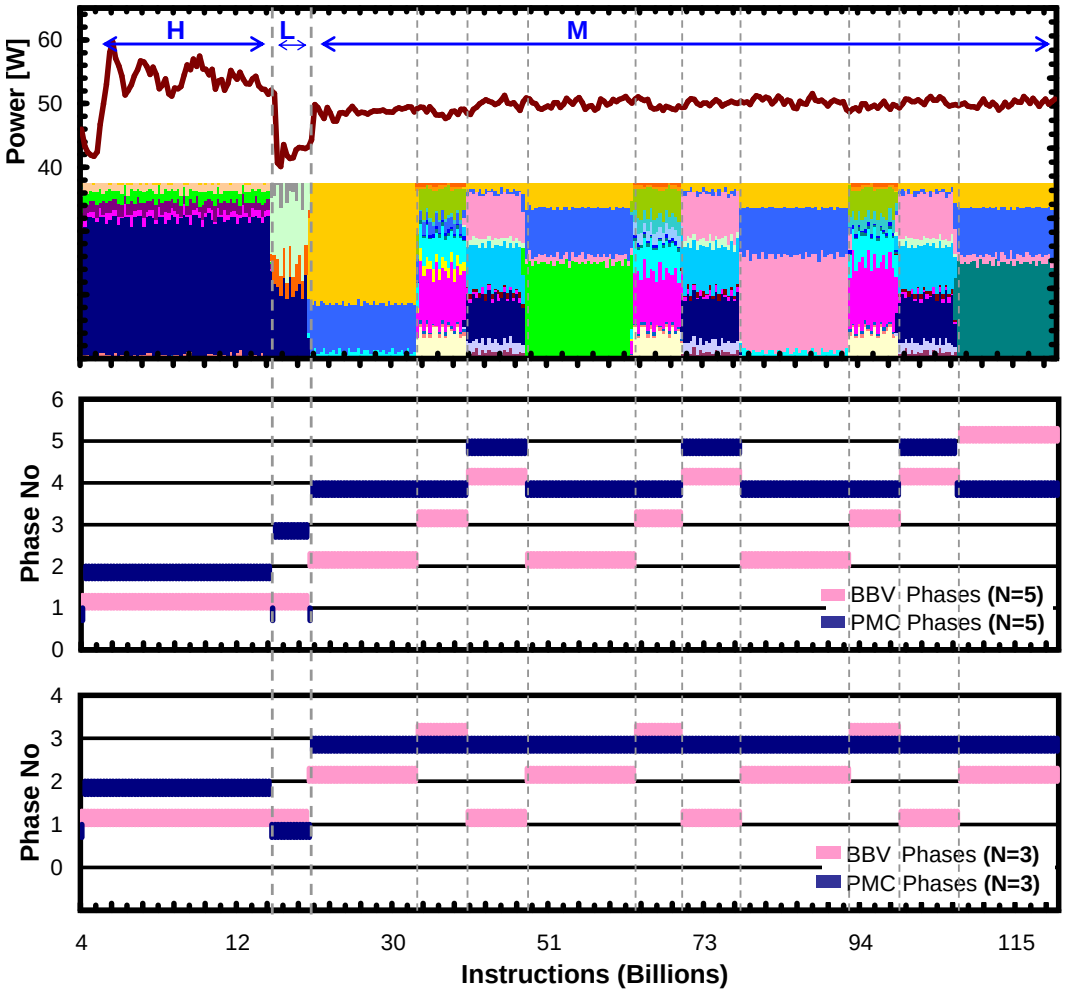


Figure 3.11: `Mesh` power and BBV signatures (top) and generated PMC and BBV phases with target cluster numbers of 5 (middle) and 3 (bottom). Multiple control flow phases with effectively same power characteristics disguise actual power phases in BBV based classification. Actual power phases are labeled as *H*, *L* and *M*, for high, low and medium power dissipation regions.

3.8.2 Effectively Same Execution

Figure 3.11 demonstrates another example of the discussed effects with the `mesh` benchmark. This example shows how PMC vectors and control flow can reach different phase characterizations due to effectively same execution.

The top graph of Figure 3.11 shows the measured power behavior. We can easily separate `mesh` execution into three power phases by observing the power trace. These "actual" power phases are labeled as *H*, *L* and *M* on the power trace, representing phases with high,

low and medium power consumption. Underneath the power trace, the figure shows the corresponding 32-dimensional BBV vector patterns for each sample. Several distinct control flow phases are observable from the BBV patterns. Vertical dotted lines separate each of these distinct regions. These regions correlate well with `mesh` tasks. The first high power phase corresponds to the sorting task after reading nodes and initialization. This task sorts nodes based on their types. It operates mainly in L1 cache and is computation intensive. The following low power phase results from *SetBoundaryData* task which sets the values for boundary nodes. This task mostly accesses L2 and has low overlapping computation, which leads to less power. After this task, `mesh` repetitively operates on three computation tasks, namely, *ComputeForces()*, *ComputeVelocityChange()* and *SmoothenVelocity()*. These constitute the medium power phase of `mesh`. All of these tasks also make significant L2 accesses. However, their overlapping FP computations lead to relatively higher power.

The lower two plots of Figure 3.11 show the phase classifications performed by BBVs and PMCs. In these plots, the y axis shows different phases ranging from 1 to 5 for the first case and 1 to 3 for the second. For each sample, we add a tick mark above the horizontal line corresponding to its phase assigned by BBV classification. We also add a tick mark below the horizontal line that corresponds to each sample's PMC phase. These marks then form the bands of phases seen in these plots. For example, for the case with 5 phases, low power phase of `mesh` is classified into phase "1" by BBVs and phase "3" by PMCs.

These plots show the impact of effectively same execution in phase classification. For $N = 5$, PMCs correctly identify the three actual power phases. BBVs on the other hand, collapse the high and low power phases into a single phase. This is because BBVs identify several different large-scale control flow phases. Clustering starts to overlap these based on their $L1$-distances, and these result in combining the high and low phases of power. The three repetitive control flow phases with effectively same power behavior are seen as the more different phases by BBVs. For $N = 3$, BBV phases still show more sensitivity to the three repetitive tasks of medium power phase and assign them to three different phases. In

this case, all high, low and parts of medium power phases are assigned to same phase ("1") by BBVs. In contrast, PMCs show very good fidelity. They successfully identify three power regions and assign them to different phases.

This example demonstrates the clear impact of effectively same execution on control-flow-based phase characterization. Overall, both BBV and PMC phases provide a good account of application power phase behavior. PMCs usually show a better mapping to power behavior due to both their proximity to the actual flow of power in the processor, as well as due to these discussed sources of disagreement between power and code signatures.

3.9 Related Work

A number of previous works investigated program phase behavior including simulation-based [35, 41, 101, 151, 152, 153] and runtime [128, 132, 131, 169] program profiling techniques to identify phase behavior. These works span diverse areas such as identifying representative simulation point samples, predicting phases, generating reduced datasets, and managing configurable hardware with program signatures. Most of these research studies focus on either control flow or performance characteristics of applications. Iyer and Marculescu [90], Dhodapkar and Smith [41], Sherwood et al. [152, 153], Huang et al. [75], and Lau et al. [109] analyze control flow behavior of applications via different features such as subroutines, working sets and basic block profiles. These studies use simulation based methods to identify application phases for summarizing performance and architectural studies. Patil et al. also look at control flow phases with real-system experiments [137].

Cook et al. show the repetitive performance phase characteristics of different applications using simulations [35]. Todi [169], Weissel and Bellosa [176], and Duesterwald et al. [44] utilize performance counters to identify performance based phases. They use performance statistics to guide dynamic optimizations and metric predictions. These works do not consider the power behavior of applications. Chang et al. apply process power profiling

to determine software power breakdowns [29]. Hu et al. describe a compile time methodology to find basic block phases at runtime for power studies [72]. This study looks at control flow information from a compiler perspective.

There are also previous studies that compare or evaluate phase characterization techniques. Dhodapkar and Smith perform a comparison between different control flow techniques [40]. Annavaram et al. sample the program counter as a proxy to control flow and show the correlations between code signatures and application performance [6]. Lau et al. also look at control flow and performance of applications by linking program counter to procedures and loops of applications via profiling [108]. In comparison, our work looks at the direct comparison of two phase characterization features, BBVs and PMCs, with runtime measurement feedback for real power evaluation on a real system.

3.10 Summary

This chapter presented a power phase analysis methodology for characterizing program power behavior based on power vectors sampled at program runtime with hardware performance counters. We used performance-counter-based vectors to identify execution regions with similar power behavior. Based on this similarity information, we could represent application power behavior with a small set of power phases that are acquired via different clustering approaches. Our experiments demonstrated that these sets of power phases capture workload power variations within 5% of actual behavior. We have developed an experimental framework for comparing both control-flow-based and performance-monitoring-based phase techniques. Our results showed that both control-flow and performance features provide useful hints to power phase behavior. However, in general, performance-counter-based phase tracking leads to approximately 33% less power characterization errors than code signatures.

Overall, the results presented here show a roadmap to effective power phase analysis in real systems. As our power phase analysis is based on a real system, it can readily be

used in architecture and systems research, and can provide significant insights for dynamic management and workload characterization techniques.

Chapter 4

Detecting Repetitive Phase Patterns with Real-System Variability

The previous chapters have focused on the characterization of workload phases, and how we can use these phases to efficiently represent application power behavior. In particular, Chapter 2 demonstrated that performance monitoring events provide useful information about the power consumption of processors. Chapter 3 showed that similarity analysis methods that are applied to these events characterize the phase behavior in the power consumption of computing systems. However, to be able to employ this phase information in real-system dynamic management studies, it is also important to develop methods that identify the repetitive phase behavior of applications. While prior chapters have focused on characterization of this phase behavior, this chapter specifically focuses on methods for detecting repetitive phases in application execution on real systems. It describes and evaluates a new framework that helps extract the recurrent information in phase behavior despite system-induced variability effects.

Most of the recent phase tracking work has focused on simulation studies. There the largely repeatable and deterministic behavior means that phases can stand out quite clearly. In order to move towards using on-the-fly phase analysis broadly in real systems, it is important to understand how system effects manifest themselves in the observed phases. Recent work shows the degree of time and space variability visible in real systems that is

generally not captured in simulations [2, 116]. This variability can stem from changes in system state that can alter cache, TLB and I/O behavior, system calls or interrupts, resulting in noticeably different timing and power/performance behavior. This work discusses the repeatability of phase extraction experiments from run to run on a real system, and demonstrates the extent and type of alterations an application can experience in different experiments. It categorizes these alterations as *time shifts*, *time dilations*, and *phase mutations*, as well as transitional *glitches* and *gradients*. This work proposes a transition-based phase characterization scheme and then develops and evaluates effective methods for recognizing phases under these alterations. A step-by-step phase recognition system tests these proposed techniques on several SPEC2000 benchmarks and common desktop applications.

There are four primary contributions of this work. First, this chapter presents a taxonomy of real system effects on phase behavior based on our application measurements. Second, it proposes a transition-based phase characterization that proves to be more effective in phase detection under variability. Third, it presents a complete flow of methods to recognize phases that are resilient to variability and sampling effects. Fourth, it provides a quantitative evaluation of these techniques on a variety of benchmarks and demonstrates their effectiveness in phase recognition.

4.1 Real-System Variability

In order for a phase technique to be applicable on a real system, the phase characterizations of applications should lead to similar classifications across different runs. In most cases, we expect that the phases of two runs of the same application should be much more similar than that of two different applications. This section presents the extent of system-induced variability in real, measured application behavior and shows how this variability is reflected in the corresponding phase sequences.

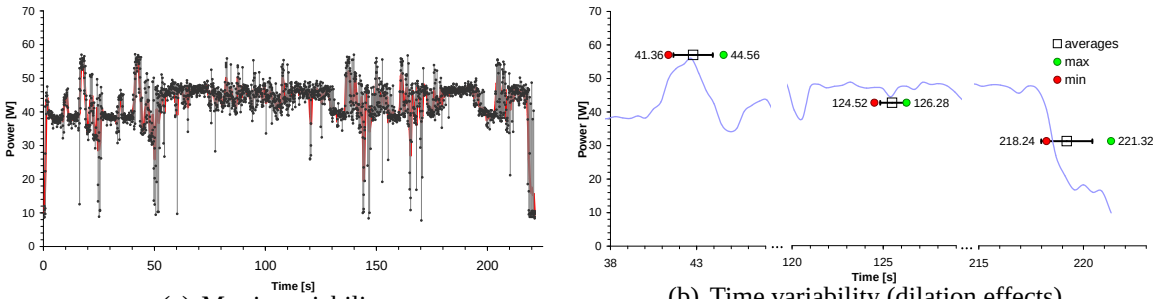
4.1.1 Variability Effects on Application Behavior

Applications exhibit two types of variability on a real system across multiple runs. First, they show slightly different instantaneous behaviors in their characteristic metrics, such as IPC, miss rates and power dissipation. Therefore, at any specific time instance, these values deviate between runs. Second, following from this, the applications show different timing behavior. This results in deviations in both total runtime and in the duration of each phase.

To quantify these two forms of variability, we collect data related to characteristic metrics and timing behavior of applications for five different runs on the same system. In all the experiments, the benchmarks are run to completion with reference datasets. After data collection, we align the traces of five runs such that all have the same first transition from idle to active phase. The first form of variability is observed in the individual measured metrics at each time sample. To show the second form of variability—different timing behavior—we specify 3 execution checkpoints for each application. We measure how long each run required to reach these points, starting from the idle-to-active transition common reference.

Figure 4.1 demonstrates an example of the observed variability for the `gcc` benchmark. The leftmost graph shows the measured power variability. Each time sample includes the average power observed across all runs as well as the range of observed power values in all five runs. The rightmost graph shows the application's time variability at the three checkpoints. Each checkpoint shows the average time, and the maximum and minimum time elapsed until the checkpoint over the five runs. It also shows the average power behavior at these regions for reference.

All benchmarks exhibit some level of both metric and time variability. The benchmarks exhibit time variabilities on the order of few seconds. This variability is a fundamental aspect of real-system behavior, and is neither a side-effect of our phase analysis methodology, nor can it be diminished with finer data sampling. Moreover, as computing systems move towards higher layers of control with hypervisors and virtual machines, managed code, and



(a) Metric variability

(b) Time variability (dilation effects)

Figure 4.1: Measured time and metric variability in the `gcc` benchmark.

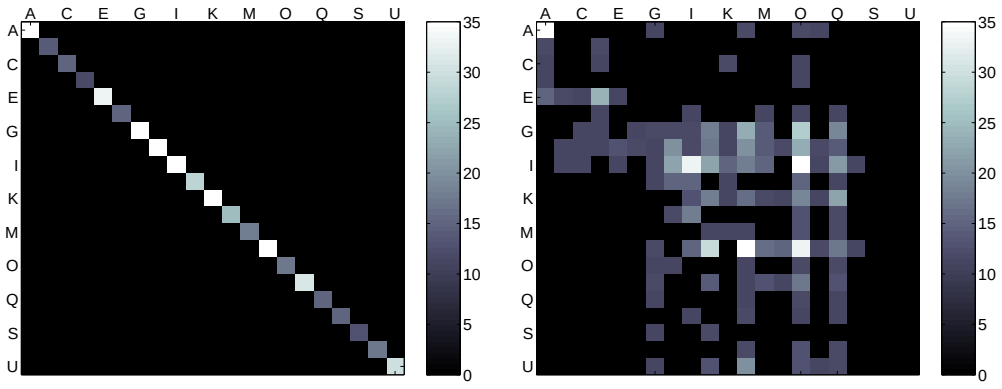
runtime systems more sources of system variability are introduced into application execution. In general, all applications result in visually similar power and performance behavior across different runs. However, some variability always exists in both characteristic metrics and runtime.

4.1.2 Variability Effects on Observed Phase Patterns

Phase analysis is inherently about gauging similarity and dissimilarity of sampled data over time. To gauge the similarity of two vector datapoints gathered by runtime PMC sampling, we use the composite similarity metric given in Equation 3.3. Our starting point in this work is a value-based phase clustering method that we had used in previous chapters. In this method, we apply a set of thresholds to this similarity metric to cluster sampled data into phases. We label encountered phases alphanumerically, starting from 'A' in each case. We call this phase representation *Value-Based Phases* (*VBP*s), where different observed phases are given different labels (phase IDs).

Although the qualitative visual behavior of a benchmark is often preserved across multiple real-system runs, differences in phase assignments occur due to inter-run variability. Even small variations can lead to the different interpretation of a phase change, thus changing the phase assignments and sequence information that follows. In addition, the durations of an application's observable phases are not identical, which also impedes exact runtime-based phase tracking techniques.

Figure 4.2 gives examples of how variability affects phases. Here, we use *joint his-*



(a) No variability (b) Actual

Figure 4.2: Joint histograms of phase distributions for two separate runs of the `gcc` benchmark. (a) An example histogram in the case of no variability (i.e., repeatable simulations). (b) The actual variability in phase behavior observed in real system runs. The letters at the top and left-hand side of the matrix plots are the phase labels.

tograms to illustrate these effects. Again with the `gcc` benchmark as an example, we use the value-based approach to split different `gcc` runs into *VBP*s. We then time-align these runs with respect to the first phase transition. The joint histogram h of two phase sequences is a matrix, where entry $h(X,Y)$ shows how many times run1 was assigned to phase X when run2 was assigned to phase Y for the same data sample. The plots show the intensity of this matrix, where brighter regions correspond to higher number of matches and darker regions show poor matches. The x and y axes on the plots show the phase labels of the two runs.

Figure 4.2(a) shows the ideal matching in the case of perfect repeatability (i.e., a simulation environment). In this case, h is only a diagonal matrix, where the diagonal values differ depending on how often each phase is encountered during application runtime. If run1 is in phase 'C' at time t, then run2 is also in phase 'C' at t. Figure 4.2(b) shows the joint histograms resulting from real-system runs of `gcc`. In these cases, the phase assignments are far from ideal. The phase assignments show a large spread, indicating significant mismatches.

In summary, the observable across-run variability seen in application power and time behavior also exists in the value-based phase characterizations of applications. This variability causes different runs of the same applications to be characterized by different phase

sequences; this conceals the underlying recurrent phase behavior.

4.1.3 Taxonomy of Phase Transformations

Figure 4.2 highlights the fact that direct, brute force comparisons of phase traces are ineffective in conveying repetitive behavior. Before discussing the proposed methods, Figure 4.3 first presents a taxonomy of the effects of variability on phases. The figure illustrates these effects and resulting phase transformations. It shows their cumulative effect on a hypothetical phase distribution, shown as the phase sequence "A, B, C, B" where the length of each labeled block indicates the duration of the corresponding phase. The first effect—*time shifts* in phase sequences—will always occur, as the processor power trace can be considered as a stream of data with no specific beginning and end. The startpoint merely depends on where we start logging the sampled power information. The second effect, *time dilations*, inevitably results from indeterministic system effects. The length of a specific task depends on the state of the machine, the available locality, number of page faults, and load of the system. *Glitches* occur when brief snippets of isolated behavior occur in some, but not all, runs. Finally, *mutations* are cases where a different phase name is seen in a run; this can be either due simply to labeling issues or it can be due to variable behavior in the application during different runs.

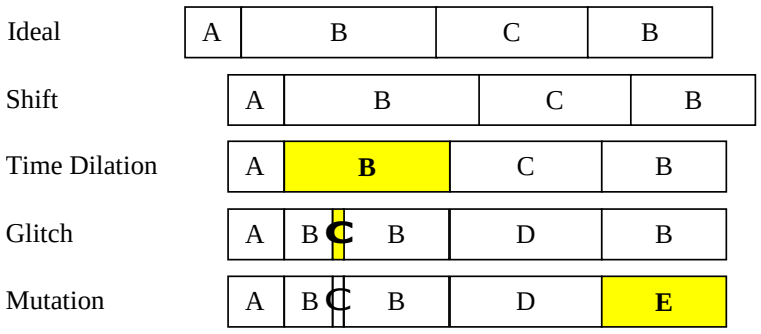


Figure 4.3: Effects of real system behavior variability on application phase distribution.

The following sections tackle each alteration in this taxonomy and propose a series of techniques for recovering the phase behavior such that the repeated runs of an application are recognized as similar.

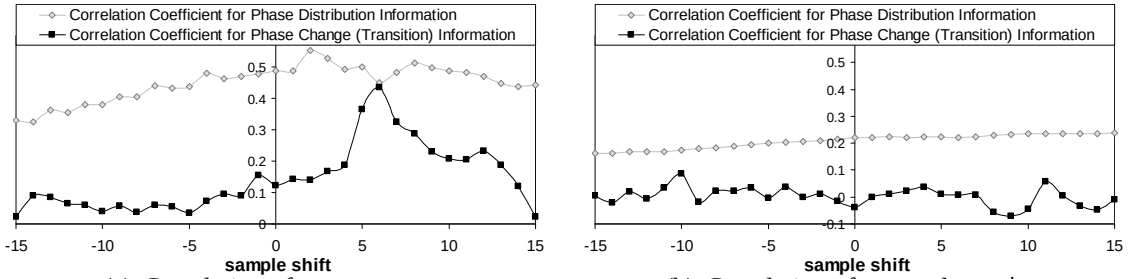
4.2 Transition-Oriented Phases

This Section proposes a representation for application phase behavior that is an alternative to the prior value-based phase (*VBP*) approach. The goal of this representation is to be more resilient to real-system variations. We suggest tracking phase transitions, instead of tracking phases themselves, and show that transitions are more effective in detecting recurrent workload behavior. We identify phase transitions at runtime by comparing the current and the previous sample vector, and by evaluating their similarity based on Equation 3.3. This transition-based representation of phase behavior, in comparison to the original *VBP* representation, is much more successful in identifying a program from its phase signature and in rejecting other application signatures based on the tested features.

One way to evaluate our claim—that tracking transitions instead of values is more successful in detecting recurrent behavior—is by computing correlations. If two phase traces vary together, they have a high correlation coefficient. Therefore, one would expect high correlations between two runs of the same application, and much lower correlations among different applications.

To perform this comparison, we enumerate *VBP* sequences with positive integers, where phase numbers are assigned to encountered different phases in increasing order. This corresponds to the original value-based representation. For the same stream, we can also represent the transition information as a binary stream, assigning 1 to phase transitions and 0 to stable regions. This is our initial proposed transition-based phase (*TBP*) representation. We call these binary sequences *Initial Transitions* (TBP_{init}).

Figure 4.4 presents the resulting correlation coefficients for two different cases. In both plots, the lighter lines plot the correlation coefficients for the original *VBP* traces. The darker lines show the results for the transition (TBP_{init}) traces. Figure 4.4(a) shows the "matching" case for two separate runs of `gcc`. Here, since we are correlating phase sequences for two runs of the same program, a good phase assignment will show a high correlation spike when the two runs are properly time-aligned. Figure 4.4(b) shows the



(a) Correlation of two gcc runs. (b) Correlation of gcc and equake.

Figure 4.4: Correlation coefficients for a range of shifts between two *different* gcc runs (a) and separate gcc and equake runs (b) (y axis shows the computed correlation coefficient values).

"mismatch" case with gcc and equake. Here, we are correlating two unrelated phase sequences, so we do not expect a high spike.

In the correlation plots, we show the results for a range of time shifts to consider the probable lag between two runs. For instance, if two traces are identical, one would expect a peak (1) in $sample\ shift = 0$. If there is only a lag of x samples between traces, the peak will move to $+x$ or $-x$.

Figure 4.4 reveals that correlating value-based phase sequences does not produce a good discrimination among benchmark signatures. In comparison, transitions provide much more useful results. Notably, we can distinctly see a peak in the TBP_{init} gcc vs. gcc case with a time-shift of 6 samples, while there is no observable peak from VBP correlations. Furthermore, correlating the transition traces of gcc and equake gives very low correlations as expected. The VBP correlations are also lower than their gcc-gcc counterpart, but transitions perform observably better, with roughly 0 correlation.

This distinguishable peak in the correlation trace for the transition-based TBP_{init} representation proves to be very useful in identifying benchmarks from their signatures. Starting with the next section, we look into these initial transitions in more detail, demonstrating how we can further improve and use this information to match application signatures in the face of real workload variability.

4.3 Techniques for Detecting Repetitive Phases with Variability

4.3.1 Removing Sampling Effects on Transitions with Glitch and Gradient Filtering

Our starting point for defining phase transitions was to say that they are sample points where the next interval's phase is different from the current phase. These transitions can be identified on-the-fly by evaluating the similarity metric in Equation 3.3 for the current and previous power vector and comparing against a similarity threshold. While Figure 4.4 illustrates that this TBP_{init} approach is already useful for phase detection, we improve on it here. In particular, we note that sampling and stability effects impede the effectiveness of transitions for representing phase behavior.

We characterize these effects as *glitches* and *gradients* (Figure 4.5). Section 4.1.3 has provided a specific example of how glitches impact phase behavior. Following the stability definitions of Dhodapkar and Smith [40], we define a *glitch* as one or more consecutive unstable sampling intervals, where the *before* and *after* of the glitch belong to the same stable phase. Because glitches are short and unstable, their single sample phase information is not likely to be useful for dynamic management techniques. A *gradient* is one or more consecutive unstable samples, where the *before* and *after* of the gradient belong to different stable phases. These regions correspond to an actual phase transition. However, some phase transitions do not happen instantaneously in a single sampling interval, but instead can actually have multiple samples along the transition gradients.

In the context of our work, glitches are false transitions and gradients are duplicated transitions. To remove these spurious effects, we propose a more intelligent transition analysis that works to filter the transitions deemed to be glitches and gradients. In *Glitch/Gradient Filtering*, extraneous transitions corresponding to glitches are discarded. Single or multi-cycle gradients en route to a new phase are converted into a single stable phase change.

Figure 4.5 shows the generic scenarios for the glitches and gradients. The upper rows

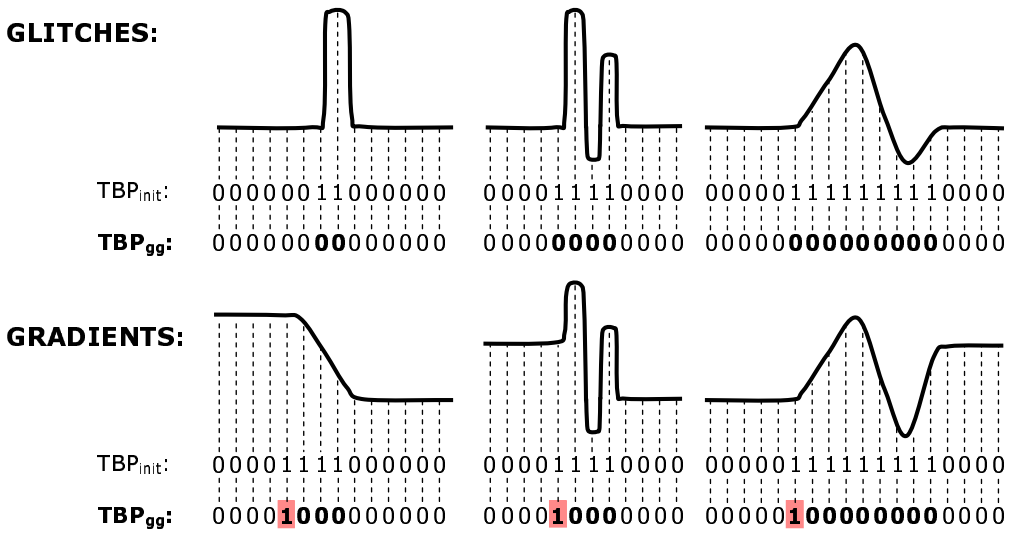


Figure 4.5: Initial transitions, TBP_{init}, with different types of glitches and gradients, and refined transitions, TBP_{gg}, after glitch/gradient filtering.

depict the initial TBP_{init} traces. ('1' denotes a transition and '0' denotes stability.) The lower rows denote the refined transition traces after we apply our glitch/gradient filtering. We refer to these transitions with glitch/gradient removal as *refined transitions* or TBP_{gg}.

Our filter implementation identifies each initial transition by monitoring the phase stream, and forms the initial binary representation TBP_{init}. From the TBP_{init} stream, we construct TBP_{gg} in the following manner. A variable-size window keeps track of the first transition in a burst of transitions. After the burst ends with a last transition to a stable region, the filter compares the stable regions before the first transition and after the last one. Then, it identifies the bursts as either glitches—if the execution regions before and after the burst are similar—or gradients—if the two regions have different characteristics. Each burst of transitions is replaced by either no transitions—if they are glitches—or a single transition—if they form a gradient. We do not allow multiple consecutive transitions in the refined TBP_{gg} signature and all gradients have a prior transition adjacent to them.

Figure 4.6 shows the application of glitch/gradient filtering to the `gcc` benchmark. The figure shows the refined transitions, as well as the regions identified as glitches and gradients, for a zoomed-in execution region. For `gcc`, the initial 212 transitions reduce to 82 once glitch/gradient filtering is applied.

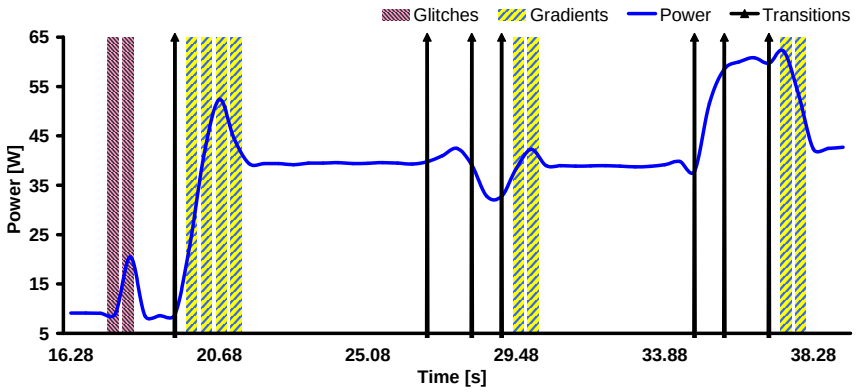


Figure 4.6: Transitions, glitches and gradients for `gcc` after glitch/gradient filtering.

4.3.2 Discerning Phase Behavior with Time Shifts

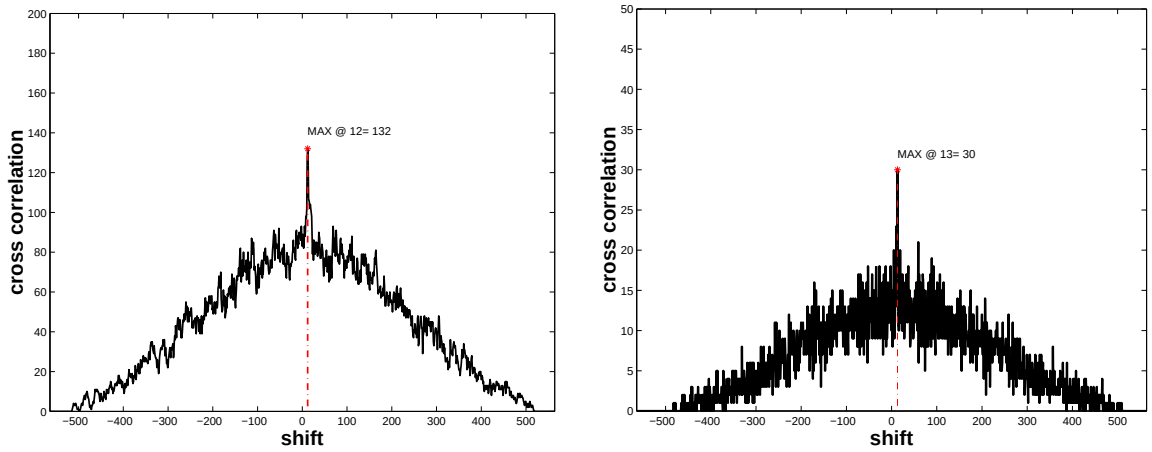
Initially we have quantitatively shown the quality of matching with transitions using computed correlation coefficients for a range of shifts. However, this method is computationally expensive and not suitable for runtime use. As the generated transition features now contain simple binary information, a simpler metric to use is cross-correlation. Correlators can be easily implemented in hardware and can be applied continuously to the incoming data stream online.

Figures 4.7 and 4.8 demonstrate again the "matching" and "mismatch" cases. In the first case, we show how well a new `gcc` run can be matched to a previous `gcc` signature. In the second case, we run `equake` and examine the severity of a false alarm. We show the results for refined (TBP_{gg}) and initial (TBP_{init}) transitions in both cases.

For the two `gcc` runs, refined transitions show a significant peak, proving a good match between the two signatures for a shift of 13 samples. For `gcc` and `equake`, the cross correlation of transitions produces no significant peak, which suggests the signatures do not match. Thus, we can see the spike behavior in case of signature match is retained with refined transitions and with the application of cross correlations.

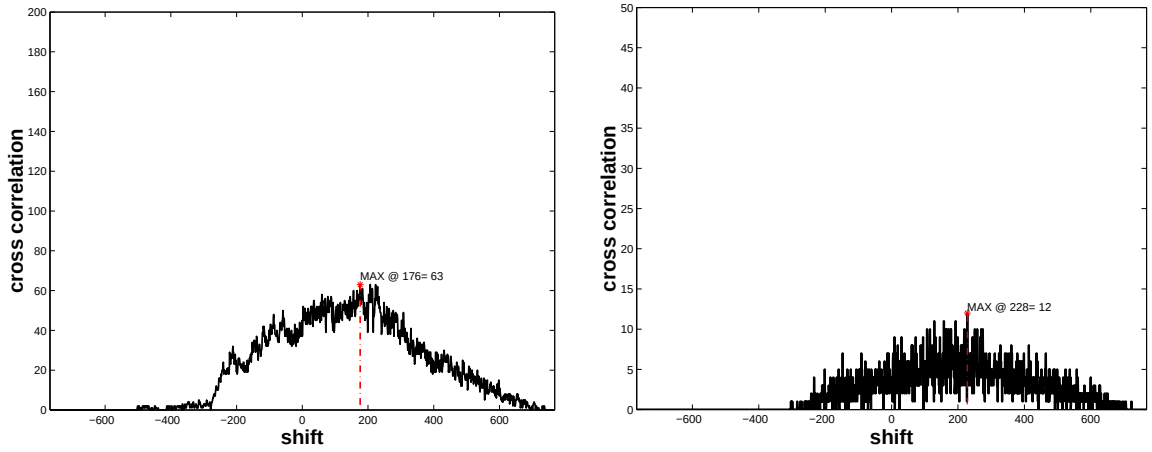
4.3.3 Handling Time Dilations with Near-Neighbor Blurring

In addition to glitches and gradients, time dilation between runs is a common problem. Recognizing the similarity of an original phase trace with a time-dilated one is a problem



(a) Cross-corr. of initial transitions. (b) Cross-corr. of refined trans-ns.

Figure 4.7: Matching of transition signatures for two `gcc` runs.



(a) Cross-corr. of initial transitions. (b) Cross-corr. of refined trans-ns.

Figure 4.8: Matching of `equake` transition signatures to `gcc`.

with similarities to many other research domains. Examples include matching a warped image in image recognition or pitch tracking in humming recognition [163]. These high-level methods can afford high complexity and they can store vast libraries of training data. In contrast, our goal is to implement an approach with simple correlators and table lookup on a small set of recent signatures.

Table 4.1 demonstrates the potential problems that time dilations pose on the transition guided phase detection scheme. Table 4.1(a) shows the high matching of processed transition information (lower trace) to a previous baseline signature (upper trace) in the absence

a. No Dilation	b. With Dilation
0 0 1 0 0 1 0 1 0 0 0 0 1 0	0 0 1 0 0 1 0 1 0 0 0 0 1 0
0 0 1 0 0 1 0 0 0 0 0 0 1 0	0 1 0 1 0 0 0 0 0 0 0 1 0
$\surd$ (match)	$\times$ (mismatch)

Table 4.1: Effect of time dilations in detecting recurrent behavior.

of time dilations. In Table 4.1(b) the lower transition trace is dilated, which shows the negative effect of time dilations on detecting recurrent behavior.

This matching problem results from considering transition information to be sharply associated with a particular deterministic sample point, while the actual transition times in each run are instead probabilistic with a modest distribution around an average. (See Figure 4.1(b) for examples.) To remedy this problem, we propose a *near-neighbor blurring* solution, which is fundamentally similar to blurring image edges for image matching. With near-neighbor blurring, we consider transitions as distributions along the time axis centered at their encountered locations. With this probabilistic approach, subtle time dilations are not penalized altogether, but instead are scaled according to their proximity to the exact location.

Tolerance: We use this metric to define the "spread" of the distribution we assume around an encountered transition time point. We define this in terms of samples. For example, a tolerance of x samples means that a transition at time sample t is considered to have a distribution in the sample range of $[t-x, t+x]$.

In our implementation, we choose a relatively primitive model, where we scale the near neighbors of transitions linearly from 1 to 0, based on the chosen sample tolerance. Further research could investigate other suitable distributions to characterize phase transitions. To apply near-neighbor blurring, the baseline refined signature (TBP_{gg}) is altered from its raw form to generate the distributions. In our evaluations we experiment with a range of tolerances from 0 to 10 samples. The second live TBP_{gg} stream, on the other hand, is not altered to avoid the runtime cost. Table 4.2 shows how the example of Table 4.1 is altered for a tolerance of 4 samples. With near-neighbor blurring, the previous mismatch due to

Baseline (refined):	0	0	1	0	0	1	0	1	0	0	0	0	1	0
Baseline (near-neighbor):	0.6	**0.8**	1.0	**0.8**	0.8	1.0	0.8	1.0	0.8	0.6	0.6	0.8	**1.0**	0.8
New run with time dilation:	0	**1**	0	**1**	0	0	0	0	0	0	0	0	**1**	0
									√ (match)					

Table 4.2: Detection with near-neighbor blurring under time dilation.

time dilations is now correctly detected as a strong match.

Applying near-neighbor blurring to TBP_{gg} results in similar cross correlations as in Figures 4.7 and 4.8. For the remainder of this chapter, we refer to TBP_{gg} augmented with near-neighbor blurring as TBP_{ggN}. In the following section, we quantify these results for our overall algorithm, using a quality metric we refer to as the *matching score*.

4.3.4 Quantifying Signature Matching with Matching Score

Matching Score: In order to quantify the success of a matching, we define the *matching score* metric, *m*, which provides a measure for the strength of matching between two signatures. Our goodness measure is the strength of the cross-correlation peak at the best alignment. Therefore, we define *m* as the ratio of best match value to the average of its closest 10 best matchings. As this value will always be greater than 1, we subtract 1 from the final value to remove this offset.

For our previous experiments with two `gcc` runs—the *matching* case—the matching scores for initial transitions TBP_{init}, refined transitions TBP_{gg} and near-neighbors TBP_{ggN} are 0.22, 0.55 and 0.32 respectively. Corresponding values for the `gcc` vs. `equake` comparison—the *mismatch* case—are 0.054, 0.16 and 0.036. Therefore, TBP_{gg} performs best for signature matching as it produces the highest matching score between the two runs of `gcc`. On the other hand, TBP_{ggN} performs significantly superior for signature rejection as it has a much lower matching score for the signatures of `gcc` and `equake`.

4.3.5 Summary of Methods

Before presenting the general quantitative results of our transition-guided recurrent phase detection method, Figure 4.9 provides a brief summary of the applied techniques. First, we

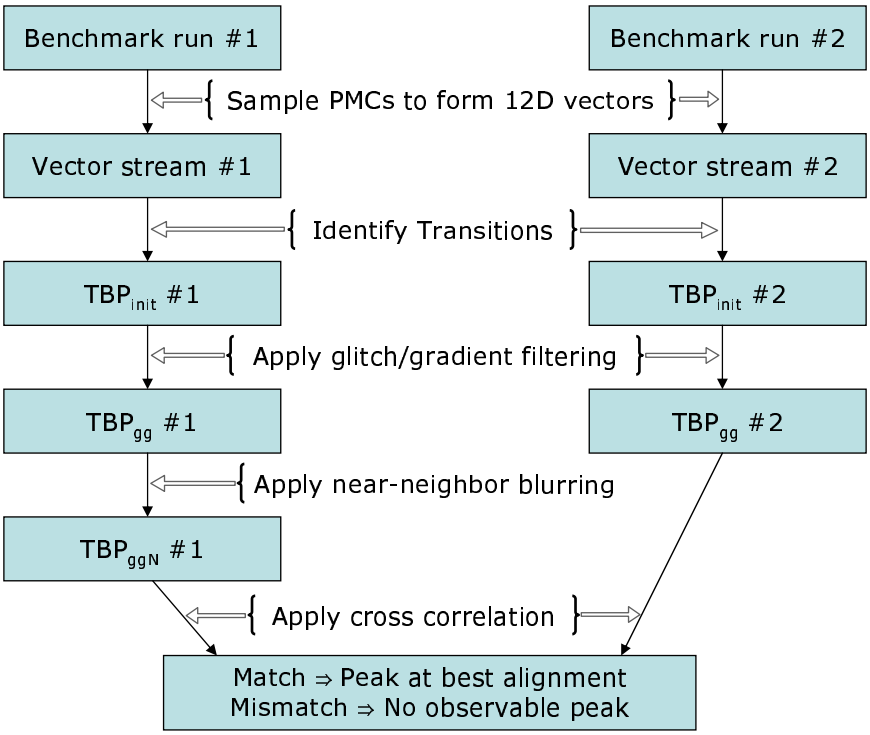


Figure 4.9: Flow of our methods.

sample PMCs during application runtime and represent benchmark execution as a stream of vectors. Then, evaluating the similarity between each current and previous vector sample, we identify initial transitions. This process converts the application execution into the binary stream TBP_{init} (Section 4.2). Next, we apply glitch/gradient filtering to TBP_{init} streams and convert them into refined transitions, TBP_{gg} (Section 4.3.1). In addition, for the first run, we apply near-neighbor blurring to TBP_{gg} and generate the baseline signature TBP_{ggN} (Section 4.3.3). After this point, any newly observed TBP_{gg} trace is cross-correlated with this baseline TBP_{ggN} to detect a signature match (Section 4.3.2). A match is determined based on the strength of an observed peak in the cross-correlation sequence, which we quantify with our matching score metric (Section 4.3.4).

4.4 Phase Detection Results

This section presents the phase detection results for a spectrum of benchmarks that include SPEC and other mainstream applications. We choose a subset of SPEC benchmarks that

	bzip2(1)	equake(2)	gap(1)	gcc(3)	gzip(1)	mcf(0)	vortex(1)	convert(7)	lame(2)
bzip2	**0.44**	0.05	0.07	0.05	0.15	0.18	0.08	0.09	0.15
equake	0.15	**0.39**	0.28	0.06	0.26	0.25	0.09	0.04	0.08
gap	0.20	0.22	**0.79**	0.07	0.10	0.33	0.04	0.05	0.12
gcc	0.05	0.04	0.05	**0.19**	0.03	0.05	0.16	0.04	0.12
gzip	0.10	0.10	0.19	0.05	**1.08**	0.16	0.10	0.03	0.07
mcf	0.18	0.18	0.23	0.04	0.16	**6.14**	0.17	0.08	0.08
vortex	0.23	0.10	0.12	0.01	0.11	0.08	**1.93**	0.03	0.05
convert	0.21	0.17	0.26	0.06	0.14	0.25	0.09	**0.22**	0.13
lame	0.12	0.11	0.12	0.04	0.13	0.20	0.06	0.02	**0.21**

Table 4.3: Matching scores for different applications. Benchmarks in each column represent the base signatures to which we apply near-neighbor blurring. The matching scores represent how well the refined phase transition signatures of the row benchmarks match to these base signatures. The superscripts in parentheses next to benchmarks show the optimum tolerance.

exhibit distinct phases in terms of power and performance metric behavior. Most of these benchmarks have high metric variability with varying transitions across different runs. Additional non-SPEC applications offer interesting phase characteristics. `Convert` is a general file conversion program that converts a large postscript file into pdf. `Convert` shows significant phases depending on the contents of the input file. We use the `lame` MP3 encoder to encode a wave file under varying quality settings. The power levels increase with finer recurrent phases at higher quality settings.

In our experiments, we run each application twice on our measurement setup. During the first run, we collect the phase transition information and apply glitch/gradient removal as they are identified. In our analysis, we consider a range of near-neighbor tolerances as well as the refined transition signatures—i.e., the outputs of glitch/gradient filtering without near-neighbor blurring. In the second run, we only generate refined transitions without any blurring.

Table 4.3 presents the matching scores for the experimented application pairs. The diagonal entries show the matching scores for the two runs of the same application—the *matching* cases. The non-diagonal entries show the matching scores between two different applications—the *mismatch* cases. The baseline signatures correspond to the columns of

Table 4.3. The transitions for the second runs are represented in the rows of the table. Therefore, the matching scores read along a column show how well a baseline signature can characterize a repeatable application phase behavior. Table 4.3 presents the matching scores corresponding to the tolerances that maximize the matching score ratio to the highest mismatch score.

As an example, for `gzip`, the baseline signature has near-neighbor blurring with a tolerance of 1 samples as indicated by the value in parentheses. Reading the `gzip` column shows that a second run of `gzip` produces a matching score of 1.08 to the baseline `gzip` signature. However, the same baseline signature produces much lower matching scores for the runs of other benchmarks, with an average of 0.13. Among these other benchmarks, `equake` is the closest match to `gzip` with a matching score of 0.26. This is significantly lower, however, than `gzip`'s matching score of 1.08. Thus, our transition-based scheme successfully detects the second run of `gzip` from its transition signature, while strongly rejecting signatures of the other benchmarks. In general, for all the benchmarks, we see the same trends. In all cases, the highest matching scores correspond to the second runs of the same application (diagonal entries), while the matching scores for different applications (non-diagonal entries) are significantly lower.

Most benchmarks achieve their best matching scores with a few levels of tolerance (1-3 samples) due to their small dilation magnitudes. The only exception is `convert` with an optimal tolerance of 7. As `convert` has only 17 transitions in its signature, each extra hit in the spread has greater relative impact, thus favoring higher tolerances. The zero tolerance case is equivalent to the TBP_{gg} signatures, without any blurring. Only for `mcf` is the best matching condition achieved by TBP_{gg}.

In general, the outcomes of our detection method are very useful. We can detect specific recurrent phase sequences under different kinds of variability, with a moderately simple technique that can be implemented at runtime with negligible overhead. In most cases, considering transitions as distributions via near-neighbor blurring improves our results fur-

ther, with the choice of small tolerance levels.

4.4.1 Receiver Operating Characteristics

As with any detection scheme, our matching scores are also prone to *misses* and *false alarms* for a particular *detection threshold*. That is, for all applications, a matching score above this single detection threshold is categorized as a 'hit'. For instance, for the runtime detection scheme of Table 4.3, if we use a threshold of 0.19, we would be able to identify all the hits. However, out of the 72 possible mismatches, we would have also detected 11 of them as hits. Thus, this scenario would have a hit detection probability of 1. However, this would also incur a *false alarm* probability of $11/72 \approx 15.3\%$. If we increase the detection threshold, the probability of false alarms diminish, while this, in turn means some hits are *missed*. It is common practice in pattern classification to demonstrate this effect in terms of *Receiver Operator Characteristic* (ROC) curves. The detection function is graphed as the hit probability as a function of the false alarm rate [42]. Figure 4.10 shows the ROC curves for our detection technique. To present the probabilities, the axes are shown from 0 to 1. However, for absolute measures, 1 on the *hit* axis represents 9 detected hits for the 9 benchmarks; and 1 in the *false alarm* axis represents 72 falsely detected hits for the 72 possible different benchmark combinations. The intermediate values are linearly scaled for both axes. Each ROC curve in the figure corresponds to a TBP_{ggN} with a specific tolerance. For each curve we first compute the matching score matrix, similar to Table 4.3, across all the benchmarks for the current tolerance value. We then compute the hit and false alarm probabilities for several detection thresholds with step sizes of 0.05 for the whole matching score range 0-6.15.

In the ROC curves, we see that our detection scheme achieves high hit probabilities with small false alarm rates. Among the applied tolerance levels, TBP_{ggN} with a sample tolerance of 1 performs best, which is followed by tolerances of 2 and 3. The zero tolerance case, which corresponds to TBP_{gg}, with no distribution, performs distinctly worse for signal rejection. This proves the effectiveness of our near-neighbor blurring technique. Our best

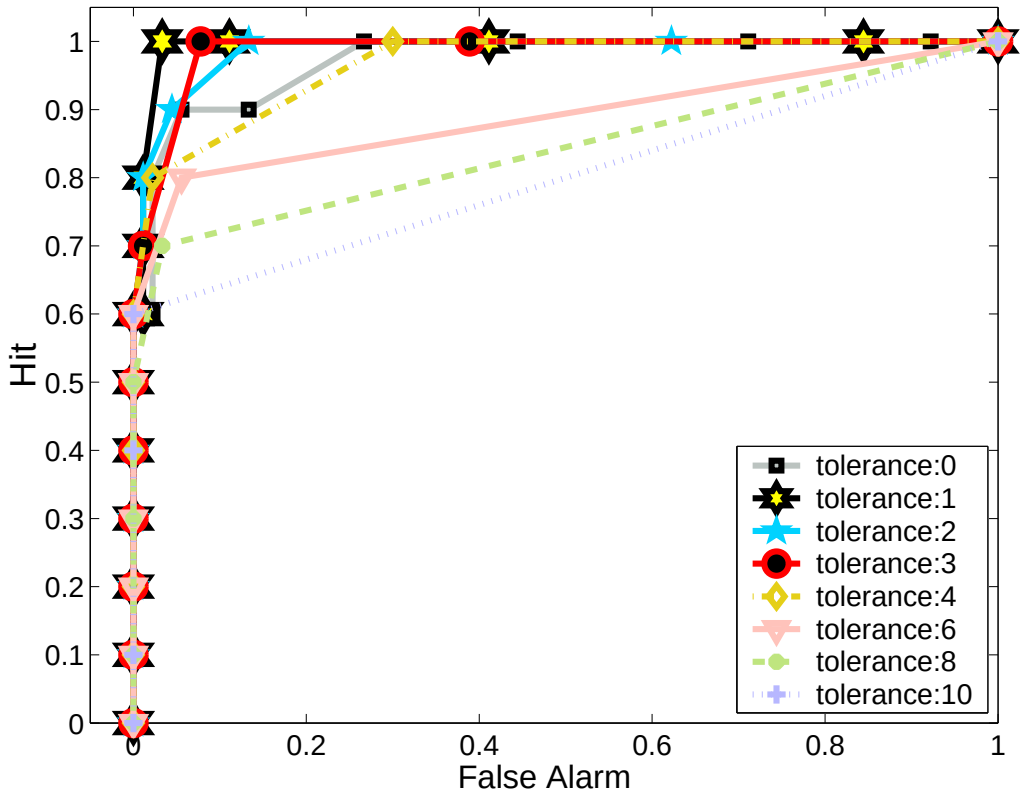


Figure 4.10: Receiver Operating Characteristic (ROC) curves for TBP_{ggN} with 0-10 range of tolerances.

detection method achieves 100% hit detection with less than 5% false alarms.

4.4.2 Comparison of Transition-Guided Approach to Value-Based Phases

Figure 4.11 provides a final comparison of detection success between the original value-based phase representation (VBPs), refined transitions (TBP_{gg}), and final near-neighbor blurred transitions (TBP_{ggN}). For this comparison, the figure shows the ratio of the matching score between two runs of the same application (matching case) to the highest matching score among all the different applications (worst mismatch case) for the same application (i.e., `vortex` for `bzip2`). Consequently, this quantifies how well each representation detects a matching signature, while rejecting other unmatching signatures. The figure shows the individual results and the average for the experimented benchmarks. Below the "break-even" line at $ratio = 1$, a technique finds another application signature as a "better match" for the current benchmark. Ratios significantly higher than 1 represent accurate signature

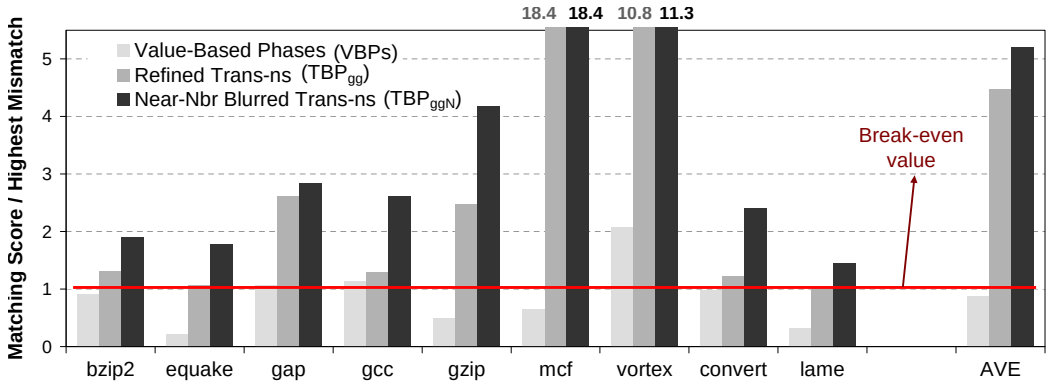


Figure 4.11: Improvement in phase detection efficiency with transition-based approach.

detection.

In all cases, transition based methods perform much better than *VBP*s. In all cases except `mcf`, TBP_{ggN} shows significant improvement over TBP_{gg}. For `mcf`, both transition techniques perform equally well, as the best tolerance for `mcf` is 0. On average, our transition based, near-neighbor blurring technique provides a 6-fold improvement in recurrent behavior detection under variability, over the original value-based phases.

4.5 Related Work

Prior phase detection work operates at various domains and granularities using a variety of characteristic metrics to track phases. Dhodapkar and Smith [41], Sherwood et al. [152, 153], Lau et al. [109], Iyer and Marculescu [90], and Huang et al. [75] track the executed code characteristics such as basic blocks and subroutine IDs to detect phases. All these works are based on cycle-level simulations and, although useful for guiding representative simulation and architectural studies, they do not reflect the available real-system variability.

Some recent research also looks at executed code characteristics with real-system experiments. Patil et al. [137] and Lau et al. [108] use dynamic instrumentation to identify basic block based phases. Hu et al. [72] discuss compile time instrumentation to find basic block phases at runtime for power studies. Annavaram et al. [6] apply program counter sampling to find similar execution paths and investigate performance behavior similarity in these regions. These approaches also account for real-system variability. However, they do

not consider detection of recurrent phase sequence signatures.

Another line of research explores performance behavior for phase tracking, using metrics such as IPC and memory references. Cook et al. [35] identify execution phases based on deterministic simulations. Todi [169] and Weissel and Bellosa [176] use runtime performance counter information on different platforms for workload characterization and reactive dynamic optimizations. Chang et al. [29] use a power profiling method triggered by consumed energy quanta to attribute software energy to processes. Duesterwald et al. [44] also use performance counters to predict metrics such as IPC and L1 misses. Their work uses previous short-term sample history to predict behavior in the next sampling period. These run-time techniques also analyze application behavior under variability, but they do not aim to detect large-scale recurrent phase sequences. Shen et al. [150] also look at detecting recurrent phases by observing reuse distance patterns. They use detailed program profiling and instrumentation to detect phases, while our work tries to identify phase transitions from runtime power vectors.

4.6 Summary

This chapter presented a novel approach to phase behavior detection that is resilient to real-system variability effects. Based on real-system measurements, we categorized the variability effects and provided methods to address these distortions of phase behavior. We proposed a *transition-oriented* phase representation and demonstrated its robustness against phase mutations and shifts with correlations. We developed *glitch/gradient filtering* to refine phase transitions from sampling effects and used *near-neighbor blurring* to handle observed moderate time dilations. By carefully discriminating these variability effects and application specific phase information, we were able to detect recurrent phase sequences prone to several real world transformations.

Overall, the results of this chapter show that this fully-automatable flow of techniques can detect recurrent application phase signatures with good accuracy for SPEC and other

benchmarks. Our best detection scheme, near-neighbor blurring with a tolerance of 1 sample, was able to detect all signatures with a false alarm probability less than 5%. In comparison to original value-based phase representation, transitions with near-neighbor blurring performed on average 6X better in detecting recurrent application signatures, while rejecting unmatching signatures.

This research has importance both in characterizing real-system variability effects and in addressing phase detection despite this variability. As phase-adaptive management techniques become available in the emerging architectures and systems, such variation-resilient phase detection techniques are essential for real-system dynamic management.

Chapter 5

Runtime Phase Tracking and Phase-Driven Dynamic Management

The increasing complexity and power demand of processors mandate aggressive dynamic power management techniques that can adaptively tune processor execution to the needs of running applications. As the previous chapters have discussed, these techniques benefit extensively from application phase information that can pinpoint execution regions with different characteristics. Recognizing these phases on-the-fly enables various dynamic optimizations such as hardware reconfigurations, dynamic voltage and frequency scaling (DVFS), thermal management and dynamic hotcode optimizations [3, 11, 14, 41, 76, 90, 155, 179]. In recent years, studies have demonstrated various approaches to characterize and detect application phase behavior [6, 41, 87, 90, 176]. Some of these studies have also discussed methods to predict future application execution characteristics [44, 89, 150, 153, 186]. However, to be able to utilize phase information effectively on a running system, a general dynamic phase prediction framework must seamlessly operate on-the-fly during workload execution. Moreover, it is essential to provide a useful and clear binding between application phase monitoring and prediction, and dynamic management opportunities, especially on real-system implementations.

This chapter brings together the phase monitoring and detection techniques discussed in previous chapters and extends these to a a complete runtime dynamic power management

framework that is controlled by phase monitoring and prediction. Overall, the runtime power and performance monitoring techniques discussed in Chapter 2 provide the foundation of the deployed real-system infrastructure. The phase analysis methods of Chapter 3 are utilized as the generic baseline phase monitoring technique. The impact of system induced variations in workload behavior that are discussed in Chapter 4 guide the phase definitions and prediction methodology presented in this chapter.

In particular, this chapter describes a fully-automated, dynamic phase prediction infrastructure deployed on a running mobile platform. It shows that a *Global Phase History Table* (GPHT) predictor, inspired by a common branch predictor technique, achieves superior prediction accuracies compared to other approaches. The GPHT predictor performs accurate on-the-fly phase predictions for running applications without any offline profiling or any static or dynamic modifications to application execution flow, and with negligible overhead. This runtime phase prediction method can effectively guide dynamic, on-the-fly processor power management using DVFS as the underlying example dynamic power management technique [55]. Our dynamic phase predictor efficiently cooperates with a DVFS interface to adjust processor execution on-the-fly for improved power/performance efficiency. This GPHT-based dynamic power management improves the energy-delay product (EDP) in our deployed experimental system by more than 15%. This methodology can be used with different phase definitions that can be aimed at serving different purposes such as bounding execution with performance degradation limits. We evaluate our methods on the SPEC CPU2000 benchmark suite, with runtime monitoring using performance monitoring counters (PMCs), and real power measurements with a data acquisition (DAQ) unit.

There are three primary contributions of this chapter. First, it presents and evaluates a live, runtime phase prediction methodology that can seamlessly operate on a real system with no observable overheads. Second, it describes a complete real-system implementation on a deployed system. This implementation can autonomously function during native operation of the processor, without any profiling or static instrumentation of applications.

Third, it demonstrates the application of the phase prediction infrastructure to dynamic power management using DVFS as an example technique. Although this work discusses specific phase definitions and power management techniques, our runtime phase prediction is a general framework. It can be applied to any feasible definition of application phases and to other dynamic management techniques, such as dynamic thermal management or bounding power consumption.

5.1 Phases for Dynamic Management

The key motivation of this work is to develop a phase prediction technique that can be accurately applied at runtime application execution to guide dynamic power management. This section explains our phase classification methodology, which is later used in the evaluations. The fundamental purpose of phase characterization is to classify application execution into similar regions of operation. This classification can be done via various features, depending on the ease of monitoring and the goal of the applied phase analysis. Similarly, how the observed features are classified into different phases depends on the target application. The previous chapters have defined phases that represent different power characteristics of workloads. While these phases are useful for general power characterization, they are not directly tied to a specific management action. On the other hand, this chapter considers DVFS as the underlying management application and defines phases that reflect the potential of different execution regions to be improved by DVFS.

We rely on hardware performance monitoring counters (PMCs) to track application behavior. These counters can be configured to monitor execution without disrupting execution flow. While Chapter 3 demonstrated that control flow information is also useful for tracking application phases, fine-grain runtime monitoring of control flow incurs significant overheads for system-level management. Therefore, we rely only on PMC events to monitor application characteristics. For system-level dynamic management, we define relatively coarse grained phases, on the order of millions of instructions. This guarantees

that monitoring of application behavior—and dynamic management responses—does not lead to any observable overheads.

The phase classifications are constrained by two factors. First, our experimental platform, described in greater detail in Section 5.5, supports simultaneous monitoring of 2 PMCs. Therefore, our classifications of application behavior can only be based on two configured counters. This leads to somewhat more restrictive phase definitions than prior chapters, where the experimental platform supported monitoring of 18 simultaneous events. The primary reason for this different experimental platform is that the previously used experimental systems do not support DVFS. Second, we monitor PMCs from within a performance monitoring interrupt (PMI) routine. Therefore, we need a simple classification method to avoid violating interrupt timing constraints as well as to have negligible performance overheads. In addition, one of the counters has to be dedicated to monitor *micro-ops (Uops) retired*, to trigger the PMI at specified instruction granularities. This instruction-based phase tracking is motivated by the variability observations presented in Chapter 4. While this approach induces significant restrictions to performance monitoring, it diminishes the impact of timing variations in observed behavior.

We draw from prior work for our choice of monitored PMC events. Wu et al. [179] make use of event counter information to assign application routines to different DVFS settings under a dynamic instrumentation framework [121]. They define the ratio of *memory bus transactions* to *Uops retired* as the measure of the "memory-boundedness" of an execution region, and use the ratio of *Uops retired* to *instructions retired* as a proxy to represent available "concurrent execution" in the same region. These two metrics then determine the available "CPU slack" in the application, which guides different DVFS settings. For our experiments, we configure the remaining independent counter to track memory bus transactions. Thus, the ratio of the memory bus transactions to our Uop granularity represents the memory-boundedness of each observed phase. This measure is referred to as "Mem/Uop" in this chapter.

In addition to Mem/Uop, the two counters, together with the time stamp counter (TSC), also enable simultaneous monitoring of Uops per cycle (UPC), which can provide additional information on application behavior. These two metrics have already been used cooperatively in other previous studies to guide dynamic power management [176]. However, for phase prediction to perform reliably, dynamic management actions should not alter the workload characteristics they are tracking. Otherwise, recorded phase histories will become obsolete at each change in the management actions. As Section 5.4 demonstrates, while Mem/Uop behavior is virtually invariant to the responses of our dynamic management technique, UPC can fluctuate strongly. Therefore, for a simple, yet robust phase classification that is largely invariant under dynamic power management, we use Mem/Uop to define application phases.

We classify Mem/Uop into different phases by observing how different Mem/Uop rates are assigned to different DVFS settings in prior work [179]. That work examines memory access rates and concurrency of different applications on a similar experimental platform. Then, it calculates the DVFS settings for different application regions based on a performance loss formulation. For our phase definitions, we convert these measures to Mem/Uop rates and available concurrency ranges for each DVFS setting. As we do not have the concurrency measure available for our runtime monitoring and prediction, we base our phase classifications on the derived Mem/Uop ranges for the common lowest observed concurrency—i.e., $Uops\ retired/instructions\ retired \approx 1$. Based on this classification, Table 5.1 defines 6 phase categories. Conceptually, Category 1 corresponds to a highly CPU-bound execution pattern that should be run as fast as possible, and Category 6 corresponds to a highly memory-bound phase, where the application can be significantly slowed down to exploit available slack.

Mem/Uop	Phase #
< 0.005	1 (highly cpu-bound)
[0.005,0.010)	2
[0.010,0.015)	3
[0.015,0.020)	4
[0.020,0.030)	5
> 0.030	6 (highly memory-bound)

Table 5.1: Definition of phases based on Mem/Uop rates.

5.2 Predictability and Power Saving Potential Characteristics of Workloads

To assess the quality of a phase prediction scheme, it is imperative to first understand the predictability characteristics of different applications. Consider, for example, a very stable application with very few changes in its phase behavior. Here, a simple predictor that assumes the last observed behavior will continue, will be highly accurate. However, on benchmarks with high variability, where the observed phases change rapidly, such an approach will experience many mispredictions. Therefore, before evaluating our phase prediction method, this section discusses the intrinsic predictability of different benchmarks.

Figure 5.1 shows the characteristics of different benchmarks in two dimensions. The *y* dimension shows the variability of benchmarks, based on the observed variation in Mem/Uop. We represent this as the percentage of time Mem/Uop changes more than 0.005 between two samples for a 100 million instruction sampling granularity. Thus, this dimension shows how "unstable" the benchmark is. Benchmarks higher along the *y* axis represent cases with temporally varying behavior, which cannot be predicted in a straightforward manner simply by assuming the benchmark will preserve its last observed behavior. On the other hand, benchmarks close to the x axis show almost completely "flat" execution behavior, where the application rarely changes its execution properties. In these cases, simply assuming the previous observed characteristics will prevail performs as well as any other method. In

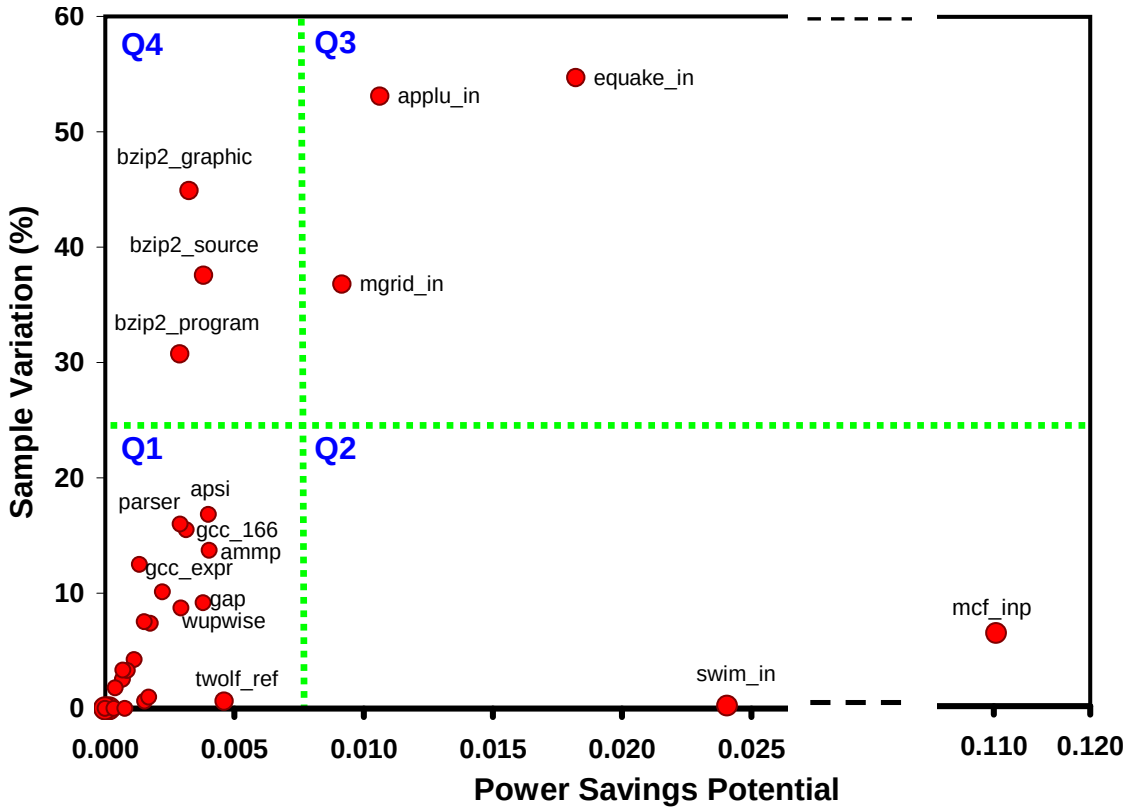


Figure 5.1: Benchmark categories based on stability (in terms of Mem/Uop variation between samples) and power saving potential (based on average Mem/Uop rates).

addition to these variability characteristics, the *x* dimension of the figure shows the average Mem/Uop rate for our applications. This shows how much potential exists to slow down the CPU frequency for each application. Thus, benchmarks further to the right exhibit higher power savings potential. There are a cluster of applications that lie very close to the origin, showing small variations and power saving opportunities. We do not label these in the figure to avoid cluttering the image.

Based on these observed properties we categorize the benchmarks into four quadrants. *Q*1 benchmarks, which include many of the SPEC applications, are very stable and show little power saving opportunities. *Q*2 benchmarks show higher power saving potential and little variability. These two categories are easily predictable with simple phase predictors. *Q*3 benchmarks `applu`, `equake` and `mgrid` are the most interesting applications for our research. These have both highly varying phase behavior and high power saving poten-

tial. $Q4$ benchmarks also show high variability, but show relatively smaller power saving opportunities. Because of their high variability, $Q3$ and $Q4$ applications are not expected to perform well under a simple phase prediction strategy that assumes that the next phase behavior will match the previously observed one.

5.3 Phase Prediction

This section first discusses different prediction options and describes our chosen technique. Afterwards, it presents our evaluations for phase prediction accuracy. For a phase prediction technique that can perform well on all corners of benchmark behavior, we propose a *Global Phase History Table* (GPHT) predictor. There exist other prior history-based predictors that also seek to estimate application performance characteristics [44, 89]. However, predictors that simply rely on the statistical measures of past behavior, such as averages or population counts, cannot perform well for highly variable benchmarks. To demonstrate this comparatively, we also consider some of the simple statistical predictors in our evaluations.

The simplest statistical predictor is the *last value* predictor. In this predictor, the next sample behavior of an application is assumed to be identical to its last seen behavior. In this case, predicted phase in the next interval can be expressed as $Phase[t+1] = Phase[t]$. This predictor can be extended to encompass longer past histories by considering a *fixed history window* predictor, where the predictions are based on the last *window size* observations. In this case, the next phase prediction can be phrased as $Phase[t+1] = f(Phase[t], Phase[t-1], ..., Phase[t-(winsize-1)])$. The function $f()$ can be a simple averaging function, an exponential moving average or a selector based on population counts. Another approach, similar to fixed history window is a *variable history window* predictor. In this case, the history can be shrunk in case of a phase transition, where previous history becomes obsolete for the subsequent phase predictions. The next phase prediction method for this predictor is similar to the fixed history window predictor. However, the window size is also a varying

parameter based on the last account of observed phase transitions.

5.3.1 Global Phase History Table Predictor

In contrast with the statistical predictors, our *Global Phase History Table* (GPHT) predictor observes the *patterns* from previous samples to deduce the next phase behavior. In such an approach, it relies on the well-known repetitive execution behavior of applications. Structurally, the GPHT predictor, depicted in Figure 5.2, is similar to a fully-associative global branch history predictor [181]. Unlike hardware branch predictors, however, the GPHT is a software technique, implemented in the operating system for high-level, dynamic phase prediction.

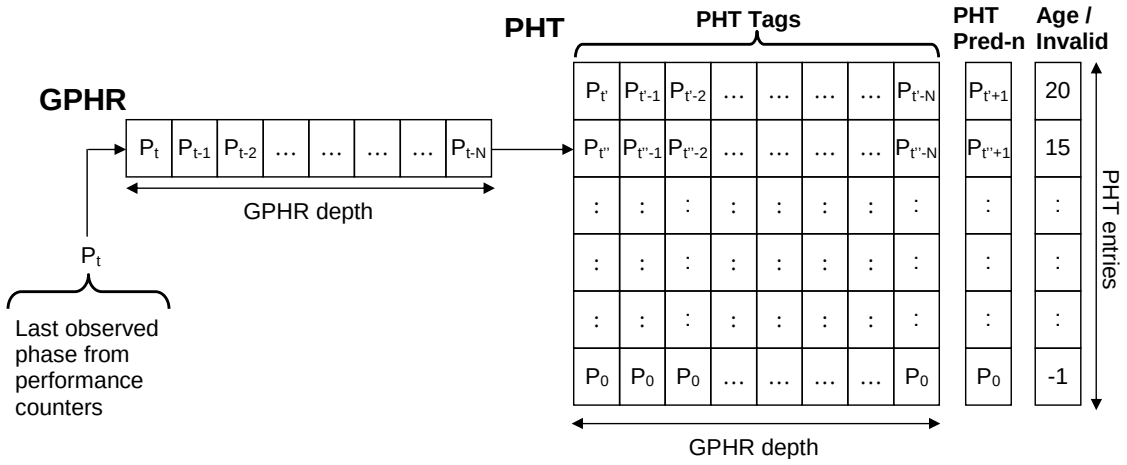


Figure 5.2: GPHT predictor structure.

Similar to a global branch predictor, a GPHT predictor consists of a global shift register, called the *Global Phase History Register* (GPHR), that tracks the last few observed phases. The length of the history is specified by *GPHR depth*. At each sampling period, the GPHR is updated with the last seen phase, as observed from the PMCs. This updated GPHR content is used to index a *Pattern History Table* (PHT). The PHT holds several previously observed phase patterns, with their corresponding "next phase" predictions based on previous experience. These phase predictions are shown as the *PHT Pred-n* vector in the PHT. The GPHR index is associatively compared to the stored valid PHT tags. If a match is found, the corresponding PHT prediction is used as the final prediction. A per-entry *Age*

/ *Invalid* value tracks the ages of different PHT tags and allows for a least recently used (LRU) replacement policy when the PHT is full. A -1 entry denotes the corresponding tag contents and prediction are not valid. The number of entries in the PHT is specified by *PHT entries*. In the case of a *mismatch* between the GPHR and the PHT tags, the last observed phase, stored in GPHR[0], is predicted as the next phase. After a mismatch, the current GPHR contents are added to the PHT by either replacing the oldest entry or by occupying an available invalid entry. In the case of a match, a PHT prediction entry is updated in the next sampling period based on the actual observed phase for the corresponding tag.

By observing the phase patterns in application execution, the GPHT predictor can perform reliable predictions even for highly variable benchmarks. Inevitably, for a hypothetical application with no visibly recurrent behavior, no existing predictor can function accurately. In such cases there is no matching pattern in the PHT and we revert to a last value predictor, thus guaranteeing to meet the accuracy of previous methods under the worst case scenarios. Most applications exhibit some amount of repetitive patterns, however, due to the common loop-oriented and procedural execution style.

Figure 5.3 gives an example of how the GPHT accurately captures varying application behavior with the `applu` benchmark. `Applu` shows highly varying behavior with distinctive repetitive phases throughout its execution. The figure shows the variation in Mem/Uop for `applu` and its corresponding phases from a sample execution region that is chosen to reflect the repetitive execution characteristics of `applu`. It shows the performed phase predictions with both the GPHT and last value predictors. While we have experimented with other statistical predictors (depicted in Figure 5.4), the figure only shows the last value predictor as the best performing statistical predictor for this application. The GPHT has a GPHR depth of 8 and 1024 PHT entries. This example shows that even for this highly variable application, GPHT predictions almost perfectly match the actual observed phases. In contrast, a last value prediction method mispredicts more than one-third of the phases due to `applu`'s rapidly varying phases. Figure 5.3 highlights two regions, showing the

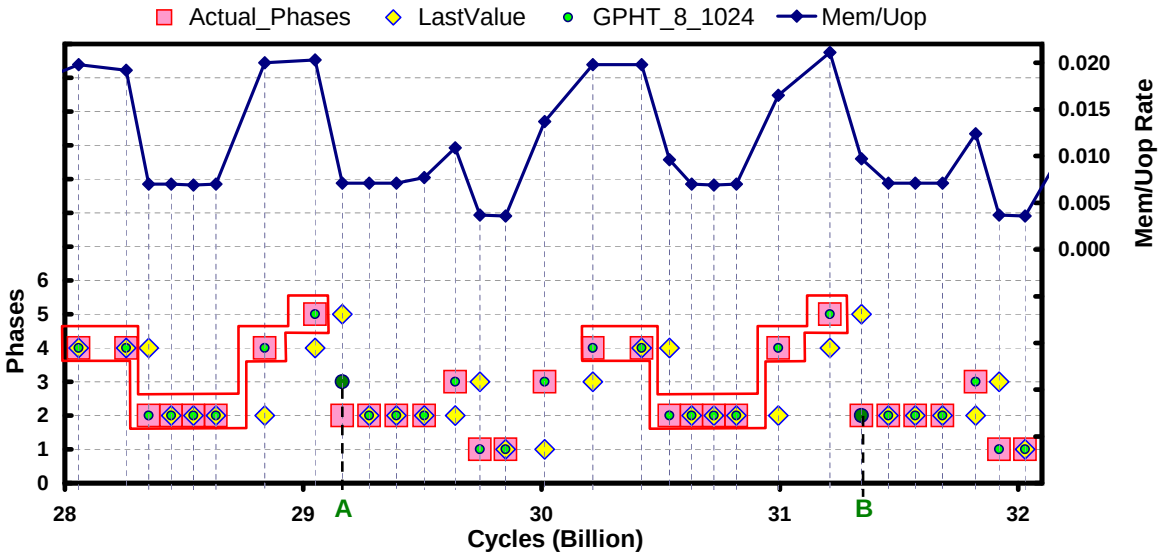


Figure 5.3: Actual and predicted phases for the `applu` benchmark.

repetitive phase behavior and how GPHT can easily capture this behavior. In addition, it shows two distinct cases, where GPHT first mispredicts the next phase at point labeled *"A"*, and later can correctly predict similar behavior at point *"B"* by learning from the previous pattern history. This example shows the clear strength of pattern-based phase prediction with GPHT over statistical approaches.

5.3.2 Phase Prediction Results

Figure 5.4 shows the achieved prediction accuracies on our applications for four prediction methods. In particular, it shows *(i)* last value prediction, *(ii)* fixed window prediction with window sizes of 8 and 128, *(iii)* variable window with a 128-entry window and phase transition thresholds of 0.005 and 0.030, and *(iv)* GPHT with a GPHR depth of 8 and 1024 PHT entries. The thresholds for the variable window predictors are chosen to achieve effective window sizes that fall between last value and fixed window predictors. We have also experimented with different PHT and GPHR sizes. The effect of PHT size on the prediction accuracy is discussed later in this section (Figure 5.6). For the GPHR depth, the near neighborhood of eight entries performs similarly to the presented results. However, GPHR sizes larger than 16 or smaller than 4 degrade accuracy. In Figure 5.4, the benchmarks are

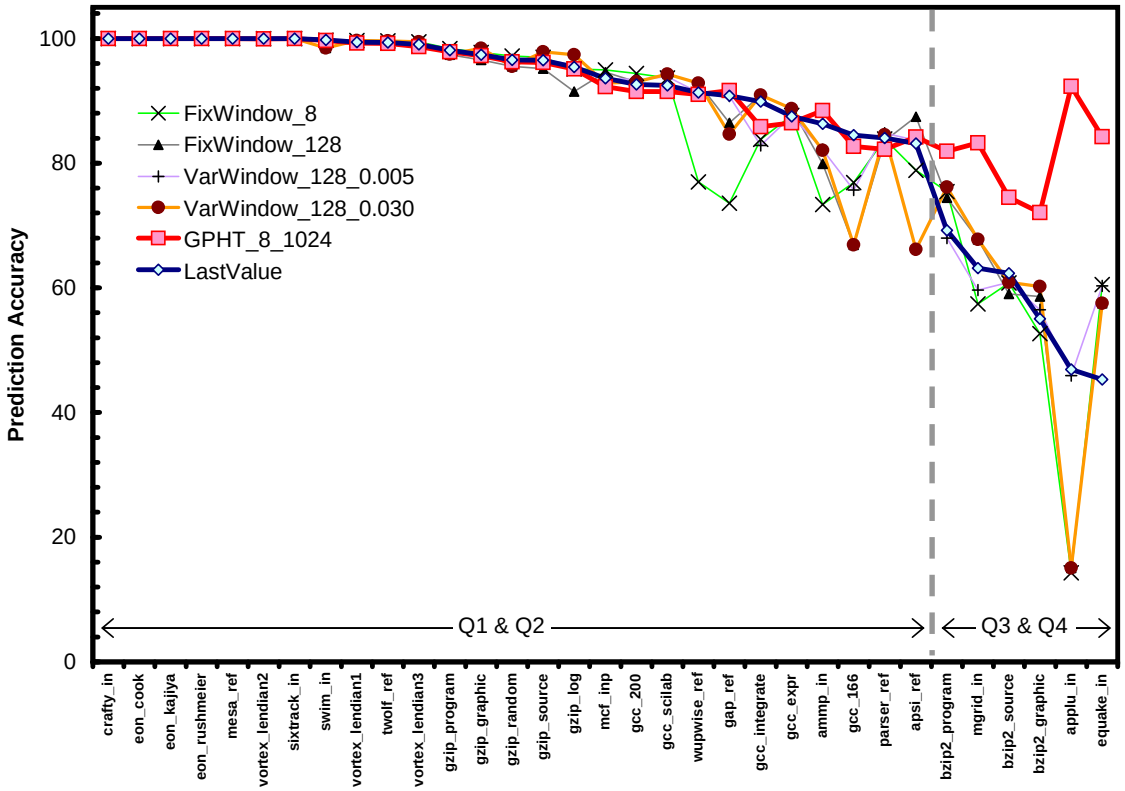


Figure 5.4: Phase prediction accuracies for experimented prediction techniques.

sorted in the order of decreasing prediction accuracy with last value prediction.

For most of the *Q*1 and *Q*2 benchmarks, almost all approaches perform very well, achieving prediction accuracies above 80%. For these mostly stable applications, last value and GPHT perform almost equivalently. However, the benefits of GPHT are immediately observed with the last 6 benchmarks, which constitute the *Q*3 and *Q*4 applications. In these more variable benchmarks, the last value, fixed window and variable window approaches experience significant drops in prediction accuracies, while GPHT can still sustain higher prediction accuracies by observing repetitive phase patterns. For `applu`, the last value predictor—the best non-GPHT predictor for this application—results in more than 53% mispredictions. In comparison, GPHT achieves less than 8% mispredictions, which improves phase mispredictions by more than 6*X*. On average, for the *Q*3 and *Q*4 benchmarks, our GPHT predictor leads to 2.4*X* fewer mispredictions than the other predictors.

The detailed results of Figure 5.4 are for the initial phase definitions described in Ta-

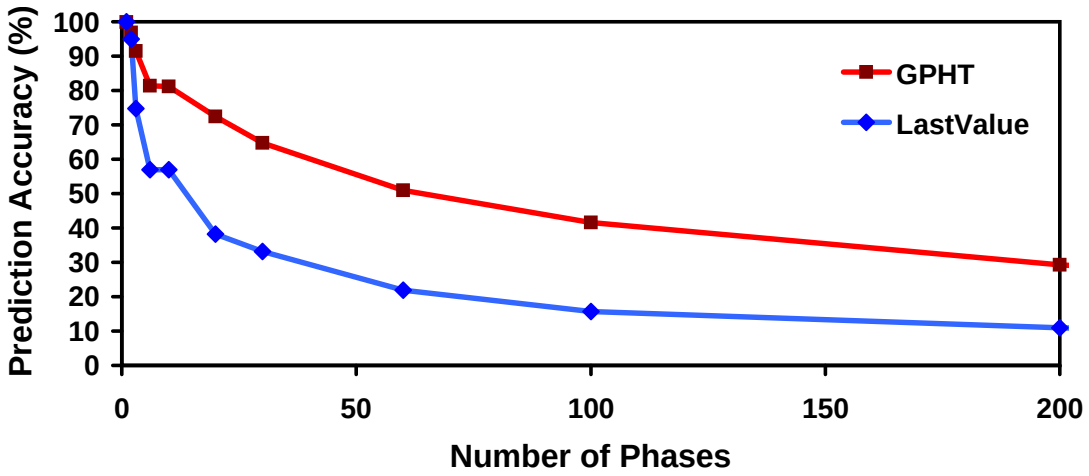


Figure 5.5: Prediction accuracies for different phase definitions and granularities.

ble 5.1. However, it is also important to verify that our phase prediction methodology is a consistent general framework regardless of the chosen phase definitions. Therefore, we experiment with a wide range of phase definitions with varying number of phases as well as with phase boundaries different from the ones in Table 5.1. Figure 5.5 shows the resulting prediction accuracies, summarized as averages across the experimented applications. The benchmarks include all SPEC applications excluding `crafty`, `eon`, `mesa`, `vortex`, `sixtrack`, `swim` and `twolf`. The excluded applications show no visible variations with less than 1% mispredictions with all predictors. This chart only compares the GPHT predictor with the last value predictor. Here, the results show that our GPHT predictor is consistently more accurate for all practical phase definitions. Both predictors start with a 100% prediction accuracy for a singleton phase and trend towards 0 with increasing phase granularities. In all the intermediate regions, the accuracy of GPHT predictor is significantly higher than that of the last value predictor.

This evaluation clearly demonstrates that our proposed GPHT predictor performs effectively in all quadrants of the benchmark categories and for a large set of phase definitions. The remainder of this work builds our dynamic power management framework upon this phase prediction methodology. However, for different implementations, storing and associatively searching through a 1024-entry PHT may be undesirable or unnecessary. There-

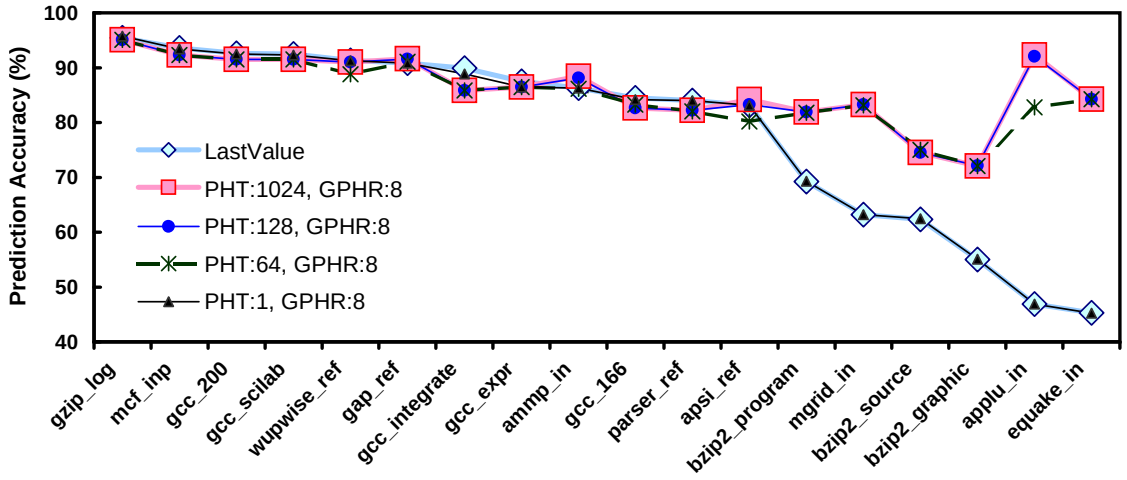


Figure 5.6: GPHT prediction accuracy for different number of PHT entries.

fore, Figure 5.6 shows how GPHT prediction accuracy changes with different numbers of PHT entries. As the figure shows, down to 128 entries, the GPHT predictor performs almost identically to the 1024 entry predictor. However, observable degradations in accuracy are seen with a 64 entry PHT. As the number of PHT entries is reduced to 1, the accuracy of the GPHT predictor converges to last value, due to almost 100% tag mismatches. In these cases, the next phase is continuously predicted as the last encountered phase from GPHR[0]. This shows that a 128-entry PHT is sufficient for our GPHT implementation. In our deployed real system, described in the following sections, we use this configuration for our final GPHT predictor implementation.

5.4 Dependence of Phases to Dynamic Management Actions

For the phase prediction methodology to be useful in a dynamic management framework, phase patterns must not be significantly altered by the dynamic management actions that respond to them. Action-dependent phases both conceal actual phase patterns, impairing the predictability of application behavior, and also lead to incorrect management decisions. Previously, Section 5.1 mentioned that the phase definitions based on memory bus transactions per micro-op (Mem/Uop) are resilient to changes in processor voltage and frequency settings. This section justifies this claim with detailed measurements.

The two metrics obtainable with our choice of monitored PMC events are Mem/Uop and Uops per cycle (UPC). We profile the application set with performance counters (PMCs) and record different observed $(UPC, Mem/Uop)$ pairs, which constitute a two-dimensional execution behavior space. Figure 5.7 shows the corresponding exploration space for all acquired $(UPC, Mem/Uop)$ sample pairs for all the set of applications with the lighter data points. These show the diverse characteristics that are covered by these applications. In addition, a boundary is observed as the maximum achievable UPC for each Mem/Uop level, depicted with the "SPEC Boundary" curve. This is an expected effect, as high memory latencies stall dependent execution. Consequently, more memory-bound applications can retire fewer instructions per cycle. To evaluate how the UPC and Mem/Uop metrics change under different DVFS settings, we develop a suite of configurable applications that can pinpoint specific $(UPC, Mem/Uop)$ coordinates in our two-dimensional exploration space. These applications consist of several configurable microkernels that are tuned via performance monitoring to achieve desired Mem/Uop and UPC characteristics. We call these applications the *"IPCxMEM suite"*. The grid points, denoted as "IPCxMEM Grid" in Figure 5.7, represent configurations of the IPCxMEM suite to cover the whole exploration space. These applications evaluate the behavior of the tracked metrics at all possible corners of execution and evaluate how these are affected by DVFS actions.

For our evaluations, we run the IPCxMEM suite in approximately 50 $(UPC, Mem/Uop)$ configurations, uniformly sampling the exploration space grid. We run all configurations at all the available frequency settings of our experimental platform. These are 1500MHz, 1400MHz, 1200MHz, 1000MHz, 800MHz and 600MHz. We monitor UPC and Mem/Uop via PMCs in these frequency settings. Figure 5.8 illustrates the frequency dependence of the two metrics for a representative subset of the representative configurations. Each curve corresponds to a specific IPCxMEM suite application—run at all frequency settings—configured to target a specific UPC and Mem/Uop at the highest frequency. These target values, referenced in the legend, correspond to the specific points of the IPCxMEM grid in

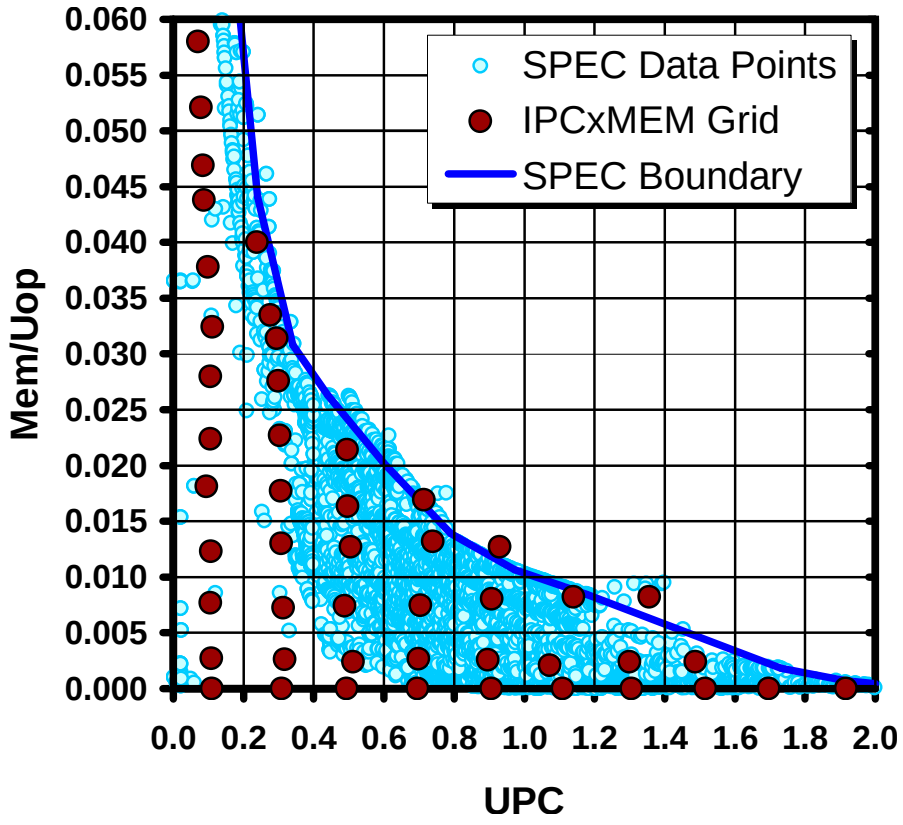


Figure 5.7: Observed (UPC,Mem/Uop) pairs for all experimented applications sampled every 100 million instructions and grid of points covered by our IPCxMEM suite.

Figure 5.7. For example, the top flat UPC curve in Figure 5.8 with legend entry "`UPC=1.9, Mem/Uop=0.0000`" corresponds to the rightmost grid point in Figure 5.7 at the location $(UPC = 1.9, Mem/Uop = 0.0)$.

Figure 5.8 shows the strong dependence of UPC to DVFS settings. UPC mostly has an increasing trend with decreasing frequency. This is because memory latencies are not scaled with DVFS, and therefore, memory accesses complete in fewer CPU cycles at lower frequencies. The frequency dependence of UPC also varies with memory intensity. UPC values for completely CPU-bound configurations (legend entries with $Mem/Uop = 0$) show no dependence to frequency. On the other hand, for highly memory-bound configurations, UPC can change up to 80% across frequencies. These demonstrate the dangerous pitfall we avoid in our phase definitions. Directly using UPC in phase classification is not reliable for dynamic management, as the resulting phases vary with different power

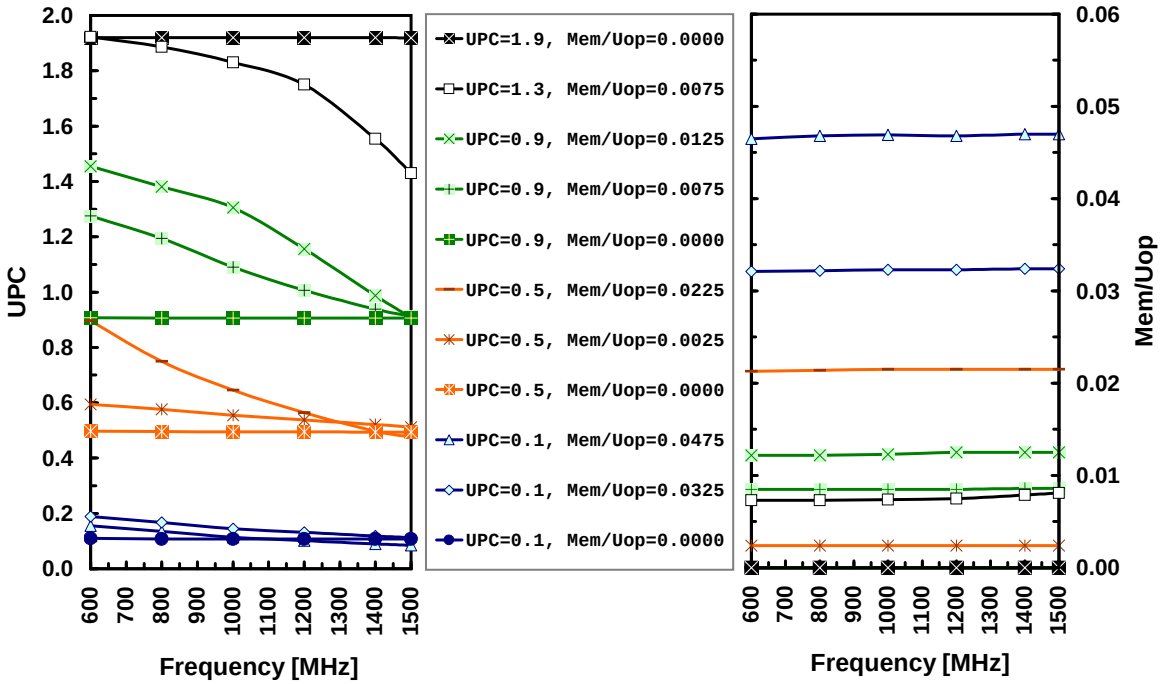


Figure 5.8: Observed UPC and Mem/Uop behavior at six different frequencies for different IPCxMEM grid configurations.

management settings.

Conversely, Figure 5.8 shows that the Mem/Uop parameter has virtually no dependence on DVFS settings. It is almost constant across all frequencies. Therefore, our phase classifications based on Mem/Uop are completely "DVFS invariant" and can be reliably used for runtime phase prediction under our target dynamic power management.

5.5 Phase-Driven Dynamic Power Management: Real-System Implementation

The actual implementation of the on-the-fly phase monitoring and prediction framework runs on a Pentium-M based, off-the-shelf laptop computer, running Linux kernel 2.6-11. Figure 5.9 shows an overview of how this overall implementation operates on our system. Our prototype implementation monitors application execution via performance counters (PMCs) and performs phase predictions at fixed intervals of 100 million instructions in a performance monitoring interrupt (PMI) handler. The runtime phase predictions guide dy-

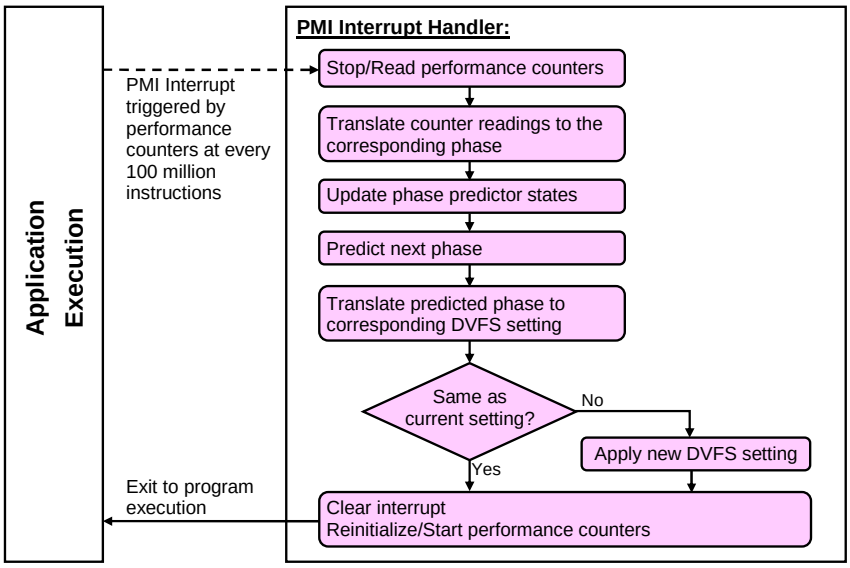


Figure 5.9: The flow of operation for our runtime phase prediction and dynamic power management framework.

namic voltage and frequency scaling (DVFS), readily available on the Pentium-M platform, as the example management application. At each interrupt invocation, after performing the next phase prediction with the GPHT predictor, the interrupt routine translates the predicted phase into a predefined DVFS setting. This setting is then applied to the processor for the next execution interval. After the initial configuration (performed once at system startup) all phase prediction and dynamic management actions operate autonomously, with no observable overheads to user applications. All applications can run natively, without any modifications or additional system or dynamic compiler support.

Figure 5.10 shows the overall prototype implementation and measurement setup for our experiments. This diagram depicts different aspects of our implementation that correspond to on-the-fly phase monitoring and prediction, dynamic power management via DVFS, and additional mechanisms for evaluating runtime phase prediction and performing real power measurements that can match each phase. The following subsections discuss the details of each of these aspects for the prototype platform.

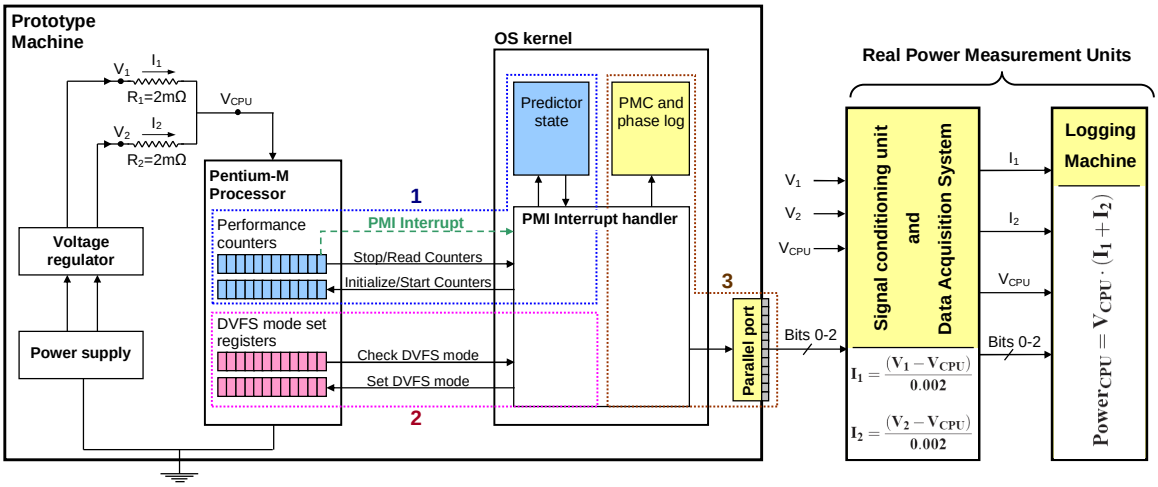


Figure 5.10: Developed measurement and evaluation platform. Regions identified as *1*, *2* and *3* via dashed lines correspond to different parts of implementation relevant to on-the-fly phase monitoring and prediction (*1*), dynamic management with DVFS (*2*) and measurement and evaluation support (*3*).

5.5.1 Runtime Phase Monitoring and Prediction

One of the fundamental challenges of phase detection and prediction on a real system is the impact of system-induced variability. The previous chapter has shown that application phases are prone to several variations at runtime, which can alter the timing and values of observed metrics. To eliminate the effect of timing variations, we monitor phases at fixed instruction granularities with the PMI. This is a relatively more intrusive method than the approach described in Chapter 4 as it requires an initial recompilation of the operating system kernel to specify the necessary bindings for interrupt handling. However, it provides a simpler means to track our performance events with minimal disturbance due to variability effects. We have implemented our PMI handler and supporting system calls as a loadable kernel module (LKM), which can be loaded and unloaded during system operation. These system calls control the PMCs and bind the interrupt handler to the PMC hardware that triggers the interrupt. The implemented LKM also holds the state for our predictors and logs the PMC values and predicted and actual observed phases for our evaluations.

For our experiments, we configured the two available PMCs in the Pentium-M processor to monitor the retired micro-ops and memory bus accesses with the UOPS_RETIRED

and BUS_TRAN_MEM event configurations. We have experimented with various instruction granularities and chose to invoke the interrupt handler every 100 million instructions. This granularity provides a safe lower bound that operates without causing significant overheads and operating system timing violations. After each invocation, the first PMC is reinitialized to overflow after 100 million retired Uops.

After every 100 million instructions, the interrupt handler stops and reads the PMCs, updates the GPHT predictor states, and performs the next phase prediction. It also logs the observed PMC values, actual observed phase for the past period, and the predicted phase for the next period for our evaluations. At its exit, the handler clears the PMC overflow bit, reinitializes the PMCs and time stamp counter (TSC), and restarts the counters.

5.5.2 Dynamic Power Management with DVFS

The on-the-fly phase prediction methodology can guide a range of dynamic management techniques. This work considers DVFS as an example implementation. DVFS is supported on our platform via Intel SpeedStep technology [55]. In our prototype implementation, we use a look-up table, defined at LKM initialization, to quickly translate the predicted phase to one of the 6 DVFS settings within the handler. Table 5.2 shows these settings for the prototype machine and the original phase definitions, which are similar to prior work [179]. For alternative phase definitions or management schemes, we can simply reconfigure this table. At each sampling interval, the handler translates the predicted phase to the corresponding DVFS setting. It then compares this to the current setting and updates the DVFS mode registers if necessary. The 100 million instruction granularity (on the order of 100 ms) guarantees that the overheads induced by interrupt handling and DVFS application (on the order of 10-100 μs) are essentially invisible to native application execution.

5.5.3 Power Measurement

To track the power consumed by the Pentium-M processor, we measure the input voltage and current flow to the processor. For this purpose, we use an external data acquisition

Mem/Uop	Phase #	DVFS Setting
< 0.005	1	(1500 MHz, 1484 mV)
[0.005,0.010)	2	(1400 MHz, 1452 mV)
[0.010,0.015)	3	(1200 MHz, 1356 mV)
[0.015,0.020)	4	(1000 MHz, 1228 mV)
[0.020,0.030)	5	(800 MHz, 1116 mV)
> 0.030	6	(600 MHz, 956 mV)

Table 5.2: Translation of phases to DVFS settings.

system (DAQ) that is connected to the processor board. The laptop board includes two 2 $m\Omega$ *precision sense resistors* that reside between the voltage regulator module and the Pentium-M CPU, shown as $R1$ and $R2$ in Figure 5.10. The total current that flows through these resistors represents the current flow into the CPU. The voltage after the resistors, denoted as V_{CPU}, represents the input voltage of the CPU.

In the measurement setup, we measure the three voltages V_1, V_2 and V_{CPU}, to track processor current and voltage. These voltages—and additional parallel port bits for evaluation support—are first fed into a National Instruments AI05 *Signal Conditioning Unit*. This unit filters the noise on the measured voltage signals and calculates the voltage drop across the two resistors. These voltage drops, $(V_1 - V_{CPU})$ and $(V_2 - V_{CPU})$, and the CPU voltage V_{CPU} are then fed into a National Instruments DAQPad 6070E *Data Acquisition System*. This unit then scales the voltage drops with the resistor values to compute the current flows as $I_1 = (V_1 - V_{CPU})/0.002$ and $I_2 = (V_2 - V_{CPU})/0.002$. The DAQ system monitors a total of eight signals, and has a sampling period of 40 μs. The two measured currents and the CPU voltage, together with additional parallel port signals, are sent to a separate *logging machine*, which logs the observed currents and voltages. The CPU power consumption for each sample is computed on this logging machine as $Power_{CPU} = V_{CPU} \cdot (I_1 + I_2)$. With this complete measurement setup, we can accurately track CPU power consumption. By also utilizing parallel port signaling, described below, our measurement setup can individually compute the power consumption and performance statistics for each 100M-instruction

phase sample as well as for the whole execution of applications.

5.5.4 Evaluation Support

The full operation of our system requires only on-the-fly phase monitoring and prediction, and dynamic power management with DVFS as highlighted in regions 1 and 2 in Figure 5.10. However, to experimentally evaluate our methods, we develop additional instrumentation in our prototype system. First, we use the previously described real power measurement setup to measure processor power consumption. In addition, for detailed power/performance and phase prediction evaluations, we employ additional mechanisms in our implementation; these fall into region 3 in Figure 5.10.

To evaluate runtime phase prediction accuracy and to analyze application behavior, we use a separate kernel log in our LKM. This log keeps track of the actual observed and predicted phases for each sample as well as memory accesses per Uop and Uops per cycle for each phase. At each invocation, the handler records relevant information in this log. Afterwards, a user-level tool can access this information via separate system calls.

The execution of the processor and the real power measurements are inherently two completely independent processes. To provide a synchronizing link between the two sides of our framework, we use parallel port bits that signal specific processor execution information to the DAQ system. We use three parallel port bits. *Bit 2* is set from the user level via system calls at the start of an application execution and is cleared when an application ends. This helps DAQ to measure power specifically during an application execution. *Bit 1* is used to distinguish between the application and interrupt execution. This bit is set by the handler at the entrance to the handler routine and is cleared at exit. Finally, *bit 0* is used to help the DAQ track each phase. The handler flips this bit at each sampling interval so that the DAQ and the logging machine can distinguish each phase and compute power and performance statistics for individual phases.

Figure 5.11 shows a detailed view of the overall operation of our deployed system with the `applu` benchmark, performing on-the-fly phase predictions with the GPHT predictor

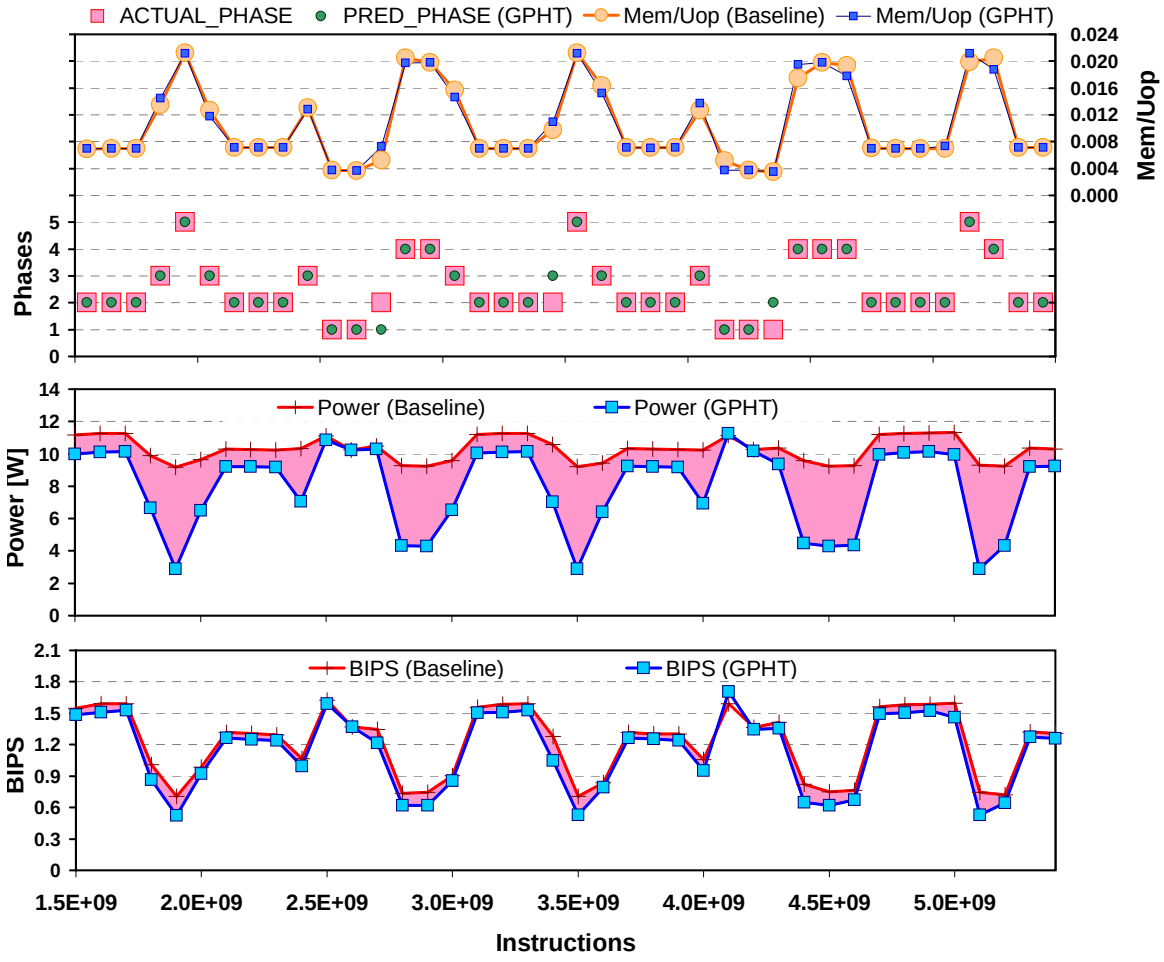


Figure 5.11: Overall operation of our framework, shown with the `applu` benchmark, in comparison to the baseline system. Top chart shows the observed Mem/Uop, actual and predicted phases. Middle and lower charts show achieved power savings and induced performance degradation in the shaded regions.

and dynamic power management with DVFS. The figure shows the measured prediction, power and performance results with respect to a baseline, unmanaged system. The top chart shows the observed Mem/Uop behavior for the two runs of `applu`, with and without the described techniques. The two curves are almost identical between the two real-system runs. This example shows: *(i)* the phases defined by Mem/Uop are DVFS invariant and can be safely used for phase prediction under dynamic management responses; and *(ii)* the fixed instruction granularity phase definitions are resilient to real-system variations. The lower part of the top chart shows the actual phases and predicted phases with the GPHT. The predictions with the GPHT predictor significantly overlap with the actual phase be-

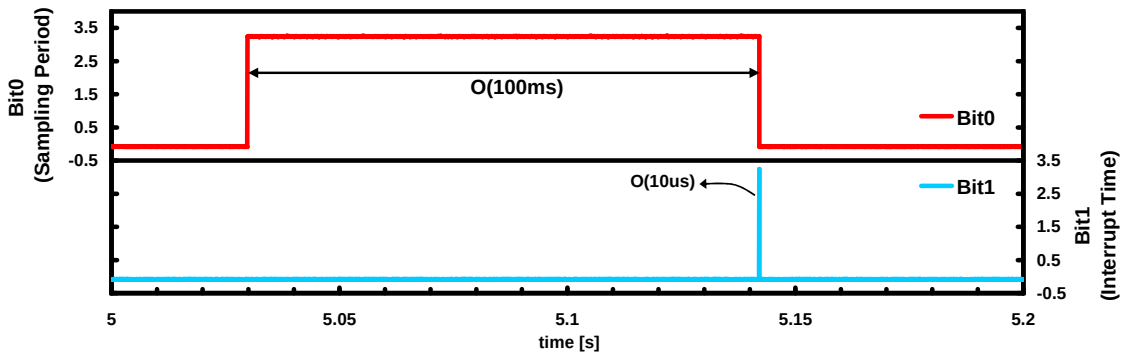


Figure 5.12: Observed overhead with our dynamic phase monitoring and prediction platform.

havior of this highly varying application. The middle chart shows the measured power for `applu` without any power management (baseline) and with GPHT-based power management (GPHT). The shaded area between the two curves demonstrates power savings achieved with our approach. The lower chart shows the observed performance as billions of instructions per second (BIPS) for the two systems, where the shaded area demonstrates the relatively small performance degradation induced by our framework. These latter two charts, together with the general results presented in Section 5.6 clearly present the advantages brought by our framework for improving power/performance efficiency. By efficiently adapting processor execution to varying application behavior, we achieve significant power savings with small degradations in performance.

5.5.5 Management Overhead

To evaluate the overhead of our dynamic management framework, we need to separate actual program execution from the phase prediction and mode setting operations. Using the previously described parallel port signaling mechanism, we monitor the entrance and exit of the interrupt handler, which performs all of the phase prediction, logging and dynamic frequency setting actions. Figure 5.12 demonstrates this operation overhead. The upper plot shows the duration of an individual phase (tracked by alternating *bit*0 values) and the lower plot shows the interrupt timing (tracked by a high *bit*1 value) as a short spike during the phase change.

This evaluation shows more than three orders of magnitude difference between the management overhead and phase durations. As a matter of fact, as our data acquisition system works at a sampling period of 40 μs, it cannot detect many of the overhead periods. For example, no spike in $bit1$ is observed during the first phase change in the plot. This corroborates the insignificant overhead of our runtime, phase-prediction-driven dynamic management strategy. On average, the observed cumulative overhead due to phase monitoring and prediction, logging and the application of dynamic management actions is less than 0.1%.

5.6 Phase-Driven Dynamic Power Management Results

The previous sections described our phase definitions and on-the-fly phase prediction methodology. They have presented a full-fledged deployed system. This section evaluates the final target of our complete framework, dynamic power management with DVFS, guided by on-the-fly, GPHT-based phase predictions. It presents the overall dynamic power management results for all the experimented benchmarks with three sets of information. Figure 5.13 depicts power and performance results with our experimental system, using the GPHT predictor, as normalized to baseline execution. The top graph of the figure shows achieved billions of instructions per second (BIPS) as a measure of performance. The middle and bottom parts plot the power and energy-delay product (EDP) as measures of power-performance efficiency for the baseline unmanaged system and our dynamic management framework. The benchmarks are shown in decreasing EDP order with GPHT-based management.

The application categories that have been previously discussed in Section 5.2 also guide our understanding of the dynamic power management results. Many of the $Q1$ benchmarks experience little power saving and small performance degradation. They have highly stable, non-varying execution behavior with little power saving potential and close to baseline performance under dynamic management. Some of the $Q1$ applications, such as `apsi`

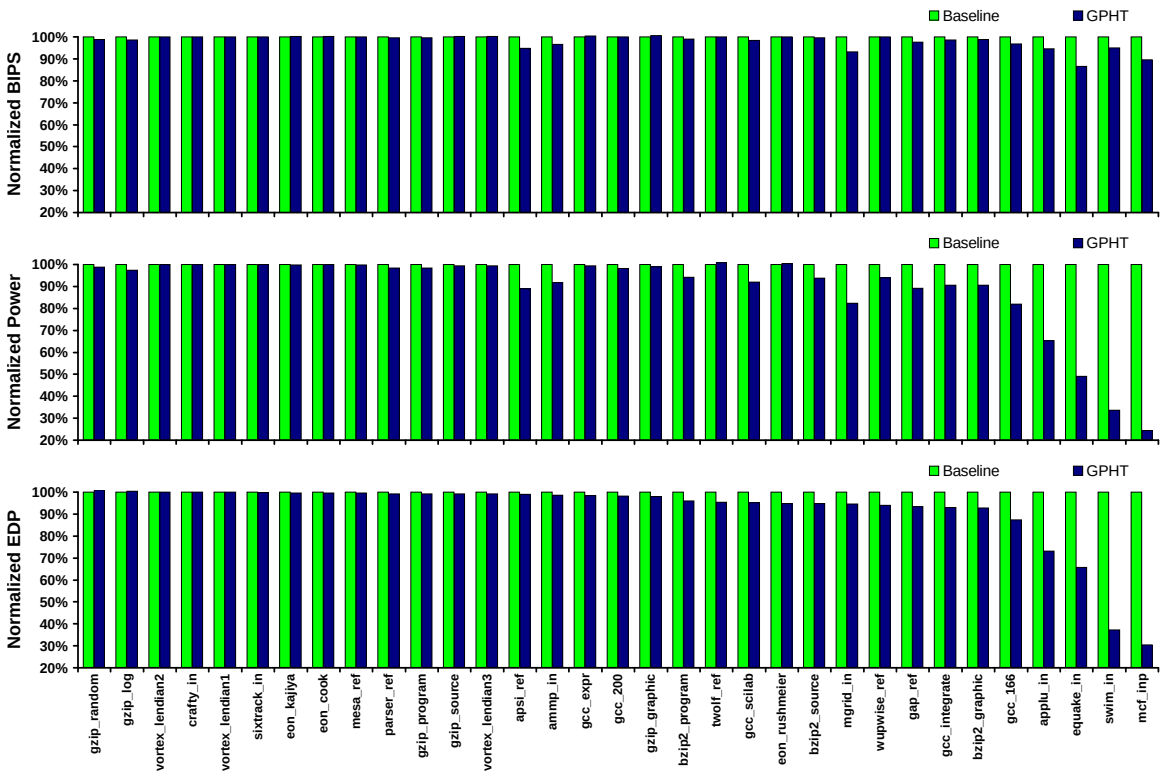
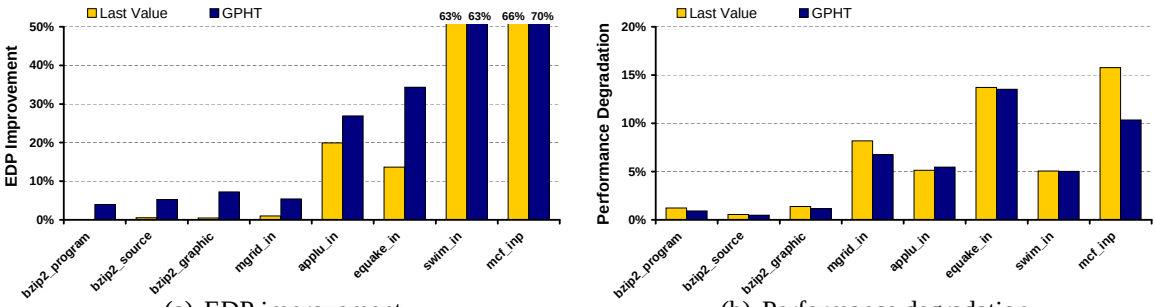


Figure 5.13: Runtime phase-prediction-guided dynamic power management results. From top to bottom, the charts show performance, power and energy delay product achieved by our framework with respect to baseline execution.

and `ammp`, actually achieve significant power savings due to their relatively higher variability. However, due to their lower power saving potential, these are also accompanied by observable performance degradations. Thus, overall EDP improvement remains less significant. On the other hand, *Q*2 and *Q*3 applications generally demonstrate substantial power savings as well as EDP improvements. The trivial *Q*2 applications `swim` and `mcf` exhibit above 60% EDP improvements. Our experimental system also achieves EDP improvements as high as 34% for the highly variable *Q*3 benchmarks, such as `equake`. One exception to this is `mgrid`. Although it shows high power savings, `mgrid` also experiences comparable performance degradation. Therefore, its EDP improvement remains smaller than the other *Q*3 applications. One probable reason for this is having higher concurrent execution at memory-bound regions. For all *Q*2, *Q*3 and *Q*4 applications, the average EDP improvement is 27%, with an average performance degradation of 5%.



(a) EDP improvement (b) Performance degradation

Figure 5.14: EDP improvement and performance degradation with GPHT and last value prediction for *Q*2, *Q*3 and *Q*4 benchmarks.

5.6.1 Improvements with GPHT over Reactive Dynamic Management

Many of the previous dynamic management techniques simply respond to previously observed application behavior. We refer to these as "reactive" approaches. Although these approaches perform well for many applications, they are prone to significant misconfigurations for workloads with quickly-varying behavior. On the other hand, our on-the-fly, GPHT-based dynamic management framework can respond to these variations proactively, providing better system adaptation. Here we compare the achieved power/performance trade-offs of our GPHT-based dynamic management framework to those of a reactive system. For the reactive method, we use last-value prediction.

Figure 5.14 shows the achieved EDP improvement and performance degradation with both dynamic management methods. It shows the results for the highly variable *Q*3 and *Q*4 benchmarks, as well as the high-power-savings and low-variation *Q*2 benchmarks. For many of the *Q*1 applications, the reactive approach performs similarly to our GPHT-based approach. For these stable applications, responding to previously seen behavior is already the near-optimal approach.

Figure 5.14 depicts the advantage of employing dynamic management guided by on-the-fly phase predictions. The two *Q*2 benchmarks behave somewhat differently. For `swim`, which has virtually no variability (lying on the *x* axis in Figure 5.1) both approaches achieve almost identical results. For `mcf`, which shows a small amount of variability, GPHT-based

management achieves a slightly better EDP and less performance degradation. For the highly-variable and memory-bound *Q*3 benchmarks, GPHT-based, proactive management achieves superior EDP improvements. The performance degradations experienced by our GPHT framework are less than or comparable to those of the last value methods. As expected, the improvements with the less memory-bound *Q*4 applications are usually less significant than the other benchmarks. Nonetheless, while the reactive approach provides almost no benefits for these applications, GPHT-based dynamic management improves their EDP by approximately 5%. On average, GPHT-based dynamic management achieves an EDP improvement of 27%, with a performance degradation of 5%. The last-value-based reactive approach achieves 20% EDP improvement and 6% performance degradation for the same set of applications. Thus, applying dynamic management under the supervision of our on-the-fly phase predictions provides a 7% EDP improvement over the reactive method, while inducing comparable or less performance degradation. These results show the significant benefits of runtime phase prediction and its application to dynamic power management.

5.6.2 Alternative Phase Definitions

Section 5.5 claimed that in our real-system implementation we can simply adjust our phase definitions and the corresponding DVFS look-up table for alternative implementations. For example, the observed performance degradations that are acceptable for some applications may not be acceptable to others. In such a scenario, it might be preferable to reduce the power savings to achieve better performance. Here, we implement such an alternative dynamic management system that aims to limit performance degradation to 5%. For this implementation, we redefine our phases to meet our performance goal with the help of previous *IPCxMEM* experiments described in Section 5.4. We look at the achieved BIPS at each DVFS setting for each of the IPCxMEM grid points, and draw the DVFS domains on our grid that satisfy our performance target. After this step, we redefine our phases to match these DVFS settings. Based on these phase definitions, our new deployed system

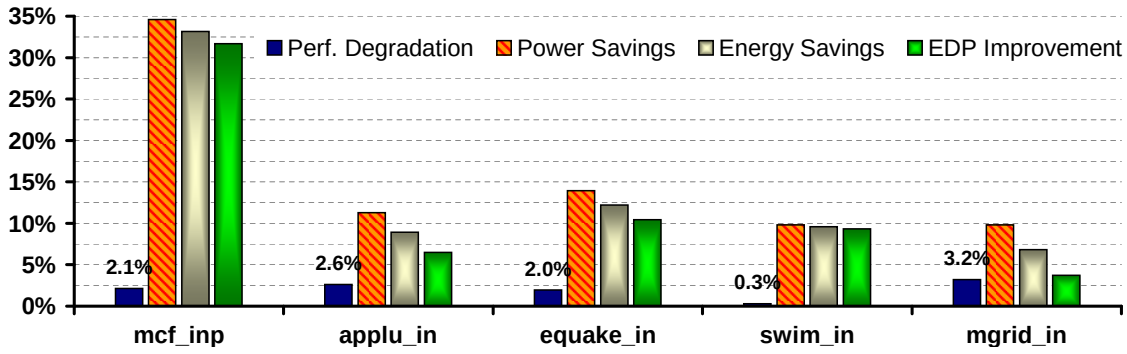


Figure 5.15: Power/Performance results for our conservative phase definitions that aim to bound performance degradation by 5%.

meets the target performance with less aggressive power savings.

Figure 5.15 shows the resulting performance degradations, power and energy savings, and EDP improvements for the five benchmarks that originally had more than 5% performance degradations. With the new conservative phase definitions, all of these applications experience performance degradations significantly lower than 5%. Thus, our new system can successfully sustain application performances within our specified degradation limit. On the other hand, due to smaller power savings, the EDP improvements are also reduced significantly to meet the performance targets.

These results show the versatility of the phase-based dynamic management framework, which can be simply configured for different targets under different scenarios. These reconfigurations can even be performed at runtime, after system deployment, with minimal intrusion to overall system operation. Thus, our complete real-system implementation, presented in this chapter, serves as an effective, generic power management framework, which can be employed on a running system to support different dynamic management goals.

5.7 Related Work

Several previous studies investigate methods to monitor and utilize application phases for architectural and system adaptations. Dhodapkar and Smith use application working set information to guide dynamic hardware reconfigurations [41]. Zhou et al. monitor memory access patterns for energy-efficient memory allocation [186]. Annavaram et al. identify

sequential and parallel phases of parallel applications to distribute threads efficiently on an asymmetric multiprocessor [5]. Weissel and Bellosa also monitor the memory boundedness of applications to adapt processor execution to different phases on the fly [176]. These works show interesting applications for different aspects of application phase behavior. However, they do not consider predicting future phase behavior of applications and perform adaptive responses reactively, based on most recent behavior.

Some earlier work also considers prediction of future application behavior. Duesterwald et al. utilize performance counters to predict certain metric behavior such as IPC and cache misses based on previous history [44]. They also show that table-based predictors perform significantly better than statistical approaches to predict variable application behavior. Lau et al. consider prediction of phase transitions as well as sample phase durations using different predictors [109]. While these works provide significant insights to predictability of application behavior, they do not evaluate the runtime applicability of these predictions to dynamic management.

Sherwood et al. describe a microarchitectural phase predictor based on the traversed basic blocks [153]. They apply this prediction methodology to dynamic cache reconfigurations and scaling of pipeline resources. This work describes fine-grained, microarchitecture-level phase monitoring and dynamic management, based on architectural simulations, while the work in this chapter describes a deployed real-system framework for on-the-fly phase prediction of running applications and system-level management. Shen et al. detect repetitive phases at runtime by monitoring reuse distance patterns with application to cache configurations and memory remapping [150]. This work employs detailed program profiling and instrumentation to detect repetitive phases. In contrast, our work identifies recurrent execution and predicts phases seamlessly during native application execution without prior instrumentation or profiling. Wu et al. also describe a real-system implementation of a runtime DVFS optimizer that monitors application memory accesses [179]. That work requires the applications to execute from within a dynamic instrumentation framework and

relies on periodic dynamic profiling of code regions, inducing additional operation overheads. In comparison, our deployed system operates autonomously on any running application, without necessitating any dynamic instrumentation support or prior profiling, and with no observable overheads to application execution.

5.8 Summary

This chapter presented a fully-automated, real-system framework for on-the-fly phase prediction of running applications. These runtime phase predictions have been used to guide dynamic voltage and frequency scaling (DVFS) as the underlying dynamic management technique on a deployed system.

This work has experimented with different prediction methods and proposed a *Global Phase History Table* (GPHT) predictor, leveraged from a common branch predictor architecture. Our GPHT predictor performs accurate on-the-fly phase predictions for running applications with no visible overheads. For highly variable applications, our GPHT predictor could reduce mispredictions by 6X, compared to the statistical approach. This phase prediction framework efficiently cooperates with DVFS to dynamically adapt processor execution to varying workload behavior. DVFS, guided by these phase predictions, has improved the energy-delay product of variable workloads by as much as 34%, and on average by 27%. Compared to a reactive approach, our method has improved the energy-delay product of applications by as much as 20% and on average by 7%.

The results of this work show the promising benefits of runtime phase prediction and its application to dynamic management. As power management continues to be an increasingly pressing concern, the necessity of such workload-adaptive techniques also increases. The fully-autonomous real-system solution presented in this chapter, with its energy-saving potential and negligible-overhead operation, can serve as a foundation for many dynamic management applications in current and emerging systems.

Chapter 6

Conclusions

The work in this thesis explores real-system techniques to characterize and predict dynamically-varying workload power behavior. It develops workload-adaptive dynamic power management methods that proactively respond to the changes in application demands. The techniques discussed in this thesis primarily operate at the hardware-software boundary. They utilize architecture-level information to guide system-level monitoring and control. The overarching contributions of this work are *(i)* the developed real-system frameworks for runtime power monitoring, phase analysis, and phase-driven dynamic power management; *(ii)* proposed workload phase monitoring, detection and prediction techniques; and *(iii)* their application to workload-adaptive power management.

In particular, this research has shown that hardware performance monitors that are available in most architectures can effectively model the architectural power consumption of processors. The prototype runtime power monitoring and estimation implementation presented in this work achieved power estimations within 10% of actual processor power dissipation. This work has demonstrated power-oriented phase analysis techniques that utilize performance monitoring information to discern varying workload power characteristics. The small set of phases acquired with these techniques represented overall power characteristics of workloads on average within 5% of the actual measured behavior. It has evaluated the efficacy of control-flow-based application features as well as performance

monitoring information in characterizing workload power characteristics. While both approaches proved to be useful in understanding workload power behavior, performance-monitoring-based phases achieved on average 33% less errors. This research has proposed phase detection techniques that are resilient to system-induced variations in tracked workload features. It showed that representing runtime workload execution in terms of phase transitions improves the detection of repetitive phases by $6X$ on average, compared to previous approaches. This thesis has also proposed a runtime global phase history table predictor that can accurately predict future application phases on a real system. This predictor achieved $2.4X$ fewer average phase mispredictions than prior approaches. Last, this research has demonstrated a complete implementation of a phase-driven, workload-adaptive power management infrastructure. This infrastructure presented the significant benefits of phase-based adaptations for power-efficient computation with 27% energy-delay product improvements on a running system.

This thesis shows a complete flow of methods from runtime power and performance monitoring to phase analysis and workload-adaptive power management. While this research describes specific implementations and applications, the techniques proposed in this thesis are applicable to a broad range of computing systems and dynamic management applications that can be employed at both microarchitecture and system level. These include workload-adaptive microarchitectural resource scaling, dynamic thermal control, and runtime management of computing systems for fault and variation tolerant execution.

6.1 Future Directions

There are several future avenues of research that are related to the techniques presented in this work. One fundamental observation that drives these research directions is that the potential of the emerging systems is defined around meeting certain workload or platform demands and adapting to the technology challenges in these platforms. For example, these demands indicate when to enable cores or specialized engines in multicore architectures or

how to manage processing elements with varying power-performance characteristics due to process variations and heterogeneous system implementations. Projecting and adapting to the varying workload and platform demands is key for achieving the potential and goals of these evolving trends.

One immediate research path for phase characterization and workload-adaptive management is considering multiple management responses in coordination. Most of the prior work focuses on isolated management schemes for singular constraints. However, comprehensive control strategies that account for the nontrivial interaction of different management responses are necessary for efficient dynamic management. This is particularly important in emerging processor architectures that are highly limited by power and temperature constraints. Efficient operation of these platforms requires runtime adaptations that can respond to the power and thermal demands of workloads effectively. This research direction faces two important challenges. First, elaborate phase classification methods that can mutually express the power, thermal and other characteristics of applications must be developed. Second, intelligent control schemes must efficiently coordinate multiple management responses that correspond to these runtime phase mappings. While this can be a challenging process, future architectures include increasingly more adaptive components and demand such control mechanisms.

An important trend in current architectures is the widespread adoption of chip multiprocessors as the common design choice. Very interesting new challenges come with this new direction, where dynamic adaptations that operate across multiple cores become at least as important as the management of individual cores. These multicore platforms require hierarchical monitoring and control techniques that distinguish between local, per-core adaptations and global, chip-level management. A holistic approach towards efficient management of these systems is a three-tier framework that spans both architectural and system-level responses. At the higher level, this includes large-scale, system-level approaches such as thread migration and parallelization techniques. Chip-level management

requires global monitoring and control mechanisms for closed-loop management driven by chip power and thermal constraints. At the core level, local core monitoring and control tracks per-core workload phase behavior and drives open-loop actions that can be employed without the knowledge of the workload characteristics in other cores. There are many interesting open questions in this management strategy such as the scalability and implementation of such control at the hardware-software boundary, joint optimization of local and global actions, and a generalized management solution for multiprogrammed and parallel multithreaded workloads. The runtime workload phase analysis and adaptation techniques presented in this thesis provide a useful foundation for the development of the hierarchical monitoring and control required for this future research direction.

The adaptive management strategies that are discussed in this thesis are also applicable in the embedded and real-time systems domain. An interesting direction in this domain is extending the phase-based dynamic adaptation techniques to these application platforms. In such a framework, mappings of phases to dynamic adaptations can be reconfigured at runtime based on the imposed deadlines. Such adaptive mappings, together with runtime phase predictions, can enable more efficient scheduling of the operations to the available processing elements in real-time embedded computing platforms.

Another important future direction for this work is considering adaptive management techniques to mitigate the emerging technology challenges. As semiconductor technologies scale down to nanometer dimensions, integrated circuits exhibit highly variable characteristics and reduced functional reliability. Under these conditions, variation and defect tolerance must become an integral component of architecture and systems design. Part of this translates into efficient dynamic management of varying processing resources. This research direction shares certain similarities with workload phase prediction at its basis. In addition to projecting workload demands at runtime, it involves extending the prediction models for predicting processor behavior across different operating modes. Moreover, dynamically changing mappings between workload phases and management actions are

required for adapting execution with varying power and temperature envelopes, as well as with changing architectural capabilities.

Overall, this thesis provides a roadmap to effective on-the-fly phase monitoring and prediction on real-systems and lays the ground work for their application to workload-adaptive dynamic management techniques. The outcomes of my research reveal the potential of such workload-adaptive management for improving processor power efficiency. As adaptive and autonomous management strategies become increasingly essential for power-efficient and reliable computing, my research offers promising practical techniques that can be integral components of emerging computing systems.

Bibliography

[1] N. AbouGhazaleh, B. Childers, D. Mosse, R. Melhem, and M. Craven. Energy Management for Real-time Embedded Applications with Compiler Support. In *Proceedings of the Proceedings of the 2003 ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2003.

[2] A. R. Alameldeen and D. A. Wood. Variability in Architectural Simulations of Multi-threaded Workloads. In *Proceedings of 9th International Symposium on High Performance Computer Architecture (HPCA-9)*, Feb. 2003.

[3] D. Albonesi, R. Balasubramonian, S. Dropsho, S. Dwarkadas, E. Friedman, M. Huang, V. Kursun, G. Magklis, M. Scott, G. Semeraro, P. Bose, A. Buyuktosunoglu, P. Cook, and S. Schuster. Dynamically Tuning Processor Resources with Adaptive Processing. *IEEE Computer*, 36(12):43–51, 2003.

[4] M. Anis, S. Areibi, and M. Elmasry. Design and Optimization of Multi-Threshold CMOS (MTCMOS) Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(10):1324–1342, Oct. 2003.

[5] M. Annavaram, E. Grochowski, and J. Shen. Mitigating Amdahl's Law Through EPI Throttling. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA-32)*, 2005.

[6] M. Annavaram, R. Rakvic, M. Polito, J.-Y. Bouguet, R. Hankins, and B. Davies. The Fuzzy Correlation between Code and Performance Predictability. In *Proceedings of the 37th International Symp. on Microarchitecture*, 2004.

[7] A. Azevedo, I. Issenin, R. Cornea, R. Gupta, N. Dutt, A. Veidenbaum, and A. Nicolau. Profile-based Dynamic Voltage Scheduling using Program Checkpoints. In *Proceedings of the conference on Design, automation and test in Europe (DATE'02)*, Mar. 2002.

[8] R. I. Bahar and S. Manne. Power and Energy Reduction Via Pipeline Balancing. In *Proceedings of the 28th International Symposium on Computer Architecture (ISCA-28)*, June 2001.

[9] R. Balasubramonian, D. H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *International Symposium on Microarchitecture*, pages 245–257, 2000.

[10] H. Bao, J. Bielak, O. Ghattas, L. F. Kallivokas, D. R. O'Hallaron, J. R. Shewchuk, and J. Xu. Large-scale Simulation of Elastic Wave Propagation in Heterogeneous Media on Parallel Computers. *Computer Methods in Applied Mechanics and Engineering*, 152(1–2):85–102, Jan. 1998.

[11] R. D. Barnes, E. M. Nystrom, M. C. Merten, and W. mei W.Hwu. Vacuum packing: extracting hardware-detected program phases for post-link optimization. In *Proceedings of the 35th International Symp. on Microarchitecture*, Nov. 2002.

[12] L. A. Barroso. The Price of Performance. *ACM Queue*, 3(7):48–53, Sept. 2005.

[13] F. Bellosa. The benefits of event-driven energy accounting in power-sensitive systems. In *Proceedings of 9th ACM SIGOPS European Workshop*, September 2000.

[14] F. Bellosa, A. Weissel, M. Waitz, and S. Kellner. Event-Driven Energy Accounting for Dynamic Thermal Management. In *Proceedings of the Workshop on Compilers and Operating Systems for Low Power (COLP'03), New Orleans*, Sept. 2003.

[15] B. Bentley. Validating the Intel Pentium 4 microprocessor. In *Design Automation Conference*, pages 244–248, 2001.

[16] R. Berrendorf and B. Mohr. *PCL - The Performance Counter Library: A Common Interface to Access Hardware Performance Counters on Microprocessors (Version 2.0)*. http://www.kfa-juelich.de/zam/PCL/.

[17] R. Bianchini and R. Rajamony. Power and energy management for server systems. *IEEE Computer*, 37(11), November 2004.

[18] W. Bircher, J. Law, M. Valluri, and L. K. John. Effective Use of Performance Monitoring Counters for Run-Time Prediction of Power. Technical Report TR-041104-01, University of Texas at Austin, Nov. 2004.

[19] W. L. Bircher, M. Valluri, J. Law, and L. K. John. Runtime identification of microprocessor energy saving opportunities. In *Proceedings of the 2005 International Symposium on Low Power Electronics and Design (ISLPED)*, 2005.

[20] B. Brock and K. Rajamani. Dynamic Power Management for Embedded Systems. In *Proceedings of the IEEE International SOC Conference*, Sept. 2003.

[21] D. Brooks, P. Bose, V. Srinivasan, M. K. Gschwind, P. G. Emma, and M. G. Rosenfield. New Methodology for Early-Stage, Microarchitecture-Level Power-Performance Analysis of Microprocessors. *IBM J. of Research and Development*, 46(5/6):653–670, 2003.

[22] D. Brooks and M. Martonosi. Dynamically exploiting narrow width operands to improve processor power and performance. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, Jan. 1999.

[23] D. Brooks and M. Martonosi. Dynamic thermal management for high-performance microprocessors. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture (HPCA-7)*, January 2001.

[24] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th International Symposium on Computer Architecture*, June 2000.

[25] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.

[26] A. Buyuktosunoglu, S. Schuster, D. Brooks, P. Bose, P. W. Cook, and D. H. Albonesi. An Adaptive Issue Queue for Reduced Power at High Performance. In *Proceedings of the First International Workshop on Power-Aware Computer Systems (PACS'00)*, 2001.

[27] B. Calder, T. Sherwood, E. Perelman, and G. Hamerly. SimPoint web page. http://www.cs.ucsd.edu/simpoint/.

[28] A. P. Chandrakasan and A. Sinha. JouleTrack: A Web Based Tool for Software Energy Profiling. In *Proceedings of the* 38^{th} *Design Automation Conference (DAC'01)*, June 2001.

[29] F. Chang, K. Farkas, and P. Ranganathan. Energy driven statistical profiling: Detecting software hotspots. In *Proceedings of the Proceedings of the Workshop on Computer Systems*, 2002.

[30] J. Chase, D. Anderson, P. Thakar, A. Vahdat, and R. Doyle. Managing energy and server resources in hosting centers. In *Proceedings of the 18th Symposium on Operating Systems Principles (SOSP)*, October 2001.

[31] M. Chin. Desktop CPU Power Survey. In *SPCR Forum*, 2006.

[32] C.-B. Cho and T. Li. Complexity-based Program Phase Analysis and Classification. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sept. 2006.

[33] K. Choi, R. Soma, and M. Pedram. Dynamic Voltage and Frequency Scaling based on Workload Decomposition. In *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED)*, Aug. 2004.

[34] G. Contreras and M. Martonosi. Power Prediction for Intel XScale Processors Using Performance Monitoring Unit Events. In *Proceedings of the 2005 International Symposium on Low Power Electronics and Design (ISLPED)*, 2005.

[35] J. Cook, R. L. Oliver, and E. E. Johnson. Examining performance differences in workload execution phases. In *Proceedings of the IEEE International Workshop on Workload Characterization (WWC-4)*, 2001.

[36] N. Corporation. NVIDIA GeForce 8800 GPU Architecture Overview. Technical Brief TB-02787-001_v01, NVIDIA Corporation, Nov. 2006.

[37] M. Curtis-Maury, J. Dzierwa, C. D. Antonopoulos, and D. S. Nikolopoulos. Online Power-Performance Adaptation of Multithreaded Programs using Event-Based Prediction. In *Proceedings of the 20th ACM International Conference on Supercomputing (ICS)*, June 2006.

[38] A. Das, J. Lu, and W.-C. Hsu. Region Monitoring for Local Phase Detection in Dynamic Optimization Systems. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, Mar. 2006.

[39] P. J. Denning. The working set model for program behavior. *Communications of the ACM*, pages 323–333, May 1968.

[40] A. Dhodapkar and J. Smith. Comparing Program Phase Detection Techniques. In 36th International Symp. on Microarchitecture, 2003.

[41] A. Dhodapkar and J. Smith. Managing multi-configurable hardware via dynamic working set analysis. In 29th Annual International Symposium on Computer Architecture, 2002.

[42] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification. Second Edition*. Wiley Interscience, New York, 2001.

[43] A. Dudani, F. Mueller, and Y. Zhu. Energy Conserving Feedback EDF Scheduling for Embedded Systems with Real-time Constraints. In *LCTES/SCOPES '02: Proceedings of the joint conference on Languages, compilers and tools for embedded systems*, 2002.

[44] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and Predicting Program Behavior and its Variability. In *IEEE PACT*, pages 220–231, 2003.

[45] D. Economou, S. Rivoire, C. Kozyrakis, and P. Ranganathan. Full-system Power Analysis and Modeling for Server Environments. In *Proceedings of the Workshop on Modeling Benchmarking and Simulation (MOBS)*, June 2006.

[46] L. Eeckhout, R. Sundareswara, J. Yi, D. Lilja, and P. Schrater. Accurate Statistical Approaches for Generating Representative Workload Compositions. In *Proceedings of the IEEE International Symposium on Workload Characterization*, Oct. 2005.

[47] D. Ernst, S. D. Nam Sung Kim, S. Pant, T. Pham, R. Rao, C. Ziesler, D. Blaauw, T. Austin, and T. Mudge. Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation. In *Proceedings of the 36th International Symp. on Microarchitecture*, Dec. 2003.

[48] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy Caches: Simple Techniques for Reducing Leakage Power. In *Proceedings of the 29th International Symposium on Computer Architecture (ISCA-29)*, May 2002.

[49] K. Flautner and T. Mudge. Vertigo: Automatic Performance-Setting for Linux. In *Proceedings of the Fifth Symposium on Operating System Design and Implementation OSDI'02*, 2002.

[50] J. Flinn. *Extending Mobile Computer Battery Life through Energy-Aware Adaptation*. PhD thesis, Computer Science Department, Carnegie Mellon University, Dec. 2001.

[51] J. Flinn and M. Satyanarayanan. Powerscope: a tool for profiling the energy usage of mobile applications. In *Second IEEE Workshop on Mobile Computing Systems and Applications*, pages 2–10, Feb. 1999.

[52] B. B. Fraguela, R. Doallo, J. Tourino, and E. L. Zapata. A Compiler Tool to Predict Memory Hierarchy Performance of Scientific Codes. *Parallel Computing*, 30(2):225–228, 2004.

[53] J. Friedrich, B. McCredie, N. James, B. Huott, B. Curran, E. Fluhr, G. Mittal, E. Chan, Y. Chan, D. Plass, S. Chu, H. Le, L. Clark, J. Ripley, S. Taylor, J. Dilullo, and M. Lanzerotti. Design of the POWER6 Microprocessor. In *IEEE International Solid-State Circuits Conference (ISSCC 2007)*, Feb. 2007.

[54] S. Ghosh, M. Martonosi, and S. Malik. Cache Miss Equations: A Compiler Framework for Analyzing and Tuning Memory Behavior. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(4):703–746, 1999.

[55] S. Gochman, R. Ronen, I. Anati, A. Berkovits, T. Kurts, A. Naveh, A. Saeed, Z. Sperber, and R. C. Valentine. The Intel Pentium M Processor: Microarchitecture and Performance. *Intel Technology Journal, Q2, 2003*, 7(02), 2003.

[56] M. Golden, S. Arekapudi, G. Dabney, M. Haertel, S. Hale, L. Herlinger, Y. Kim, K. McGrath, V. Palisetti, and M. Singh. A 2.6GHz Dual-Core 64b x86 Microprocessor with DDR2 Memory Support. In *IEEE International Solid-State Circuits Conference (ISSCC 2006)*, Feb. 2006.

[57] M. Gschwind. Chip Multiprocessing and the Cell Broadband Engine. IBM Researh Report RC-23921, IBM T. J. Watson Research Center, Feb. 2006.

[58] S. P. Gurrum, S. K. Suman, Y. K. Joshi, and A. G. Fedorov. Thermal Issues in Next-Generation Integrated Circuits. *IEEE Transactions on Device and Materials Reliability*, 4(4):709–714, Dec. 2004.

[59] S. Gurumurthi, A. Sivasubramaniam, M. J. Irwin, N. Vijaykrishnan, M. Kandemir, T. Li, and L. K. John. Using Complete Machine Simulation for Software Power Estimation: The SoftWatt Approach. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 2002.

[60] S. Gurumurthi, A. Sivasubramaniam, M. Kandemir, and H. Franke. DRPM: Dynamic Speed Control for Power Management in Server Class Disks. *Computer Architecture News*, 31(2):169 – 181, May 2003.

[61] S. Gurun and C. Krintz. A Run-Time, Feedback-Based Energy Estimation Model For Embedded Devices. In *Proceedings of the International Conference on Hardware-Software Codesign and System Synthesis (CODES+ISSS)*, Oct. 2006.

[62] J. Haid, G. Kafer, C. Steger, R. Weiss, , W. Schogler, and M. Manninger. Run-time energy estimation in system-on-a-chip designs. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan. 2003.

[63] T. Heath, A. P. Centeno, P. George, L. Ramos, Y. Jaluria, and R. Bianchini. Mercury and freon: Temperature emulation and management in server systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2006.

[64] T. Heath, B. Diniz, E. V. Carrera, W. Meira Jr., and R. Bianchini. Energy conservation in heterogeneous server clusters. In *Proceedings of the 10th Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2005.

[65] T. Heath, E. Pinheiro, J. Hom, U. Kremer, and R. Bianchini. Code Transformations for Energy-Efficient Device Management. *IEEE Transactions on on Computers*, 53(8):974– 987, Aug. 2004.

[66] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman Publishers, 2003. Third Edition.

[67] S. Heo, K. Barr, and K. Asanovic. Reducing Power Density through Activity Migration. In *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED), Seoul, Korea*, Aug. 2003.

[68] M. J. Hind, V. T. Rajan, and P. F. Sweeney. Phase Shift Detection: A Problem Classification. IBM Researh Report RC-22887, IBM T. J. Watson, Aug. 2003.

[69] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal, First Quarter 2001*, 2001. http://developer.intel.com/technology/itj/.

[70] J. Hom and U. Kremer. Inter-program Compilation for Disk Energy Reduction. In *Workshop on Power-Aware Computer Systems (PACS'03)*, 2003.

[71] C.-H. Hsu and U. Kremer. The design, implementation, and evaluation of a compiler algorithm for CPU energy reduction. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 38–48, 2003.

[72] C. Hu, D. Jimenez, and U. Kremer. Toward an Evaluation Infrastructure for Power and Energy Optimizations. In *Workshop on High-Performance, Power-Aware Computing*, 2005.

[73] S. Hu, M. Valluri, and L. K. John. Effective Adaptive Computing Environment Management via Dynamic Optimization. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, Mar. 2005.

[74] M. Huang, J. Renau, and J. Torrellas. Profile-Based Energy Reduction in High-Performance Processors. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization*, December 2001.

[75] M. Huang, J. Renau, and J. Torrellas. Positional Adaptation of Processors: Application to Energy Reduction. In *Proceedings of the International Symp. on Computer Architecture*, 2003.

[76] C. Hughes, J. Srinivasan, and S. Adve. Saving energy with architectural and frequency adaptations for multimedia applications. In *Proceedings of the 34th Annual International Symposium on Microarchitecture (MICRO-34)*, Dec. 2001.

[77] IBM. PMAPI structure and function Reference. http://www16.boulder.ibm.com/pseries/en_US/files/aixfiles/pmapi.h.htm.

[78] Intel Corporation. *VTune^{TM} Performance Analyzer 1.1*. http://developer.intel.com/software/products/vtune/vlin/.

[79] Intel Corporation. Intel Pentium 4 and Intel Xeon Processor Optimization Reference Manual, 2002. http://developer.intel.com/design/Pentium4/manuals/248966.htm.

[80] Intel Corporation. *Intel Pentium 4 Processor in the 423 pin package / Intel 850 chipset platform*, 2002. http://developer.intel.com/design/chipsets/designex/298245.htm.

[81] Intel Corporation. Intel 64 and IA-32 Architectures Software Developers Manual, Volume 3B: System Programming Guide, 2006.

[82] C. Isci, G. Contreras, and M. Martonosi. Hardware Performance Counters for Detailed Runtime Power and Thermal Estimations: Experiences and Proposals. In *Proceedings of the Hardware Performance Monitor Design and Functionality Workshop in the 11th International Symposium on High-Performance Computer Architecture (HPCA-11)*, Feb. 2005.

[83] C. Isci, G. Contreras, and M. Martonosi. Live, Runtime Phase Monitoring and Prediction on Real Systems with Application to Dynamic Power Management. In *Proceedings of the 39th ACM/IEEE International Symposium on Microarchitecture (MICRO-39)*, 2006.

[84] C. Isci and M. Martonosi. Identifying Program Power Phase Behavior using Power Vectors. In *Proceedings of the IEEE International Workshop on Workload Characterization (WWC-6)*, 2003.

[85] C. Isci and M. Martonosi. Runtime Power Monitoring in High-End Processors: Methodology and Empirical Data. In *Proceedings of the 36th International Symp. on Microarchitecture*, Dec. 2003.

[86] C. Isci and M. Martonosi. Runtime Power Monitoring in High-End Processors: Methodology and Empirical Data. Technical report, Princeton University Electrical Eng. Dept., Sep 2003.

[87] C. Isci and M. Martonosi. Detecting Recurrent Phase Behavior under Real-System Variability. In *Proceedings of the IEEE International Symposium on Workload Characterization*, Oct. 2005.

[88] C. Isci and M. Martonosi. Phase Characterization for Power: Evaluating Control-Flow-Based and Event-Counter-Based Techniques. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA-12)*, 2006.

[89] C. Isci, M. Martonosi, and A. Buyuktosunoglu. Long-term Workload Phases: Duration Predictions and Applications to DVFS. *IEEE Micro: Special Issue on Energy Efficient Design*, 25(5):39–51, Sep/Oct 2005.

[90] A. Iyer and D. Marculescu. Power aware microarchitecture resource scaling. In *Proceedings of Design Automation and Test in Europe, DATE*, Mar. 2001.

[91] R. Jenkins. Hash functions. *Dr. Dobb's Journal*, 9709, Sept. 1997.

[92] R. Joseph, D. Brooks, and M. Martonosi. Control techniques to eliminate voltage emergencies in high performance processors. In *Proc. of the 9th International Symposium on High Performance Computer Architecture (HPCA-9)*, February 2003.

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

[94] P. Juang, Q. Wu, L.-S. Peh, M. Martonosi, and D. Clark. Coordinated, Distributed, Formal Energy Management of Chip Multiprocessors. In *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED'05)*, Aug. 2005.

[95] I. Kadayif, T. Chinoda, M. T. Kandemir, N. Vijaykrishnan, M. J. Irwin, and A. Sivasubramaniam. vEC: virtual energy counters. In *Workshop on Program Analysis for Software Tools and Engineering*, pages 28–31, 2001.

[96] S. Kaxiras, Z. Hu, and M. Martonosi. Cache decay: Exploiting generational behavior to reduce cache leakage power. In *Proceedings of the 28th International Symposium on Computer Architecture (ISCA-28)*, June 2001.

[97] A. Keshavarzi, S. Ma, S. Narendra, B. Bloechel, K. Mistry, T. Ghani, S. Borkar, and V. De. Effectiveness of Reverse Body Bias for Leakage Control in Scaled Dual Vt CMOS ICs. In *Proceedings of the 2001 International Symposium on Low Power Electronics and Design (ISLPED)*, Aug. 2001.

[98] C. H. Kim and K. Roy. Dynamic Vth Scaling Scheme for Active Leakage Power Reduction. In *Proceedings of the conference on Design, automation and test in Europe (DATE'02)*, Mar. 2002.

[99] J. Kim, S. V. Kodakara, W.-C. Hsu, D. J. Lilja, and P.-C. Yew. Dynamic Code Region (DCR) Based Program Phase Tracking and Prediction for Dynamic Optimizations. *Lecture Notes in Computer Science*, 3793:203–217, 2005.

[100] T. Kistler and M. Franz. Continuous Program Pptimization: A Case Study. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(4):500–548, 2003.

[101] A. KleinOsowski, J. Flynn, N. Meares, and D. J. Lilja. Adapting the SPEC2000 benchmark suite for simulation-based computer architecture research. In *Workshop on Workload Characterization, International Conference on Computer Design*, Sept. 2000.

[102] U. Kremer, J. Hicks, and J. Rehg. Compiler-Directed Remote Task Execution for Power Management. In *Proceedings of the Workshop on Compilers and Operating Systems for Low Power (COLP'00)*, 2000.

[103] R. Kumar, K. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *Proceedings of the 36th International Symp. on Microarchitecture*, Dec. 2003.

[104] E. Kursun, S. Ghiasi, and M. Sarrafzadeh. Transistor Level Budgeting for Power Optimization. In *Proceedings of the 5th International Symposium on Quality Electronic Design (ISQED'05)*, 2004.

[105] P. E. Landman. High-level power estimation. In *Proceedings of the 1996 International Symposium on Low Power Electronics and Design (ISLPED)*, Oct. 1996.

[106] P. E. Landman and J. M. Rabaey. Black-box Capacitance Models for Architectural Power Analysis. In *Proceedings of the International Workshop on Low Power Design*, Apr. 1994.

[107] P. E. Landman and J. M. Rabaey. Activity-sensitive Architectural Power Analysis for the Control Path. In *Proceedings of the International Workshop on Low Power Design*, Apr. 1995.

[108] J. Lau, J. Sampson, E. Perelman, G. Hamerly, and B. Calder. The Strong Correlation between Code Signatures and Performance. In *IEEE International Symposium on Performance Analysis of Systems and Software*, Mar. 2005.

[109] J. Lau, S. Schoenmackers, and B. Calder. Transition Phase Classification and Prediction. In *11th International Symposium on High Performance Computer Architecture*, 2005.

[110] A. R. Lebeck, X. Fan, H. Zeng, and C. Ellis. Power Aware Page Allocation. *ACM SIGOPS Operating Systems Review*, 34(5):105 – 116, Dec. 2000.

[111] B. Lee and D. Brooks. Accurate and Efficient Regression Modeling for Microarchitectural Performance and Power Prediction. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII)*, October 2006.

[112] K. Lee and K. Skadron. Using Performance Counters for Runtime Temperature Sensing in High-Performance Processors. In *Workshop on High-Performance, Power-Aware Computing*, 2005.

[113] S. Lee, A. Ermedahl, S. L. Min, and N. Chang. An accurate instruction-level energy consumption model for embedded RISC processors. In *LCTES/OM*, pages 1–10, 2001.

[114] S. Lee and T. Sakurai. Run-time Voltage Hopping for Low-power Real-time Systems. In *Proceedings of the* 37^{th} *Design Automation Conference (DAC'00)*, 2000.

[115] J. Li and J. Martinez. Dynamic Power-Performance Adaptation of Parallel Computation on Chip Multiprocessors. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA-12)*, 2006.

[116] T. Li and L. K. John. Run-time Modeling and Estimation of Operating System Power Consumption. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2003.

[117] X. Li, Z. Li, F. David, P. Zhou, Y. Zhou, S. Adve, and S. Kumar. Performance Directed Energy Management for Main Memory and Disks. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XI)*, 2004.

[118] M. Liu, W.-S. Wang, and M. Orshansky. Leakage Power Reduction by Dual-Vth Designs Under Probabilistic Analysis of Vth Variation. In *Proceedings of the 2004 International Symposium on Low Power Electronics and Design (ISLPED)*, Aug. 2004.

[119] J. R. Lorch and A. J. Smith. Improving Dynamic Voltage Scaling Algorithms with PACE. In *Proceedings of the 2001 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, 2001.

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

[121] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Programming Language Design and Implementation (PLDI)*, June 2005.

[122] G. Magklis, M. Scott, G. Semeraro, D. Albonesi, and S. Dropsho. Profile-based Dynamic Voltage and Frequency Scaling for a Multiple Clock Domain Microprocessor. In *Proceedings of the 30th International Symposium on Computer Architecture (ISCA-30)*, 2003.

[123] S. Manne, A. Klauser, and D. Grunwald. Pipeline gating: Speculation control for energy reduction. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 132–41, June 1998.

[124] C. McNairy and R. Bhatia. Montecito: A Dual-Core, Dual-Thread Itanium Processor. *IEEE Micro*, 25(2):10–20, Mar/Apr 2005.

[125] H. Mehta, R. M. Owens, and M. J. Irwin. Energy characterization based on clustering. In *Proceedings of the* 33^{rd} *Design Automation Conference (DAC'96)*, 1996.

[126] H. Mehta, R. M. Owens, and M. J. Irwin. Instruction Level Power Profiling. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP'96)*, May 1996.

[127] A. Merkel. *Balancing Power Consumption in Multiprocessor Systems*. PhD thesis, Sept. 2005. System Architecture Group, University of Karlsruhe, Diploma Thesis.

[128] M. C. Merten, A. R. Trick, R. D. Barnes, E. M. Nystrom, C. N. George, J. C. Gyllenhaal, and W. mei W. Hwu. An architectural framework for runtime optimization. *IEEE Transactions on Computers*, 50(6):567–589, 2001.

[129] G. E. Moore. Cramming more components onto integrated circuits. In *Electronics*, pages 114–117, Apr. 1965.

[130] J. Moore, J. Chase, P. Ranganathan, and R. Sharma. Making scheduling cool: Temperature-aware workload placement in data centers. In *Proceedings of USENIX '05*, June 2005.

[131] P. Nagpurkar, C. Krintz, M. Hind, P. Sweeney, and V. Rajan. Online Phase Detection Algorithms. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, Mar. 2006.

[132] P. Nagpurkar, C. Krintz, and T. Sherwood. Phase-Aware Remote Profiling. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, Mar. 2005.

[133] K. Olukotun and L. Hammond. The Future of Microprocessors. *ACM Queue*, 3(7):27–34, Sept. 2005.

[134] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K.-Y. Chang. The Case for a Single-Chip Multiprocessor. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, Oct. 1996.

[135] H. H. Padmanabhan. Design and Implementation of Power-aware Virtual Memory. In *Proceedings of USENIX*, 2003.

[136] V. Pandey, W. Jiang, Y. Zhou, and R. Bianchini. DMA-Aware Memory Energy Management. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA-12)*, Feb. 2006.

[137] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing Representative Portions of Large Intel Itanium Programs with Dynamic Instrumentation. In *Proceedings of the 37th International Symp. on Microarchitecture*, 2004.

[138] C. Poirier, R. McGowen, C. Bostak, and S. Naffziger. Power and Temperature Control on a 90nm Itanium-Family Processor. In *IEEE International Solid-State Circuits Conference (ISSCC 2005)*, Feb. 2005.

[139] D. Ponomarev, G. Kucuk, and K. Ghose. Reducing Power Requirements of Instruction Scheduling Through Dynamic Allocation of Multiple Datapath Resources. In *Proceedings of the 34th Annual International Symposium on Microarchitecture (MICRO-34)*, Dec. 2001.

[140] M. Powell, S.-H. Yang, B. Falsafi, K. Roy, and T. N. Vijaykumar. Reducing Leakage in a High-Performance Deep-Submicron Instruction Cache. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(1):77–90, 2001.

[141] R. Ronen, A. Mendelson, K. Lai, S.-L. Lu, F. Pollack, and J. P. Shen. Coming Challenges in Microarchitecture and Architecture. *Proceedings of the IEEE*, 89(3):325–340, Mar. 2001.

[142] J. Russell and M. Jacome. Software power estimation and optimization for high performance, 32-bit embedded processors. In *Proceedings of the International Conference on Computer Design*, October 1998.

[143] D. G. Sachs, W. Yuan, C. J. Hughes, A. Harris, S. V. Adve, D. L. Jones, R. H. Kravets, and K. Nahrstedt. Grace: A hierarchical adaptation framework for saving energy. Technical report, Computer Science, University of Illinois Technical Report UIUCDCS-R-2004-2409, 2004.

[144] N. Sakran, M. Yuffe, M. Mehalel, J. Doweck, E. Knoll, and A. Kovacs. Implementation of the 65nm Dual-Core 64b Merom Processor. In *IEEE International Solid-State Circuits Conference (ISSCC 2007)*, Feb. 2007.

[145] T. Sato, M. Nagamatsu, and H. Tago. Power and Performance Simulator: ESP and Its Applications for 100 MIPS/W Class RISC Design. In *Proceedings of the 1994 International Symposium on Low Power Electronics and Design (ISLPED)*, Oct. 1994.

[146] R. Schmidt. Liquid Cooling is Back. *Electronics Cooling*, 11(3), Aug. 2005.

[147] G. Semeraro, G. Magklis, R. Balasubramonian, D. Albonesi, S. Dwarkadas, and M. Scott. Energy-Efficient Processor Design Using Multiple Clock Domains with Dynamic Voltage and Frequency Scaling. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture (HPCA-8)*, 2002.

[148] J. S. Seng and D. M. Tullsen. The effect of compiler optimizations on Pentium 4 power consumption. In *7th Annual Workshop on Interaction between Compilers and Computer Architectures*, Feb. 2003.

[149] Server System Infrastructure (SSI) consortium. Power Supply Management Interface Design Guide, Rev. 2.12, Sept. 2005.

[150] X. Shen, Y. Zhong, and C. Ding. Locality Phase Prediction. In *Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*, Oct. 2004.

[151] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2001.

[152] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. In Tenth International Conference on Architectural Support for Programming Languages and Operating Systems, Oct 2002.

[153] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Proceedings of the 28th International Symposium on Computer Architecture (ISCA-30)*, June 2003.

[154] D. Shin, J. Kim, and S. Lee. Low-Energy Intra-Task Voltage Scheduling Using Static Timing Analysis. In *Proceedings of the* 38^{th} *Design Automation Conference (DAC'01)*, June 2001.

[155] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-aware microarchitecture. In *Proceedings of the 30th International Symposium on Computer Architecture*, June 2003.

[156] L. Spracklen and S. G. Abraham. Chip Multithreading: Opportunities and Challenges. In *11th International Symposium on High Performance Computer Architecture (HPCA-11)*, 2005.

[157] E. Sprangle and D. Carmean. Increasing Processor Performance by Implementing Deeper Pipelines. In *Proceedings of the 29th International Symposium on Computer Architecture (ISCA-29)*, May 2002.

[158] B. Sprunt. *Brink and Abyss Pentium 4 Performance Counter Tools For Linux*, Feb. 2002. http://www.eg.bucknell.edu/bsprunt/emon/brink_abyss/brink_abyss.shtm.

[159] B. Sprunt. Pentium 4 Performance-Monitoring Features. *IEEE Micro*, 22(4):72–82, Jul/Aug 2002.

[160] B. Sprunt. Managing The Complexity Of Performance Monitoring Hardware: The Brink and Abyss Approach. *International Journal of High Performance Computing Applications*, 20(4):533–540, 2006.

[161] A. Srivastava and D. Sylvester. Minimizing Total Power by Simultaneous Vdd/Vth Assignment. In *ASPDAC: Proceedings of the 2003 conference on Asia South Pacific design automation*, Jan. 2003.

[162] P. Stanley-Marbell, M. S. Hsiao, and U. Kremer. A Hardware Architecture for Dynamic Performance and Energy Adaptation. In *Proceedings of the Workshop on Power-Aware Computer Systems*, 2002.

[163] D. Talkin. *A robust algorithm for pitch tracking (RAPT). Speech Coding and Synthesis*. Elsevier Science B. V., New York, 1995.

[164] T. K. Tan, A. Raghunathan, and N. K. Jha. Software Architectural Transformations: A New Approach to Low Energy Embedded Software. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE'03)*, Mar. 2003.

[165] The Standard Performance Evaluation Corporation. SPEC CPU2000 Results. http://www.spec.org/cpu2000/results/.

[166] The Standard Performance Evaluation Corporation. SPEC CPU2000 Suite. http://www.specbench.org/osg/cpu2000/.

[167] G. Theocharous, S. Mannor, N. Shah, P. Gandhi, B. Kveton, S. Siddiqi, and C.-H. Yu. Machine Learning for Adaptive Power Management. *Intel Technology journal*, 10(4):299–311, 2006.

[168] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: A first step towards software power minimization. *IEEE Transactions on VLSI Systems*, 2(4):437–445, December 1994.

[169] R. Todi. Speclite: using representative samples to reduce spec cpu2000 workload. In *Proceedings of the IEEE International Workshop on Workload Characterization (WWC-4)*, 2001.

[170] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd International Symp. on Computer Architecture*, pages 392–403, June 1995.

[171] United States Environmental Protection Agency. ENERGY STAR Program Requirements for Computers, Version 4.0. Oct. 2006.

[172] P. Unnikrishnan, G. Chen, M. Kandemir, and D. R. Mudgett. Dynamic Compilation for Energy Adaptation. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design (ICCAD)*, 2002.

[173] O. Unsal and I. Koren. System-Level Power-Aware Design Techniques in Real-Time Systems. *Proceedings of the IEEE*, 91(7), July 2003.

[174] V. Venkatachalam and M. Franz. Power Reduction Techniques for Microprocessor Systems. *ACM Computing Surveys (CSUR)*, 37(3):195–237, 2005.

[175] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. S. Kim, and W. Ye. Energy-Driven Integrated Hardware-Software Optimizations Using SimplePower. In *Proceedings of the 27th International Symposium on Computer Architecture*, June 2000.

[176] A. Weissel and F. Bellosa. Process cruise control: Event-driven clock scaling for dynamic power management. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES 2002), Grenoble, France,*, Aug. 2002.

[177] A. Weissel, B. Beutel, and F. Bellosa. Cooperative I/O-A Novel I/O Semantics for Energy-Aware Applications. In *Proceedings of the Fifth Symposium on Operating System Design and Implementation OSDI'02*, 2002.

[178] Q. Wu, P. Juang, M. Martonosi, and D. W. Clark. Voltage and Frequency Control with Adaptive Reaction Time in Multiple-Clock-Domain Processors. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA-11)*, 2005.

[179] Q. Wu, V. Reddi, Y. Wu, J. Lee, D. Connors, D. Brooks, M. Martonosi, and D. W. Clark. A Dynamic Compilation Framework for Controlling Microprocessor Energy and Performance. In *Proceedings of the 38th International Symp. on Microarchitecture*, 2005.

[180] F. Xie, M. Martonosi, and S. Malik. Compile-Time Dynamic Voltage Scaling Settings: Opportunities and Limits. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2003)*, June 2003.

[181] T. Y. Yeh and Y. N. Patt. Alternative implementations of two-level adaptive branch prediction. In *19th Annual International Symposium on Computer Architecture*, May 1992.

[182] J. J. Yi, D. J. Lilja, and D. M. Hawkins. A Statistically Rigorous Approach for Improving Simulation Methodology. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA-9)*, Feb. 2003.

[183] J. J. Yi, R. Sendag, L. Eeckhout, A. Joshi, D. J. Lilja, and L. K. John. Evaluating Benchmark Subsetting Approaches. In *Proceedings of the IEEE International Symposium on Workload Characterization*, Oct. 2006.

[184] H. Zeng, X. Fan, C. Ellis, A. Lebeck, and A. Vahdat. ECOSystem: Managing energy as a first class operating system resource. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, Oct. 2002.

[185] M. T. Zhang. Powering Intel Pentium 4 generation processors. In *IEEE Electrical Performance of Electronic Packaging Conference*, pages 215–218, 2001.

[186] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic Tracking of Page Miss Ratio Curve for Memory Management. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XI)*, 2004.