Architectural and Compiler Techniques for Microprocessor Power and Performance Management

Qiang Wu

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science

September 2006

Abstract

As computing technology continues to progress very rapidly, many technical challenges emerge. One of these is the issue of power dissipation. Indeed, power delivery and dissipation are becoming primary limiters of performance and integration for microprocessors. In response, architectural and software level power-reduction techniques, which extend traditional circuit-level energy techniques, have gained more and more attention and become an active research area in the last few years.

The work in this thesis focuses on one important problem in this area, namely dynamic power and performance management in high-performance processors. Dynamic adaptive techniques are appealing because they offer the ability to adjust on the fly according to the current run-time power and performance situation. This thesis investigates architectural and compiler techniques for controlling power and performance in microprocessors. The overall contributions of this work are the proposed new concepts, methods, and framework for intelligent power and performance management.

Specifically, this work has had two major thrusts. First, formal control-theoretic techniques will be discussed in the context of hardware-based energy control. The environment is a multiple clock domain processor. An analytical system model is first proposed that describes relationships among performance demand, capability, and clock frequency. A controller is then designed to balance the speeds of different clock islands. Experimental results show that the proposed technique is 2-3 times more efficient in terms of energy delay product improvement, compared to a previous heuristic approach. In addition, the new technique is more robust with a guaranteed stability margin even under extreme cases. For the above design, both fixed-interval and adaptive interval control schemes have been investigated. Second, software-layer energy control opportunities are explored in a general dynamic compilation system. It is shown that a dynamic compiler driven scheme has several unique features and advantages over existing energy control schemes. Such a scheme is then designed, implemented, and deployed on real hardware (with a Pentium-M processor). Experimental results from physical power measurements show up to 70% energy saving is accomplished for SPEC benchmarks. In addition, because of its orthogonal features and advantages, the dynamic compiler driven scheme can be an effective complement to existing hardware-based energy control schemes.

Acknowledgements

The work presented in this thesis has been possible only through the contributions of many wonderful people.

I wish to express my deep thankfulness to my thesis advisors, Doug Clark and Margaret Martonosi. I thank Doug for his thoughtful direction that helped the thesis take shape, and his endless guidance that sustained the thesis as it moved forward. Throughout these years, he has been always available, helped me keep everything in perspective, and managed to keep my spirits up in crises. I thank Margaret for her invaluable advice and support throughout this thesis work. She has not only helped me with my research, but also taught me how to develop myself intellectually and believe in my ideas and myself. I am privileged to have learned from Doug and Margaret. The interaction with them has been one of the best experiences of my professional life.

Many thanks are extended to David August, Kai Li, and Youfeng Wu. David introduced me to the field of computer architecture and compilers at Princeton. I appreciate greatly his help. Kai has contributed significantly to my intellectual development. I am very grateful for his support and help in the thesis work and his advice on my career development. Youfeng has been my industrial mentor since my first year at Princeton. Throughout these years, he has constantly supported my work. I am indebted to him for his encouragement and support along the way.

Many thanks to Andrea LaPaugh, David Brooks, and Dan Connors for their helpful and constructive suggestions. Special thanks to Melissa Lawson and CS Staff for all their help during my study at Princeton. Big thanks to my fellow students in the CS and EE departments — Philo Juang, Gilberto Contreras, Canturk Isci, James Donald, Yong Wang, Chris Sadler, Eric Chi, Fen Xie, Pei Zhang, David Penry, Ram Rangan, Bolei Guo, Guilherme Ottoni, and Easwaran Raman. Your friendship that accompanied my study will always be cherished.

Financial support provided by the Intel foundation graduate fellowship and National science foundation grants (ITR CCR-0086031 and CNS-0410937) is greatly acknowledged.

Last but not least, I thank my wife and our parents for their enduring love and support all these years. My wife, Hong, and I have shared all the ups and downs of our lives together since we first met as teenagers. It would have been impossible for me to finish this thesis without her.

Contents

Chapter 1

Introduction

1.1 Background and Motivation

Computing technology has progressed rapidly in the past 20 years due to two major reasons. First, advances in circuit integration technology make it possible to put smaller and faster transistors into a chip and double the scale of integration roughly every 18 months. Second, computer architects and compiler designers make use of the increasing number of transistors available in a chip to exploit parallelism and further increase the computing capability through techniques like deeper pipeline, superscalar, and VLIW [29]. As a result, we have seen an exponentially increasing computing performance in the past two decades, as illustrated by Figure 1.1 for Intel microprocessors [54].

However, with the increasing computing performance come also exponentially increasing design complexity and processor power consumption. Figure 1.2 gives an illustration for power consumptions of Intel processors [54]. There are several serious problems with the increasing power. First, higher power dissipation means more energy consumption in general. For portable computing devices, this means shorter battery life, as battery technology is not improving at the same rate. For desktop and server computers, this means more electricity cost. For example, Google has noted that their electricity bill is too high and is

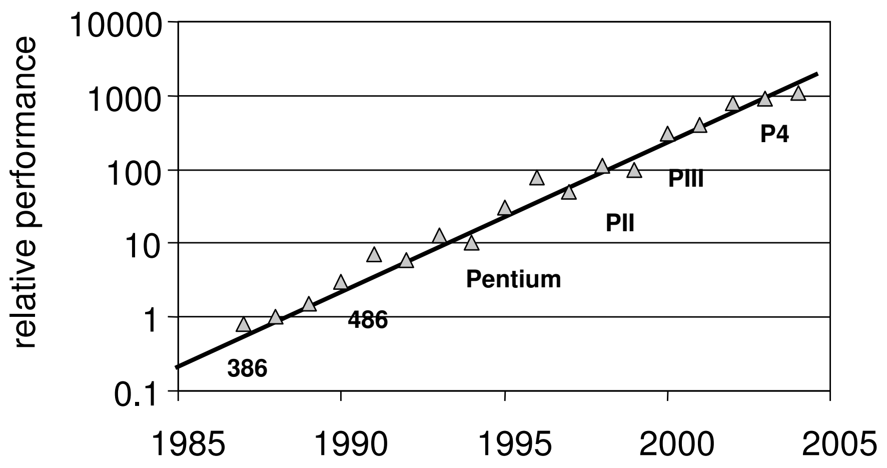

Figure 1.1: Relative performance over time for Intel's processors.

getting higher [7].

A second problem is thermal dissipation. We have seen the trend of needing no heat sink for processors in the 1980s, to moderate-size heat sinks for processors in the 1990s, to today's giant heat sinks. With the power consumption moving crossing the line of 100 watts, the conventional air cooling system has been pushed toward its limit. With the current trend of power increasing, it is possible that we will be forced to use more expensive cooling systems like water or hydrogen-based, which will increase the overall system cost and become impractical for lower-end computing systems. A third problem is that it is becoming more difficult to maintain the quality of power supply in a processor. With more current flowing, maintaining a stable and high-quality supply voltage becomes more challenging due to issues like inductive noise or dI/dt [38]. Indeed, the power and thermal problem is becoming a primary limiter of performance and integration, and has been regarded as one of the grand challenges in current and future processor design.

In response to the challenge, significant research efforts have been devoted to power-efficient computing. While traditional efforts in this direction focus on circuit level techniques [16], more and more attentions have been paid to techniques at architecture or system software level in the past few years [2, 10, 12, 24, 31, 32, 34, 36, 37, 38, 45, 46, 48, 49, 58, 68]. Numerous power-aware architecture/compiler/OS design, analysis, and manage-

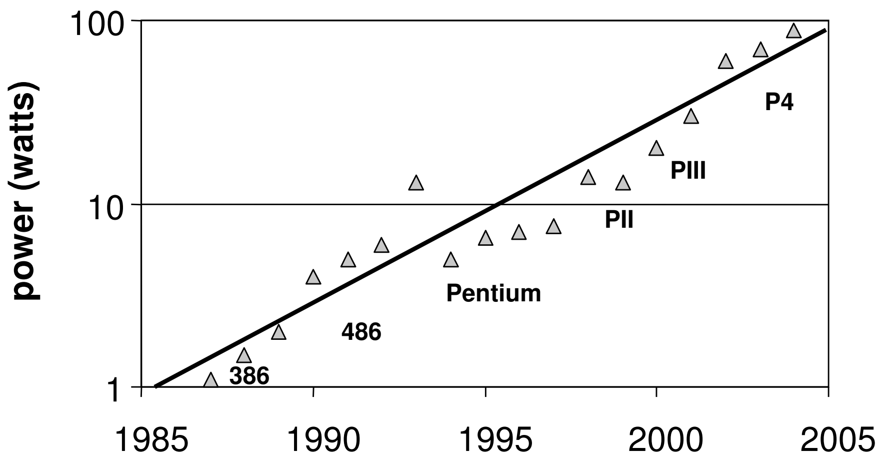

Figure 1.2: Power consumption over time for Intel's processors.

ment techniques have been proposed and studied in literature. For example, we have seen architectural level power modeling [10], dynamic thermal management [9], adaptively configurable function units or cache structure [2, 14, 32], system-level voltage and frequency scaling [12, 58], power-aware compilation and instruction selection [39], and power-aware paging and swapping [42].

The work in this thesis focuses on one important problem in this area, namely dynamic power and performance management in high-performance processors. Dynamic adaptive management is crucial in modern computers because of the workload variability and the unpredictability of moment-to-moment performance and power demand in real computer systems. With dynamic adaptive techniques, the system can adjust on the fly according to the current run-time power and performance situation, rather than the worst-case estimates.

1.2 Related Work

In this section, I give a high-level discussion and classification of existing work. Later, in each chapter, I give more detailed discussions of the work related and specific to each of the chapters.

Existing work in the area of dynamic power and performance management can be categorized in several dimensions. The first dimension is the level of control hierarchy: ranging from OS-based, compiler-based, to architecture or hardware-based. Current practice for power management is OS-based and is relatively coarse-grained. As implemented in the latest operating systems (such as Windows XP or Linux 2.6), power and performance are managed through a set of operating states – active, idle, deep sleep, etc. The OS chooses an appropriate operating state according to user specifications, current workload, and the battery or power supply condition. Besides these OS approaches in real systems, more advanced OS-level techniques have been proposed in literature, such as power-aware scheduling [63] in which the OS selects an appropriate voltage/frequency setting when a new application or task is scheduled. At the compiler level, a number of techniques have been proposed using a static compiler or binary re-writer as the tool for power and performance management [31, 68]. They typically use profiling and offline analysis to choose appropriate power/performance settings. At the architectural or hardware level, numerous techniques have been proposed to adaptively control the execution for power and performance [19, 49, 58]. They typically employ some online control mechanisms to monitor the system condition and adjust the system parameters for better performance and power efficiency.

A second dimension is the control granularity relative to the application or task. On one hand, the inter-task schemes only consider possible adaptation points between different applications [63]. On the other hand, the intra-task schemes strive to take advantage of the program phase changes inside an application, and thus are more fine-grained [32, 58]. In general, most OS-level techniques are inter-task, while most compiler or architectural level techniques are intra-task. A third dimension is based on different control means employed in the scheme. Dynamic voltage and frequency scaling (DVFS) is one popular control method, which scales down voltage and frequency at run time based on the current power and performance requirement [12, 65]. Another popular method is adaptively configurable execution units such as resizable issue queues [14] or cache [2]. A fourth dimension is on different target devices in a computer system. There are many research ideas for various processors from low-end to high-end [17, 25, 40]. There are also quite a few works focusing on memory [43] and disk [26, 28].

Relative to the above classification, the work in this thesis focuses on architectural and compiler level techniques for processor adaptation. Because it is so prevalent, DVFS is used as the form of energy control. But I believe the techniques and methodology described in this thesis should be applicable for other control means as well. In addition, the focus is on intra-task DVFS as it is more fine-grained and more aggressive in managing power, compared to inter-task DVFS.

1.3 Thesis Overview and Contributions

The work in this thesis investigates architectural and compiler techniques for controlling power and performance in high-performance processors. At a high level, the features of this work are three-fold. First, it is one of the first efforts to use a rigorous, analytical approach in architectural level energy control. Second, it is one of the first research attempts to explore energy control opportunities in system software, especially in a dynamic compiler. Third, it develops a real system and deploys the system in real hardware with physical power measurements.

The main materials are presented in Chapters 2-4 of this thesis. An overview of them is as follows.

Chapter 2. Formal online methods for voltage and frequency control in multiple-clock-domain processors

Most existing architectural level hardware-based energy control schemes take an *ad hoc* approach and employ a set of heuristics. However, there are several problems associated with the heuristic-based approaches. First, the trial-and-error tuning process for heuristic parameters is very time consuming. Second, with heuristic parameters, there are concerns about worst-case control behavior and the control effectiveness on atypical applications. Third, there is a scalability problem for control in a large system or a system with multiple control loops, as it becomes extremely difficult to pick good heuristic rules and parameters to handle a large number of interacting factors.

In this part of the thesis, I have investigated and designed formal control-theoretic techniques for microprocessor voltage and frequency control. Control theory is used to formally reason about what will happen and how to improve the control effectiveness and robustness. Specifically, this work is carried on in the context of a multiple-clock-domain (MCD) processor, which is a novel power-efficient design recently proposed in literature. I first model the MCD processor as a queue-domain network and the DVFS problem as a feedback control problem with the issue queue occupancies as feedback signals. After an analytical system model is derived, I design a PID-based (Proportional, Integral, and Derivative) control and verify it via stability analysis. Finally, the proposed DVFS scheme is evaluated by a cycle-accurate architecture simulator with a broad set of applications. Compared to the best-known DVFS scheme for MCD, which is heuristic-based, the proposed scheme is two times more efficient in term of energy delay product (EDP) improvement.

The primary contributions of this part of the thesis are as follows. First, it focuses on voltage/frequency control for MCD architectures, which are relatively new and have great potential in terms of energy-saving and performance improvement. Second, the proposed control-theoretic techniques applied to DVFS in MCD processors have led to a 2-3 fold increase in efficiency compared to the best-known previous heuristic-based DVFS scheme in MCD processors. In addition, the control-theoretic technique is more resilient, complete, and boundable as well. For example, it can guarantee stability, achieve significant energy savings, and offer more graceful degradation even under extreme cases. Third, this is one of the first rigorous analytic approaches to DVFS control. Previous control theoretic techniques exist [46], but only for multimedia processors with predictable workloads, while this work is for general workloads.

Chapter 3. A novel energy control scheme with adaptive reaction time

Most existing hardware-based DVFS schemes (including the one in Chapter 2) use a fixed time intervals between possible voltage/frequency changes. The downside to their approach is that the interval boundaries are predetermined and independent of workload changes. Thus, they can be delayed in responding to large, severe activity swings.

In Chapter 3, I propose a novel DVFS scheme for MCD processor, in which the control reaction time is self-tuned and adaptive to application and workload changes. As part of the scheme, I have designed a DVFS controller that has a simple decision process and reacts directly to recent queue occupancy conditions. In addition to designing such a scheme, I model the proposed DVFS controller and use the derived model in a formal stability analysis. The obtained analytic insight is then used to guide and improve the design in terms of stability margin and control effectiveness. Experimental results show that the new and simpler scheme has achieved significant energy savings over all studied benchmarks: 19% energy savings with 3% performance loss on average. This is close to the best results from existing fixed-interval DVFS schemes.

The primary contributions of Chapter 3 are follows. First, rigorous modeling and stability analysis techniques have been applied to the design to provide insight and guidance, and make the design more efficient and more resilient. Second, the proposed interval-less control scheme has a simpler decision process and better responsiveness than existing fixed-interval schemes. A simpler decision process means smaller and cheaper hardware. So this scheme is useful for processors with limited hardware budgets. In addition, for a group of applications with fast workload variations, the new scheme outperforms existing schemes by 18% or more due to its self-tuning nature and responsiveness. So this scheme is also useful and suitable to processors where the type of application behavior (i.e. rapid workload variations) is known in advance.

Chapter 4. A dynamic compilation framework for controlling microprocessor energy and performance

In addition to hardware, the compiler and system software can also play an important role in energy and performance management. While existing software-driven energy control techniques are primarily based on OS time-interrupt or static compiler, very little has been done to explore control opportunities in a general dynamic compilation system. There are, however, several unique features and advantages to deploying DVFS through the use of a dynamic compiler. Most importantly, dynamic compiler driven DVFS is fine-grained, code-aware, and adaptive to the current microarchitecture environment, partially because of its ability to infer about program code attributes as well as to access run-time hardware information.

Chapter 4 presents a design framework of the run-time DVFS optimizer in a general dynamic compilation system. A prototype of the DVFS optimizer is implemented and integrated into an industrial-strength dynamic compilation system. The obtained system is deployed in a real hardware platform that directly measures CPU voltage and current for accurate power and energy readings. Experimental results, based on physical measurements for over 40 SPEC or Olden benchmarks, show that significant energy savings achieved with little performance degradation. For example, SPEC benchmarks benefit with energy savings of up to 70% (with 0.5% performance loss), and Olden benchmarks save up to 61% energy (with 4.5% performance loss). In addition, because of its orthogonal features and advantages, the dynamic compiler driven scheme can be an effective complement to existing hardware-based energy control schemes.

Chapter 4 has two main contributions. First, I have designed and implemented a run-time DVFS optimizer, and deployed it on real hardware with physical power measurements. The optimization system is more effective in terms of energy and performance efficiency, as compared to existing approaches. Second, to my knowledge, this is one of the first efforts to develop dynamic compiler techniques for microprocessor voltage and frequency control. A previous work [27] provides a partial solution to the Java Method DVFS in a Java Virtual Machine, while this work provides a complete design framework and applies DVFS in a more general dynamic compilation environment with general applications.

Chapter 2

Formal Online Methods for Voltage and Frequency Control in Multiple Clock Domain Microprocessors

2.1 Introduction

The energy control work in this chapter is conducted in the context of multiple-clock-domain (MCD) processors [59]. An MCD processor constitutes a novel power-efficient architecture which uses the Globally Asynchronous and Locally Synchronous (GALS) clocking style [37, 50]. Inside an MCD, each function domain operates with an independent clock; a synchronization circuit or queue is used for inter-domain communication. A brief review of MCD is provided in Section 2.2.

The focus in this chapter is on hardware-based online DVFS schemes, as they are driven by dynamic workload behavior, and thus are more adaptive and applicable than static profile-based offline ones [48, 59]. The goal of this work is to design an *analytic* online DVFS scheme for an MCD processor. Nearly all existing hardware-based DVFS schemes are heuristic-based [36, 49, 58]. They typically include a set of manually selected rules and threshold values. At run time, certain processor metrics, such as cache miss rate [49] or queue occupancy [36, 58], are monitored. These metrics are then compared to the threshold values and one of the rules is applied depending on the result of the comparison. The best known DVFS scheme for an MCD processor is the AttackDecay algorithm by Semeraro *et al* [58]. However, there are significant limitations in heuristic-based schemes. First, the trial-and-error tuning process for parameters is very time consuming. Second, for a given set of rules and parameters, it is not analytically clear how to improve the control effectiveness and how to bound the worst-case control behavior. Third, it is generally hard to scale the heuristics for a large system or a system with multiple interacting control loops, as the number of rules and the tuning effort required can grow exponentially.

To overcome the above limitations, the DVFS scheme in this work takes a rigorous, analytical approach. I model an MCD processor as a queue-domain network, and formulate the DVFS problem as a feedback control problem with issue queue occupancies as feedback signals. Specifically, a stochastic model is proposed for the queue-domain dynamics. Since the queue-domain system is inherently nonlinear, it is first linearized through an accurate *feedback linearization*. The controller is then designed and verified by stability analysis. Next, a possible hardware implementation of the controller is described. Finally the proposed online DVFS scheme is evaluated by a cycle-accurate architecture simulator with a broad set of applications selected from the MediaBench and SPEC2000 benchmark suites. Overall, this work achieves a power-saving to performance degradation ratio of 6.2 (i.e. on average, 6.2% power is saved for 1% performance degradation), as compared to a ratio of 2.5 for the conventional synchronous voltage scaling.

Compared to the best known prior online DVFS approach for an MCD processor [58] which is heuristic-based, the proposed DVFS scheme has several significant advantages. First, the analytic DVFS scheme is more robust and more effective in terms of energy saving for the same level of performance degradation. Experimental results show an energy-delay product improvement 146% higher than that of the best-known heuristic-based online scheme in [58], and 11% higher than the semi-oracle-based DVFS scheme in [59]. I attribute this promising result to the automatic frequency/voltage regulation ability in the proposed DVFS controller, which leads to a more effective and precise decision on when, where, and how much to scale. In addition, the proposed analytic online DVFS scheme requires less tuning effort than heuristic-based DVFS as the analytic DVFS chooses its control parameters through stability analysis. Furthermore, the analytic DVFS scheme is more scalable; we do not have to re-tune parameters or re-set rules in order to include new resources. Also, we can extend it to a centralized or multi-modal DVFS scheme, which will handle interactions among multiple clock domains (as is discussed in Section 2.5).

The rest of the chapter is structured as follows. In Section 2.2, we give a brief review of the MCD processor design and implementation. Section 2.3 describes the modeling, design, and analysis of our online DVFS controller. This is followed in Section 2.4 by experimental results for the purpose of evaluation. In Section 2.5, I give a general outline of centralized DVFS design. Finally, Section 2.6 summarizes this chapter.

2.2 Background: Multiple Clock Domain Microprocessors

Some computer architects and researchers have predicted that, in order to overcome the increasing severe problems of clock distribution and power consumption, future high-performance microprocessors may need to have multiple clock domains (MCD) or use some form of asynchrony [3, 51]. MCD processors use the Globally Asynchronous Locally Synchronous (GALS) clocking style [50]. Each function block or domain operates with an independently generated clock, and synchronization circuits ensure reliable inter-domain communication.

Advantages of MCD processors include less clock distribution and skew burden, less power consumption due to the absence of a global clock tree, DVFS flexibility, and design modularity [50]. The primary disadvantage of MCD processors is the inter-domain

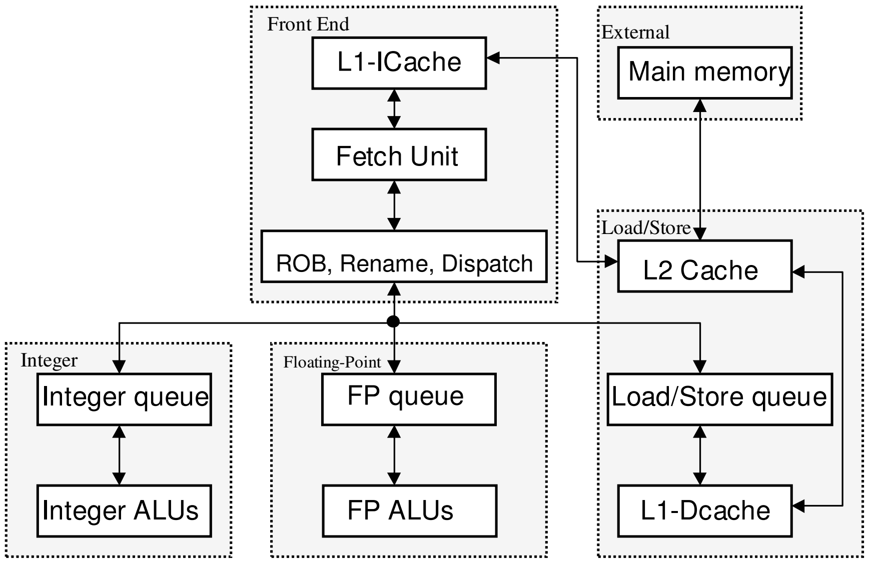

Figure 2.1: The clock domain partitions in an MCD processor by Semeraro *et al.*

communication and synchronization overhead. An interface circuit is needed if data passes between two domains.

One key design issue for an MCD processor is the choice of where to partition the clock domains. It is still an open research question on how to partition in order to maximize the power-performance benefit. Most existing MCD implementations use architectural functional blocks as *natural* boundaries for clock domains. For example, Figure 2.1 shows a 4-domain partition used by the MCD implementation by Semeraro *et al.* [59], which consists of the front end, integer processing core (INT), floating point processing core (FP), and load store unit (LS). The main memory is considered as an external separate clock domain not controlled by the processor (for more details see [59]). Another popular MCD implementation by Iyer and Marculescu [37] uses a 5-domain partition, which is similar to that in Figure 2.1 but with the front-end split into two clock domains.

Another key design issue for an MCD processor is the synchronization interface design. A good interface design needs to have low latency, high throughput, and virtually no synchronization failure (i.e. no metastability). Nearly all of the existing MCD interface designs use some kind of queue structures for efficiency. One group of designs [18] uses token-ring based FIFOs, which have a very low latency and low synchronization overhead. (There is no synchronization cost if the token-ring FIFO is neither full nor empty.) Another group of designs uses arbitration-based queue structures (often with a stoppable clock) [61, 69]. The designs in this group are typically failure-free, but may need to check synchronization for each data transfer. For example, the design by Sjogren and Myers [61] includes arbitration and synchronization circuits which can detect whether the source and destination clock edge are far enough apart (i.e., greater than the so-called synchronization window size in [58]), in order for the source generated signal to be successfully accessed at the destination. This design has been used by the MCD implementation in [59]. Note that for a situation like that in Figure 2.1, where issue queues already exist between some domains, the interface queue structure can be integrated with the existing issue queue to form a combined issue/interface queue structure.

The MCD implementations in [37] and [59] also provide the capability of independently configuring the frequency and voltage in each clock domain. An aggressive XScale-style DVFS model [20] is assumed, in which a clock domain can execute through the DVFS transition and the penalty is negligible due to a domain being idle waiting for the PLL [20]. Also, the XScale model allows any frequency to be used within the allowable range.

The quantitative benefit/overhead studies in [37, 59] have shown that an MCD processor has the potential to achieve significant power and performance efficiency. However, many design and control issues remain open and need more investigation in order to fully take advantage of the power/performance benefits brought by the MCD processors [50]. Online DVFS is one of these issues, as we will show next.

2.3 Online DVFS Design: An Analytic Approach

2.3.1 Problem formulation

Conceptually, the online DVFS problem for an MCD processor is to scale the frequency f to match the varying performance demand in each clock domain. In other words, we want to

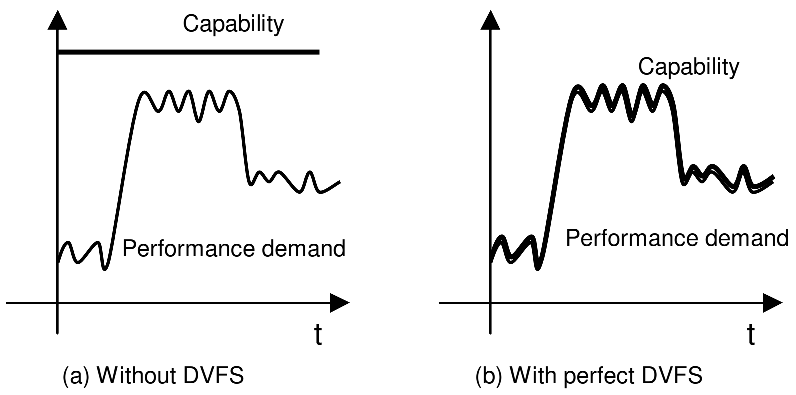

Figure 2.2: A conceptual illustration of a perfect online DVFS for a clock domain in an MCD processor.

adapt frequency to workload changes. Figure 2.2a depicts a typical scenario with varying demand or workload. For a clock domain, changing clock frequency will generally change the execution speed or capability of the domain. So, a perfect DVFS scheme will lead to a perfect match between the demand and domain execution capability, as illustrated in Figure 2.2b. In the figure, *perfect* is in the sense of no performance degradation and elimination of all energy wasted due to excessive capability slack. Thus, the goal of our online DVFS scheme is to get results as close as possible to that of perfect DVFS.

Note the excessive capability slack in Figure 2.2a is just the gap between the capability line and the demand line. We refer to the energy wasted due to this slack as E_{slack}. So for a *perfect* DVFS in Figure 2.2b, the energy savings will be all of the E_{slack}.

In addition, in Figure 2.2, we can continue to scale the execution capability line arbitrarily low below the performance need for non-realtime applications to get more energy savings, but that comes with a price of performance degradation, as the performance demand can not be satisfied. We call this part of energy savings *non-slack energy savings* or $E_{\text{non-slack}}$, to distinguish it from the "free" energy savings due to the elimination of E_{slack}.

In this paper, we want to make use of the queues in an MCD to guide DVFS. Recall from Section 2.2 that there are interface queues between clock domains for synchronization. Intuitively, these queues give clues about the speed balance between the sender do-

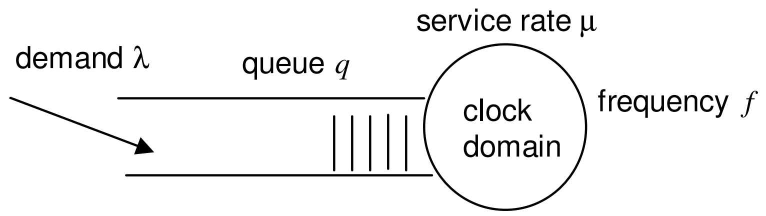

Figure 2.3: A single queue model for a clock domain with an input queue.

main and the receiver domain. For example, an emptying queue may mean the receiver is too fast relative to the sender; a filling queue may mean it is too slow; a stationary queue means a perfect match of receiver speed relative to the sender. This suggests a feedback control scheme for DVFS which uses the queue occupancy as a feedback signal to control the domain frequency, making its execution speed adaptive to varying demand. If the adaptiveness is fast enough, the DVFS scheme should get results close to the perfect matching in Figure 2.2.

There are, however, significant challenges to pursuing this idea along a rigorous analytic direction. First, to design a simple, hardware controller based on control theory, we need an analytical model for the involved queue and domain dynamics. Second, the queue-domain relationships are inherently time varying and nonlinear. So techniques may be needed to solve the nonlinearity problem accurately before we can design an effective DVFS controller.

In the rest of this section, we describe modeling and design of an online DVFS scheme for local queues and domains. That is, the DVFS controller considers each queue locally and separately, assuming the interactions between queues are weak and can be ignored (the so-called decentralized DVFS scheme). We leave the issue of centralized DVFS to Section 2.5.

2.3.2 Overview of modeling and design of DVFS controller

This subsection gives an overview of the modeling and design of our online DVFS controller without going through control theoretic and mathematical details (those details will

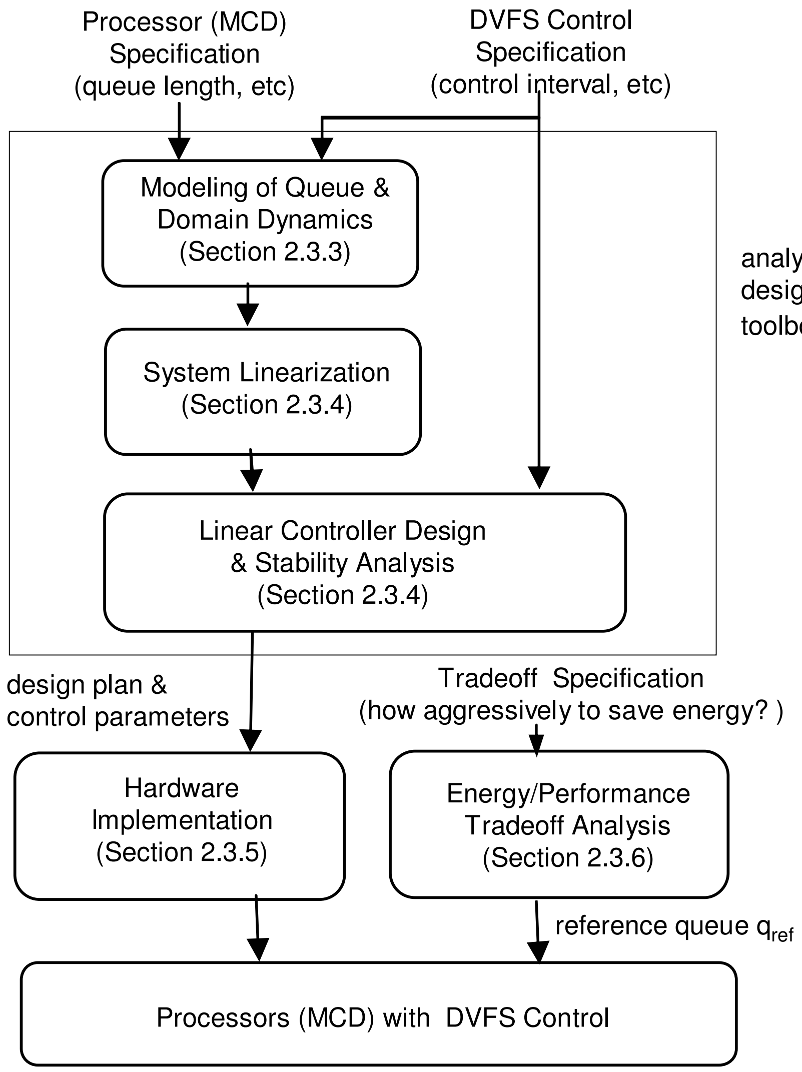
Processor (MCD) Specification (queue length, etc)
DVFS Control Specification (control interval, etc)
Modeling of Queue & Domain Dynamics (Section 2.3.3)
System Linearization (Section 2.3.4)
Linear Controller Design & Stability Analysis (Section 2.3.4)
analysis/ design toolbox
design plan & control parameters
Tradeoff Specification (how aggressively to save energy?)
Hardware Implementation (Section 2.3.5)
Energy/Performance Tradeoff Analysis (Section 2.3.6)
reference queue q_{ref}
Processors (MCD) with DVFS Control

Figure 2.4: Design flow for our DVFS controller in MCD processors. Section numbers inside the blocks indicate the location where the details of each design step can be found.

be given in subsections 2.3.3 - 2.3.6).

We use a local queue model as shown in Figure 2.3. The circle in that figure represents the clock domain being considered (such as a floating point functional unit). The domain has a frequency f and an execution capability or service rate μ, which is a function of f. The performance demand is denoted as λ. The queue has a finite size. (Note this model uses an input queue to a domain. For the case of an output queue, duals of the arguments in this section will hold.)

Figure 2.4 shows the design flow of our DVFS controller, with the block diagrams corresponding to the major steps for our methodology. The inputs to the modeling and design blocks are the processor (MCD) specification and the DVFS control specification. These include queue length, clock domains and their frequency ranges, control interval (i.e. the time interval for one possible change of frequency/voltage), sampling period, queue measures used as feedback signals, and other control performance requirements such as maximum percent overshoot allowed.

The first major step is the modeling of the involved queue and clock domain dynamic. (The details are described in subsection 2.3.3.) Based on the processor and control specifications, we derive a mathematical model for the controlled system, which is expressed as a set of difference equations. These equations describes the dynamic relationship among the feedback signal (q), the demand (λ), and the frequency (f).

The next two major steps are the system linearization and the linear controller design (both are described in subsection 2.3.4). Since the controlled queue/domain system is inherently nonlinear, we first linearize the system using an accurate linearization technique called *nonlinear transformation* [4]. As shown in Figure 2.5, this technique essentially adds a nonlinear transformer to the feedback path to compensate for the nonlinearity in the original controlled system. We then design a variation of the Proportional-Integral-Derivative (PID) controller for this linearized system, which is commonly used in industry [41, 62]. Intuitively, the PID-based controller adjusts the execution rate to adapt to the workload change. This adjustment is in proportion to the value of the workload change, the rate of the workload change, and takes into account the prior history of workload changes. Figure 2.5 shows the control block diagram, where q is the measured feedback signal; q_{ref} is the target or nominal queue operating point; e is the error signal and the input to the PID-based controller; μ is the obtained control signal (or actuate signal) which is proportional to the value of the error, the integral of previous errors, and the rate of the error change; f is the frequency and is obtained from μ through a nonlinear transformation; demand λ is the disturbance input which the controller is trying to adjust the execution rate to match.

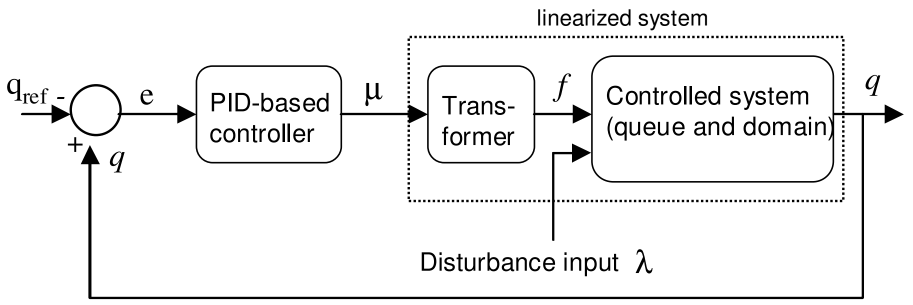

Figure 2.5: Control block diagram for a Proportion-Integral-Derivative (PID) controller for DVFS .

The control parameters (or control gains) for the PID-based controller are decided by the stability and control performance analysis. Note these control gains are the main output of the analysis/design toolbox. The next major step is to take the design with the control gains, and implement it in hardware (we show a possible hardware implementation in subsection 2.3.5). This finishes the design cycle of a DVFS controller in MCD processors.

A key design setting for the above DVFS controller is the reference queue occupancy q_{ref}. As we will show in subsection 2.3.6, the value of q_{ref} specifies the actual tradeoff between performance degradation and energy saving. We can increase q_{ref} to make DVFS control more aggressive in saving energy, or decrease q_{ref} to value performance more. Note that a design could make the value of q_{ref} adjustable by the OS or application software (e.g. through a special mode set instruction). This mechanism provides an opportunity for hardware/software cooperation in the sense that the hardware is responsible for implementing the fine details of speed adaptation while the OS/application software will make the overall policy decision (through q_{ref}) on how aggressively to save energy or preserve performance.

In the rest of this section, we will describe in detail the design steps in Figure 2.4. We will start with the modeling and linear controller design, which involves some control theoretic and derivation details. (People who are already familiar with these derivations or are not interested in these details may wish to skip subsections 2.3.3 and 2.3.4.) In practice, these control theoretic techniques/details can be incorporated into an analysis/design toolbox. A designer is able to use the toolbox to get the control parameters, and then use them directly in the DVFS policy controller design.

2.3.3 Analytic modeling of queue and clock domain dynamics

As mentioned earlier, we use a local queue model as shown in Figure 2.3. For a clock domain, the frequency and corresponding voltage cannot change instantaneously, and there is a minimum time requirement for one possible change of frequency. So we have a control interval such that the frequency is fixed inside an interval. Using T as the length of a control interval, the k_{th} control interval is just the time period $[kT, (k+1)T]$. Also we denote N as the total number of sampling periods in a control interval, and assume each sampling period has a length of Δt, so $T = N\Delta t$.

We first want to model the performance demand and service rate in a control interval. The demand $\lambda_{(t)}$ and service rate $\mu_{(t)}$ inside each control interval are modeled as an independent and stationary random process along the time axis [30] (i.e., they have identical distributions for all t inside an interval). We denote the expected values and variances (or noise levels) of $\lambda_{(t)}$, $\mu_{(t)}$ as $\bar{\lambda}$, $V_{(\lambda)}$, $\bar{\mu}$, and $V_{(\mu)}$ respectively. (From now on, for a variable x, we will use $\bar{x}$ and $V_{(x)}$ to represent the expected value and variance of x.) Also, we use a subscript k as in $\bar{\lambda}_k$, $V_{(\lambda_k)}$ to indicate these values are for the k_{th} control interval.

Consider $q(t)$ as the queue occupancy at time t. Then the basic queue equation is expressed as follows (essentially a simplified version of the Lindley equation [44]) :

$$q_{(t+\Delta t)} - q_{(t)} = (\lambda_{(t)} - \mu_{(t)})\Delta t \tag{2.1}$$

Intuitively, this means that queue occupancy change in a unit time is equal to the number of arrived elements minus the number of departed elements in that unit time.

Next we want to use the above basic queue equation to model the queue-domain dynamics across different control intervals. First, we need to define the feedback signal and other necessary dynamic state variables. For a control interval, there are a number of ways to measure the queue utilization and use it as a feedback signal to the controller. In this paper, we use the average queue occupancy over all sampling points in the previous interval as the feedback signal for the current interval (as was used by the heuristic-based DVFS approach in [58]). We denote this feedback signal for the k_{th} control interval as q'_k. Also, the queue occupancy at the starting point of the k_{th} interval is q_k, so we have $q_k = q(kT)$. If we express these two dynamic state variables q'_k and q_k in terms of values from the previous interval, we have

$$\begin{aligned} q'_k &= \tfrac{1}{N}\sum_{i=1}^{N} q_{((k-1)T+i\Delta t)} \\ q_k &= q_{((k-1)T+N\Delta t)} \end{aligned} \tag{2.2}$$

Next we recursively expand equations (2.2) using the the basic queue dynamics in (2.1) and take the expected value of both sides of the expanded equation. Since both λ and μ are stationary in a control interval, $\bar{\lambda}_k$ and $\bar{\mu}_k$ are independent of time t for any k. So we have

$$\begin{aligned} \bar{q'}_k &= \bar{q}_{k-1} + \tfrac{1}{N}\sum_{i=0}^{N-1}\sum_{j=0}^{i}(\bar{\lambda}_{k-1} - \bar{\mu}_{k-1})\Delta t \\ &= \bar{q}_{k-1} + \tfrac{T+\Delta t}{2}(\bar{\lambda}_{k-1} - \bar{\mu}_{k-1}) \\ \bar{q}_k &= \bar{q}_{k-1} + (\bar{\lambda}_{k-1} - \bar{\mu}_{k-1})T \end{aligned} \tag{2.3}$$

Intuitively, this means $\bar{q'}_k$, the average queue occupancy over a control interval, is equal to the sum of the queue occupancy at the beginning of the interval ($\bar{q}_{k-1}$) and the average queue changes due to the differences between the demand rate $\bar{\lambda}_{k-1}$ and the service rate $\bar{\mu}_{k-1}$. A similar interpretation exists for $\bar{q}_k$, the queue occupancy at the beginning of next interval, in the above equation.

The above equation describes the dynamics in the considered queue-domain system across different control intervals. The RHS of equation (2.3) is expressed in terms of the expected value of the service rate $\bar{\mu}_k$, while the real control signal is f_k. So we need to model their relationship. We use the following model

$$\bar{\mu}_k = \frac{1}{\bar{t}_1 + \frac{C_2}{f_k}} \tag{2.4}$$

The above equation comes from the fact that, in most clock domains, execution time can be separated into two parts – one that is independent of frequency and one that is dependent. For example, in a load/store domain, the time spent for accessing asynchronous memory due to a cache miss is independent of frequency, while the time for querying and accessing cache is dependent on frequency. Accordingly, in the above model, $\bar{t}_1$ is the average amount of unit time (or task step) per instruction that is independent of frequency, and $\bar{C}_2$ is the average number of frequency-dependent cycles per instruction. Parameters $\bar{t}_1$ and $\bar{C}_2$ can be estimated dynamically using techniques similar to those in [19, 68].

Putting (2.3) and (2.4) together, and dropping Δt in (2.3) as $T \gg \Delta t$, we have an analytic model for the considered queue-domain system as

$$\begin{aligned} \bar{q}'_k &= \bar{q}_{k-1} + \tfrac{T}{2}(\bar{\lambda}_{k-1} - \tfrac{1}{\bar{t}_1 + \frac{\bar{C}_2}{f_{k-1}}}) \\ \bar{q}_k &= \bar{q}_{k-1} + (\bar{\lambda}_{k-1} - \tfrac{1}{\bar{t}_1 + \frac{\bar{C}_2}{f_{k-1}}})T \end{aligned} \tag{2.5}$$

In the above modeling of queue dynamics, we assume the queue at the nominal operating point is partially full. The queue-domain relations when the queue is completely empty or full will be different from that in (2.1), which must be modeled using some discrete binary functions like those we will use in Section 2.5.

In addition, in the above model, we have only considered the expected values of λ and μ, and have not considered the variance or noise level of the input $V_{(\lambda)}$, $V_{(\mu)}$. In particular, we want to check how the input noise propagates in the feedback signal $\bar{q}'$, since the noise in the feedback signal may affect the efficiency of a controller. We compute the variance $V_{q'_k}$ in one interval, assuming the queue occupancy is known at the beginning of the interval. By expanding the first sub-equation in (2.2), taking the variance of both sides of the expanded equation, and following the variance calculation rules [30], we have

$$\begin{aligned} V_{(q'_k)} &= \tfrac{T+\Delta t}{2T}\left(V_{(\lambda_{k-1})} + V_{(\mu_{k-1})}\right) \\ &\approx \tfrac{1}{2}\left(V_{(\lambda_{k-1})} + V_{(\mu_{k-1})}\right) \end{aligned} \tag{2.6}$$

From (2.6), we see the variance or noise in the feedback signal q' is not amplified and stays

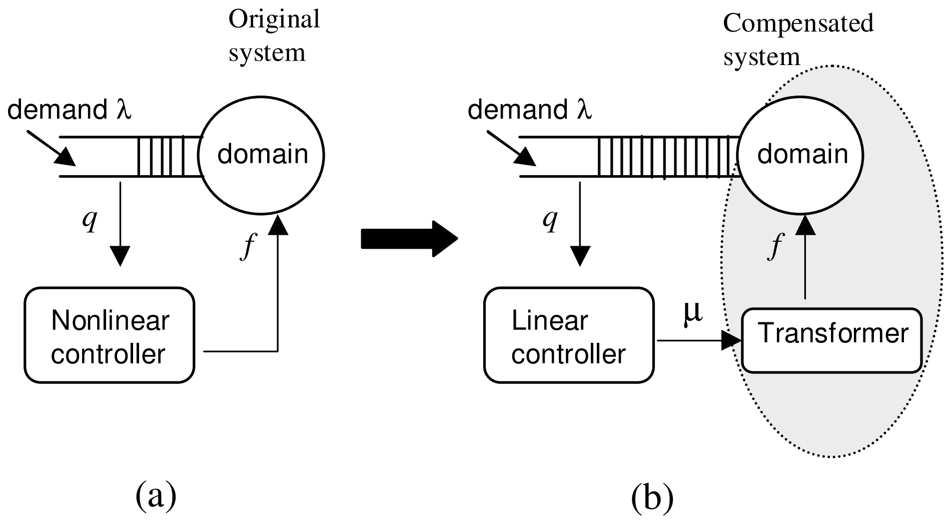

Figure 2.6: Linearization of the original dynamic system through a nonlinear transformation on the feedback path

at roughly the same level as the input noise. This is beneficial to the controller because smaller noise in the feedback signal tends to have less negative impact on the efficiency of the controller.

2.3.4 DVFS controller design

A straightforward design approach for our goals would be to design a controller for f_k, as shown in Figure 2.6a. However, as indicated in equation (2.5), this control system is nonlinear, and it is generally hard to design an effective controller for a nonlinear system, as there are very limited control techniques and tools for a general nonlinear system [41]. Fortunately, the nonlinearity inside this system can be separated, so we can have a nonlinear transformation on the feedback path to compensate for the nonlinearity in the original system dynamics; in this way, the compensated system becomes a linear one. This accurate linearization technique is the so-called *feedback linearization* or *nonlinear transformation* [4], as shown in Figure 2.6b. Specifically, we take μ_k as the control signal for the compensated linear system. The actual or internal control signal f_k is obtained through a transformation on the feedback path.

After the linearization, we can design the controller using a rich set of linear control techniques [41]. A popular choice would be a variation on the Proportion-Integral-Derivative (PID) controller. In this work, we use a PI controller because it is relatively simple to design and robust in terms of steady-state control performance. The controlled system with the PI controller can be described by the following state equations.[1]

$$\begin{aligned}
\bar{q'}_k &= \bar{q}_{k-1} + \tfrac{T}{2}(\bar{\lambda}_{k-1} - \bar{\mu}_{k-1}) \\
\bar{q}_k &= \bar{q}_{k-1} + (\bar{\lambda}_{k-1} - \bar{\mu}_{k-1})T \\
e_k &= \bar{q'}_k - q_{ref} \\
\bar{\mu}_k &= \bar{\mu}_{k-1} + K_I e_k + K_P(e_k - e_{k-1}) \\
f_k &= \tfrac{\bar{C}_2\bar{\mu}_k}{1-\bar{t}_1\bar{\mu}_k}
\end{aligned} \tag{2.7}$$

where the first two sub-equations are simply the analytic queue-domain model in (2.5); q_{ref} is the reference queue occupancy, i.e. the target or nominal operating queue point (more discussion of this in Section 2.3.6) ; e_k is the error signal; μ_k is the new service rate coming from the PI controller, with K_I and K_P the control parameters (or the so-called control gains); f_k is the new clock frequency obtained by solving (2.4) for f_k.

We then can proceed with stability and transient performance analysis of (2.7) and choose appropriate control gains K_I and K_P [41], as described next.

In equation (2.7), we define μ_k as the output signal, and λ_k as the disturbance input signal. Also we define new constants $K'_I = K_I T$, $K'_P = K_P T$. We can get the transfer function through the input-output difference equation and the z-transformation [41]. From the transfer function, we can get the corresponding characteristic equation as

$$2z^3 + (K'_I + K'_P - 4)z^2 + (2 + K'_P)z - K'_P \;=\; 0 \tag{2.8}$$

From stability theory [41], the system in (2.7) is stable if and only if the roots of (2.8) are all inside the unit cycle of the z-plane. Assuming the roots of (2.8) are z_1, z_2, and z_3, we

[1]Note these equations describe the dynamic relationship between the state and control variables. For the control relationship between these variables, please refer to the control block diagram in Figure 2.5 in Section 2.3.2.

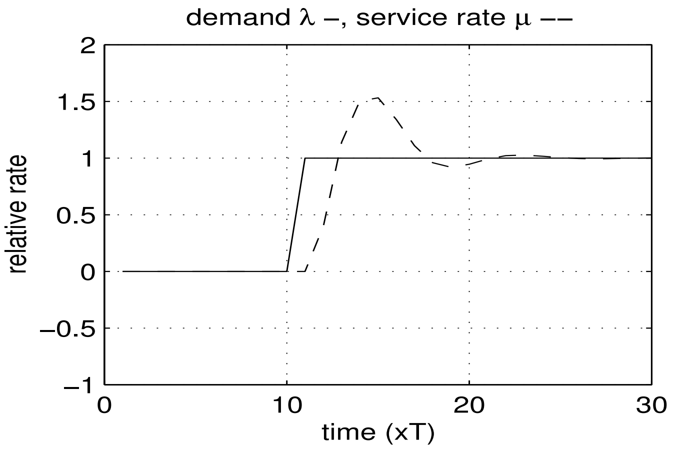

Figure 2.7: Step response for the system in (2.7) with $K_I' = 0.2$, $K_P' = 0.6$.

have the following relationships between the roots and the setting K_I', K_P', using a standard coefficient matching technique.

$$\begin{aligned} K_I' &= 2(z_1z_2 + z_2z_3 + z_3z_1) - 2 \\ K_P' &= 2z_1z_2z_3 \\ z_3 &= \tfrac{3-z_1z_2-z_1-z_2}{z_1z_2+z_2+z_3+1} \end{aligned} \tag{2.9}$$

With equation (2.9), we can use the poles-placement technique [41] to choose the appropriate K_I',K_P' setting for a given stability margin and transient performance requirement. For example, if z_1 and z_2 are chosen with a very small magnitude (e.g. $z_1 = z_2 = 0.25$, as smaller magnitudes give faster settling times), z_3 will be placed outside the unit cycle in the z-plane (e.g. $z_3 = 1.6$) and the system will be unstable. So to make the system stable, we need to choose z_1 and z_2 with a relatively large magnitude and sacrifice the settling time a little bit. We also may want it relatively underdamped in order to have a fast rising time, assuming we can tolerate the overshoot caused by the underdamping. Considering all of the above, one good placement is to have $z_{1,2} = 0.5 \pm 0.5i$, $z_3 = 0.6$, and the corresponding $K_I' = 0.2$, $K_P' = 0.6$. Figure 2.7 shows the step response of the system in (2.7) for this setting.

In the above controller design, we assume frequency f_k can vary without bounds. In practice, f_k has an upper bound and a lower bound. Also, there is a maximum possible

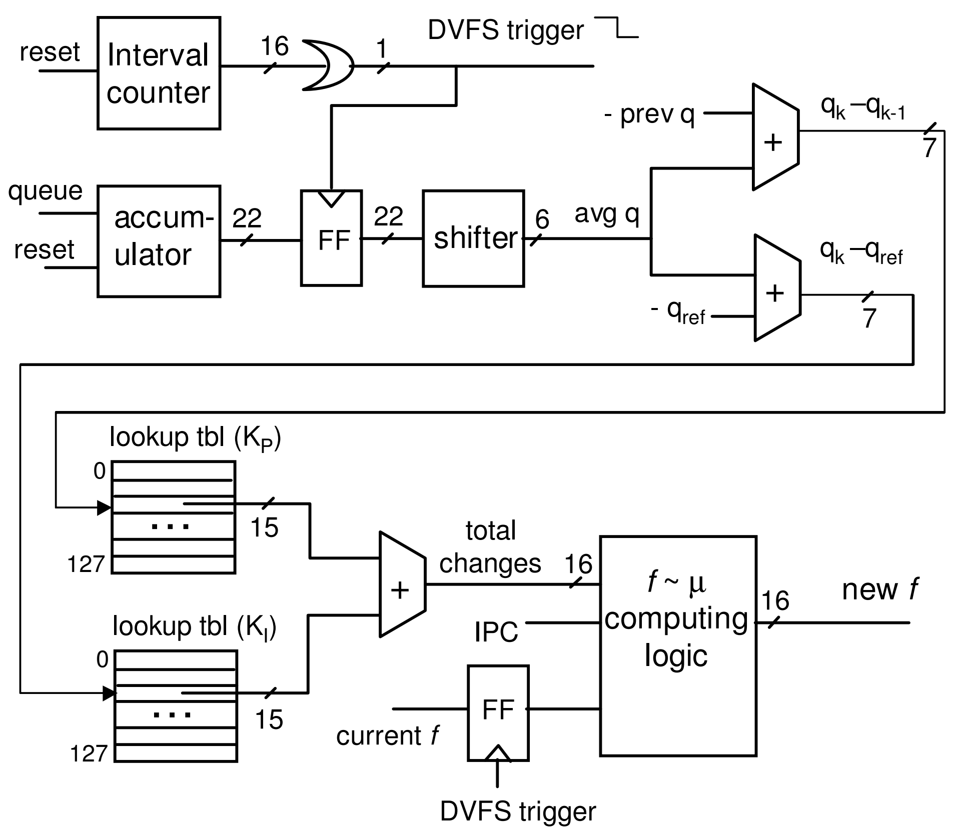

Figure 2.8: Block diagram for a possible hardware implementation of the DVFS controller in a clock domain.

frequency change in one control interval for a given interval length and clock changing rate. So, in practice, the DVFS controller would need to check with these bounding values. In addition, we assume an XScale-style frequency control which allows continuous frequency changes. For processors which allow only stepwise frequency changes, the DVFS controller would need to choose a discrete step which is closest to the requested continuous value.

2.3.5 Hardware implementation of the DVFS controller

The most important hardware required to implement the proposed DVFS scheme are the queues. As these queues already exist as synchronization queues between clock domains, the online DVFS controller for an MCD processor uses the existing hardware in a "two for one" style.

The rest of the required hardware includes two counters, which are similar to the hardware counters in [58]. One is used to frame the control interval (a time counter or instruction counter). Since the length of a control interval is typically a few thousand cycles or instructions, a single 16-bit counter should be sufficient. The other one is a queue counter to get the cumulative queue occupancy. Given that a queue size is typically around 20 ($\ll 2^6$) and we have to total up such occupancies over $\leq 2^{16}$ cycles, a 22-bit queue counter should be sufficient. One possible implementation for computing the average queue occupancy is to choose the interval length as power of two and use a shifter as shown in Figure 2.8. The most complicated part is the logic required to compute the control signal (f). Since it is done only once for each control interval and is not time critical, this computation can be finished in multiple cycles. Specifically, this logic first needs to compute μ_k as

$$\mu_{k-1} + K_I(\bar{q'}_k - q_{ref}) + K_P(\bar{q'}_k - \bar{q'}_{k-1})$$

One possible implementation for this task is shown in Figure 2.8, which uses two adders to compute the queue differences, and two pre-computed lookup tables to get the actual control gains or changes (one for K_I and one for K_P). The actual size of the lookup tables depends on the queue size, the resolution of the queue values, and the control gains. For a queue size of 32, a resolution of 0.25 for queue values, and a 7-bit K_P or K_I resolution, each lookup table requires 128 entries and 15 bits per entry as illustrated in Figure 2.8.

This logic also needs to compute values of $\mu(f)$ with $\mu = 1/(\bar{t}_1 + \bar{C}_2/f)$. The $\bar{t}_1$ and $\bar{C}_2$ can be estimated online by some performance counters [68]. The rest can be computed by a multiplier or again by using some pre-computed lookup tables. In order to reduce the hardware requirement for this part, we approximate the $\mu(f)$ function with some piece-wise linear functions. (An illustration is given in Figure 2.9 with $\bar{t}_1 = 0.35$, $\bar{C}_2 = 0.1$.) That is, we divide the frequency range into many small segments, each of which is centered at an operating point. Then for f inside a particular segment, μ can be computed using a linear function $\mu = IPC \cdot f$, where IPC is the effective IPC at the center operating point. For the

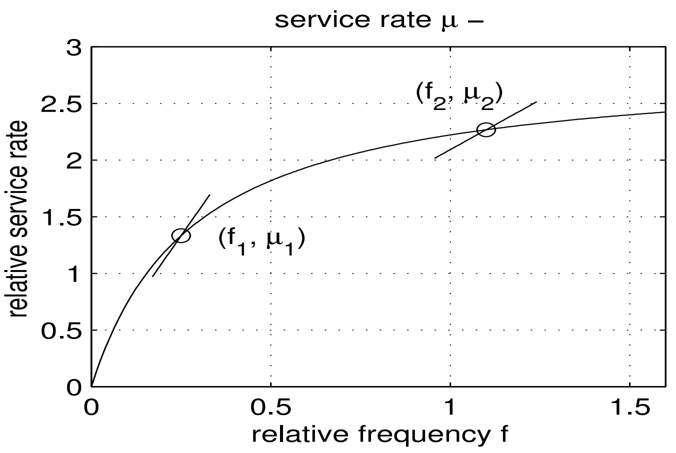

Figure 2.9: Piece-wise linear functions to approximate the real $\mu(f)$ function around two operating points.

example in Figure 2.9, we have $\mu = IPC_1 f$ for f inside a small segment around operating point (f_1, μ_1), and $\mu = IPC_2 f$ for the small segment around (f_2, μ_2). If the size of the frequency segment is small, this will give a quite good approximation. (In the next section we will see that because the frequency change rate is relatively slow in an MCD processor, the maximum possible frequency change in an interval is relatively small; thus the error from this piecewise approximation is relatively small.) With this approximation, the hardware required to compute $\mu \sim f$ will be reduced. We will need to estimate $IPC_{effective}$ online using a regular IPC counter, then compute the new frequency f using the current f, total control gains (changes), and IPC as shown in Figure 2.8. The computing logic for this part can be implemented using a pre-computed lookup table as discussed above, or using a 16-32 bit multiplier depending on the frequency resolution. As mentioned earlier, this computation is not time critical, so we can use some serial multiplication techniques to further reduce the hardware requirement. Finally, the new frequency signal f will be used by the frequency and voltage control mechanism in a clock domain.

Overall, only a modest amount of hardware support is needed to implement the online DVFS controller.

2.3.6 Specifying energy and performance tradeoffs with $\mathbf{q_{ref}}$

In the DVFS controller design in previous subsections, the reference queue q_{ref} specifies the nominal operating queue point. In principle, q_{ref} can be any value which is neither full nor empty (i.e $0 < q_{ref} < 1$, if expressed relative to the queue size). In this subsection, we show the position of q_{ref} specifies the actual tradeoff between performance degradation and energy efficiency.

We continue to consider the single queue model in Figure 2.3. Notice that, for this queue system, the steady state throughput (i.e. the steady state performance) degrades if and only if the queue is full. That is because, at that point, the performance demand from the upstream domain cannot be satisfied and the arriving process is forced to stall. Similarly, energy is wasted if and only if the queue is empty, because the domain is running idle and not doing any useful work. Note, control errors and input noise/variation are the two major causes for the queue to become full or empty.

In addition, we notice that the distance from the nominal operating point (q_{ref}) to the full-end queue point, $[q_{ref}, 1]$, reflects the relative margin for the queue to tolerate the control errors and input variation before the queue becomes full and loses performance. Similarly, the distance from q_{ref} to the empty-end queue point, $[0, q_{ref}]$, reflects the margin to tolerate errors before the queue becomes empty and wastes energy.

Therefore, if q_{ref} is increasing, then the distance $[q_{ref}, 1]$ is decreasing and the system is more likely to suffer performance degradation. On the other hand, for an increasing q_{ref}, the distance $[0, q_{ref}]$ is also increasing and the system is less likely to waste energy. Qualitatively, the bigger the q_{ref}, the more performance degradation and the more energy savings in general. We can choose how much energy performance tradeoff we want by choosing an appropriate q_{ref} in our online DVFS controller.

As mentioned in Section 2.3.2, in our design, we can also leave the policy parameters like q_{ref} adjustable by the OS or application software. So the OS/application can direct the overall power management with a simple lever (making the overall decision on how aggressively to save energy or preserve performance) while leaving the actual implementation details of speed adaptation in hardware. This mechanism shows an example of hardware/software cooperation in DVFS control.

In the rest of this subsection, we will further give a quantitative theoretic estimation of the performance degradation and energy savings as a function of q_{ref}. (Later, in Section 2.4, we will show the actual experimental results.) Denote the control error as ε_c, the input variation as ε_v, and the queue length as L. In concept, we have the following

$$\begin{aligned} \text{Performance} &= \text{PerfectP} - \text{Degradation} \\ &= \text{PerfectP} - D(q_{ref}, L, \varepsilon_c, \varepsilon_v) \\ \text{Energy Savings} &= E_{\text{slack}} - \text{Wasted} + E_{\text{non-slack}} \\ &= E_{\text{slack}} - W(q_{ref}, L, \varepsilon_c, \varepsilon_v) + E_{\text{non-slack}} \end{aligned} \tag{2.10}$$

where PerfectP is the perfect performance of Figure 2.2, D is the performance degradation, E_{slack} and $E_{\text{non-slack}}$ are the *slack energy savings* and the *non-slack energy savings* defined in Section 2.3.1.

From (2.10), we see the value of Energy Savings is mostly decided by the E_{slack} in the original program and the value of *non-slack energy savings*. The energy wasted W is affected directly by q_{ref} as we mentioned. So in general, as q_{ref} increases, W decreases, (also *non-slack energy savings* increases), and the overall Energy Savings increases.

The relative performance, on the other hand, is more closely related to q_{ref} as shown next. To get a theoretical estimation of the relative performance, we assume the program variation pattern is more or less exponentially distributed, so the demand and server process can be approximated by a Markov model [8]. The queue system considered in this section can be modeled as an M/M/1/L queue [8]. Also, assuming the controller error ε_c is 0, we have $\mu_k = \lambda_k$ and the server utilization $\rho = 1$ [8]. For a given q_{ref} and queue size L, the probability that the queue becomes full is

$$P_{loss} = \frac{1}{(1 - q_{ref})L + 1} \tag{2.11}$$

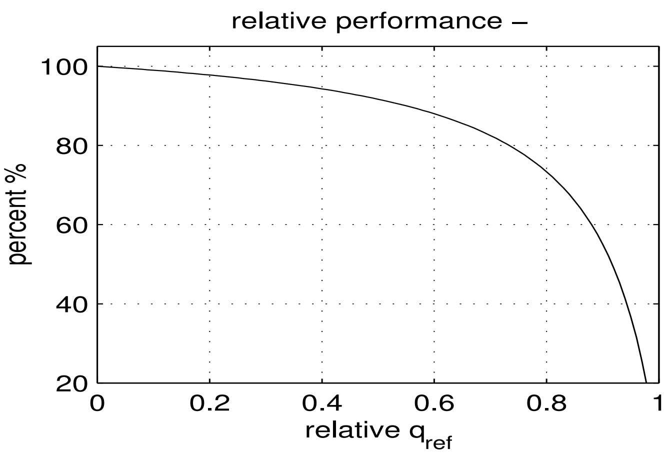

Figure 2.10: An analytic estimation of the relative performance of the single queue-domain system as a function of the reference queue q_{ref} using a M/M/1/L queue model.

Using equations (2.10) and (2.11) , we have the performance as

$$\begin{aligned} \text{Performance} &= \text{PerfectP} - P_{loss} \cdot \text{PerfectP} \\ &= \left(1 - \frac{1}{(1-q_{ref})L+1}\right) \text{PerfectP} \end{aligned} \tag{2.12}$$

If we use the performance at $q_{ref} = 0$ as the base performance, then we can compute the relative performance as a function of q_{ref} from (2.12) as

$$\text{Relative Performance} = \frac{(1-q_{ref})(L+1)}{(1-q_{ref})L+1} \tag{2.13}$$

Figure 2.10 shows an example curve from equation (2.13) with $L = 10$. It is observed that the relative performance degrades as q_{ref} increases, and the slope increases dramatically.[2]

The intention of the above discussion is to get some analytical insight on how the relative performance degrades with q_{ref}. In Section 2.4, we will give the actual experimental results on performance and energy saving as a function of q_{ref}.

[2]In reality, the applications' variation patterns are typically not strict-exponentially distributed. So the actual performance-q_{ref} curve might be slightly different from that in Figure 2.10. Further, different applications may have different execution variation patterns, which may lead to different performance-q_{ref} curves.

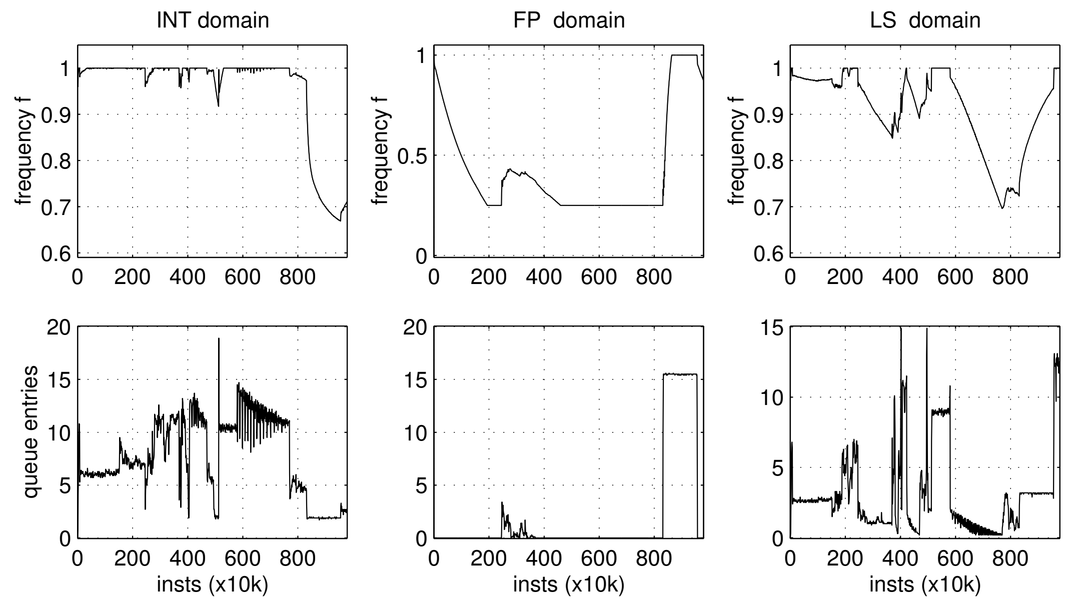

Figure 2.11: Frequency trace from DVFS for the benchmark EPIC-Decode in the top row, and the corresponding queue trace in the bottom row.

2.4 Experimental Results

In this section, we present experimental results to illustrate and evaluate the effectiveness of the proposed online DVFS scheme.

2.4.1 Simulation methodology and setup

Our simulation environment is based on the SimpleScalar toolset [13] with the Wattch [10] power estimation extension and the MCD processor extension [59]. The MCD extension by Semeraro *et al.* in [59] has 4 clock domains as shown in Figure 2.1. It also includes a cycle-by-cycle computation of the synchronization overhead due to independent clock frequency, phase, and clock jitter. An XScale-like dynamic voltage and frequency changing mechanism has been implemented, which allows any frequency to be used within the allowable range, as described in Section 2.2.

We have made two major modifications to the simulation core, in order to have a more accurate energy and performance estimation. First, the load-store queue in the MCD simulator has been split into a separate load-store issue queue and a load-store retirement buffer.

Table 2.1: Summary of All Simulation Parameters

Simulation Parameters	**Value**
Reference queue point	6 INT, 5 FP, 3 LS
Domain frequency range	250MHz – 1.0GHz
Domain voltage range	0.65V - 1.20V
Frequency/voltage change speed	73.3 ns/MHz, 171 ns/2.86mV
Control interval length	10000 instructions
Domain clock jitter	$\pm$110ps, normally distributed
Inter-domain synchro window	300ps
Branch predictor:	
2-level	L1 1024, hist 10, L2 1024
Bimodal	size 1024, BTB 4096 sets, 2-way
Combined	size 4096
Decode/Issue/Retire width	4/6/11
L1 data cache	64KB, 2-way
L1 instr cache	64KB, 2-way
L2 unified cache	1MB, direct mapped
Cache access time	2 cycle L1, 12 cycles L2
Memory access latency	80 first chunk, 2 inters
Integer ALUs	4 + 1 mult/div unit
Floating-point ALUs	2 + 1 mult/div/sqrt unit
Issue queue size	20 INT, 16 FP, 16 LS
Reorder buffer size	80
LS retire buffer size	64
Physical Register file size	72 INT, 72 FP

Second, the energy computation in the latest MCD simulator [48] uses a formula inherited from Wattch, which computes the energy as the sum of power on a cycle-by-cycle basis. While this formula works fine for a processor with a fixed frequency, it may give overly-optimistic energy results for a processor with dynamically varying frequency. So we modified the energy computation formula to account for the varying frequency in an MCD processor with DVFS.

We implemented the online DVFS controller for local queues and domains, following the design in Section 2.3. Also, as in [48, 58], we made the front end domain run at a fixed maximum speed, and allowed the INT, FP, and LS domains to be controlled by the DVFS controller. We assume a performance degradation target of about 4%, which is roughly the same as that in [58]. Using the curve in Figure 2.10 as a general guide, we chose the reference queue point q_{ref} as roughly $\frac{1}{3}$ of the total size for INT and FP domains. (That is, q_{ref} is 6 for the INT domain, 5 for FP.) Since the LS domain is relatively more critical to the overall performance as shown in [50], we chose its q_{ref} to be 3 which is roughly $\frac{1}{5}$ of the total size. For the μ and f relations in (2.4), we used the piece-wise linear approximation $\mu = IPC_{effective}f$ as discussed in Section 2.3.5. Also, we assume clock gating will be applied whenever the unit is not used. All other architecture parameters are chosen to have the same values as those in [48, 58]. A summary of all simulation parameters is in Table 2.1.

We want to evaluate our online DVFS scheme with a broad set of applications. To show variety, we will present results for 6 MediaBench applications, 8 SPECint applications, and 4 SPECfp applications as shown in Figure 2.12. We chose roughly the same subset of SPECint and SPECfp as those used in [48, 58]. We believe these programs form a representative set as they display a range of program behavior. For MediaBench benchmarks, we use the official data input set in the MediaBench web site and the whole program as the simulation window; for SPEC2000 benchmarks, we use the reference input set and choose the simulation window using the published Early SimPoint numbers [56].

As an illustrative example, in Figure 2.11 we show the frequency setting from the online DVFS controller for the benchmark *EPIC-Decode*. The corresponding average queue occupancy trace is also shown there. Clearly, there is a strong correlation between the queue traces and the obtained DVFS frequency settings. This correlation is most obvious for the FP domain where the FP issue queue is empty except for two distinct phases. At the beginning, the DVFS controller detected the queue emptiness and gradually reduced the FP frequency to $f_{min} = 0.25GHz$. Then there was a modest frequency recovery in the first phase. During the second phase, the DVFS controller detected a dramatic increase of queue entries (i.e. increasing demand to the FP clock domain) and quickly adjusted the clock frequency to $f_{max} = 1.0GHz$. The correlations for the INT and LS domains are

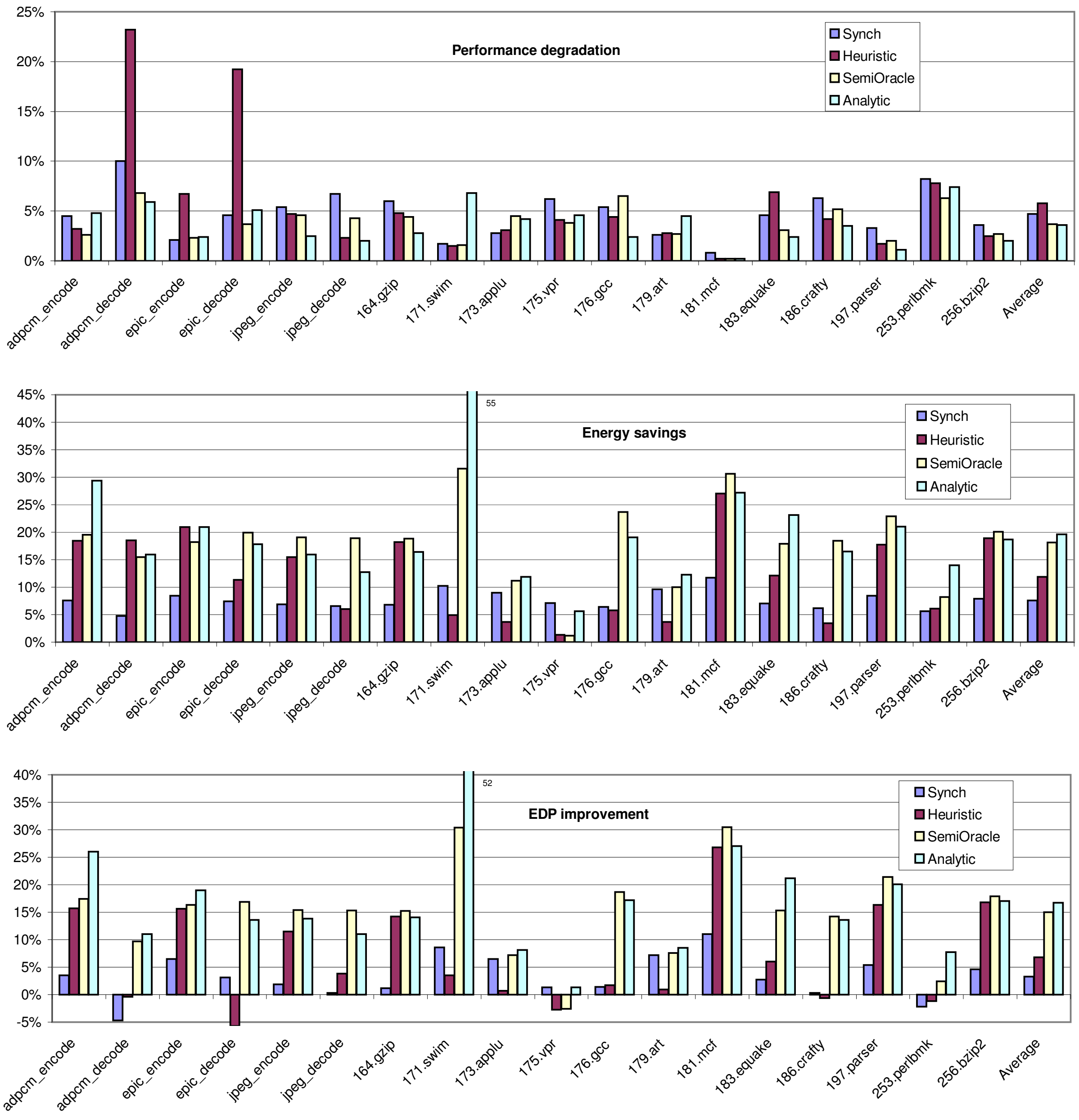

Figure 2.12: Performance degradation, Energy savings, and Energy-delay product improvement for each benchmark; different schemes are *Synchronous* voltage scaling, *Heuristic*-based online DVFS, *SemiOracle* DVFS, and *Analytic* online DVFS.

similar, but are less obvious as the queue traces become relatively more complicated.

Next, we will look at the energy/performance efficiency of the proposed online DVFS controller, and compare results with those from some best-known prior work.

2.4.2 Energy and performance results for different approaches

To compare our results to other prior DVFS approaches, we hold performance roughly the same for all approaches and look at metrics such as energy savings, energy-delay product improvement, and power/performance ratio. We define the power/performance ratio as percentage power saved per percentage performance degradation – that is, what percentage power is saved for one percent performance degradation. Note this definition is the same as that in [58].

We will present results from our analytic online DVFS scheme (denoted as *Analytic*). We compare them to those from the heuristic-based online DVFS in [58] (denoted as *Heuristic*), and those from the semi-oracle-based DVFS in [59] (denoted as *SemiOracle*). The *SemiOracle* assumes the DVFS, by oracle, has full knowledge of existing slack in a program, and uses a *Shaker* algorithm to decide DVFS settings. So its results are not realistic, but serve as a comparison. (However, as stated in [59], the *SemiOracle* result is not the upper bound for all possible DVFS results.) The parameters we use for *Heuristic* and *SemiOracle* are taken from [58] and [59], with a 5% performance degradation target for *Heuristic* and 1% for *SemiOracle* because these target values will lead to an actual performance degradation similar to *Analytic*. We also want to compare our results to those from the conventional (fully) synchronous voltage scaling (denoted as *Synchro*) which scales the frequency/voltage for the whole processor. It is set to get roughly the same performance degradation as other approaches.[3]

The performance degradation, energy savings, and energy-delay product for each benchmark are shown in Figure 2.12. The results are relative to the conventional (fully) synchronous processor without voltage scaling. So all results for MCD include about 1.5% percent inherited performance and energy overhead from the baseline MCD processor, as discussed in [58]. The average results over all 18 benchmarks are summarized in Table 2.2.

[3]We were not able to implement synchronous dynamic voltage scaling, so the *Synchro* results are for static scaling which may under-represent the benefit of synchronous voltage scaling.

Table 2.2: Average results for different schemes, relative to a conventional (fully) synchronous processor.

Schemes	Performance degradation	Energy savings	Energy-Delay product improvement	Power/ Performance ratio
Analytic	3.6%	19.6%	16.7%	6.2
SemiOracle	3.7%	18.1%	15.0%	5.6
Heuristic	5.8%	11.9%	6.8%	3.0
Synchro	4.7%	7.6%	3.3%	2.5

From Table 2.2 and Figure 2.12, there are several interesting observations. First, the overall results from *Analytic* DVFS are very promising. We achieve a power/performance ratio of 6.2 on average relative to a fully synchronous processor. Second, compared to results from *Heuristic* DVFS, *Analytic* DVFS achieves far better results in terms of Energy-Delay Product (EDP) improvement and power/performance ratio. For example, the average EDP improvement by *Analytic* is 146% higher than that of *Heuristic* (16.7% vs 6.8%, with numbers relative to a synchronous processor). *Analytic* DVFS also produces an average result better than those of *SemiOracle*. For example, the average EDP improvement by *Analytic* is about 11% higher than that by *SemiOracle* (16.7% vs 15.0%)[4] These results show the effectiveness of the proposed analytic online DVFS scheme due to the automatic regulation ability of a DVFS controller.

Lastly, all MCD DVFS results (both analytic and heuristic-based) are much better than those by the synchronous voltage scaling, which shows the energy savings potential of a MCD processor due to the extra flexibility in DVFS control. Note that the *Synchro* numbers from our experiments are lower than what is usually expected from the fact that performance scales linearly with f while energy scales quadratically with v. The main reason is

[4]The *SemiOracle* results from our experiments are close to the published results (relative to a synchronous processor) in [58, 59] . We noticed, however, that the *Heuristic* results are not close to the results reported in [58]. We carefully examined the differences and consulted the authors. We found, in addition to the differences due to the energy computation formula mentioned earlier, there are other possible reasons including different simulation windows, different implementations of the voltage changing mechanism in the latest distribution of the MCD simulation toolset [60], and different ways of computing average numbers for media benchmarks.

the voltage range in our experiments is only half that of the frequency range. As stated in [58], this reflects the current trend of shrinking voltage ranges in processor designs as the supply voltage continue to scale aggressively relative to the threshold voltage.

2.4.3 Energy-performance tradeoffs as a function of $\mathbf{q_{ref}}$

In the last subsection, the reference queue operating point q_{ref} for all benchmarks were set to the values in Table 2.1—roughly $\frac{1}{3}$ of the total size. Also, in Section 2.3.6, we have shown analytically how the performance and energy vary as a function of q_{ref} in general. In this subsection, we show that experimentally.

We use the 6 MediaBench benchmarks in the last subsection for this study, as they are relatively small. For each benchmark, we apply the *Analytic* online DVFS control with a relative q_{ref} varying from 0.0 to 1.0 (the same q_{ref} value for all three INT, FP, and LS queues). We then compute the performance and energy savings relative to the values for the baseline MCD.

Figure 2.13 shows the relative performance as a function of q_{ref} for 3 of the Media-Bench benchmarks. (Note, in our simulation, each domain has a frequency lower bound of 0.25GHz, so the lowest relative performance in Figure 2.13 is not 0.) From this figure we see that, due to different execution variation patterns in different applications, their performance curves have slightly different shapes and slopes. We also computed the average relative performance over all 6 MediaBench benchmarks, as shown in Figure 2.14, which has a shape relatively close to the analytic estimation in Figure 2.10.

Recall that from Section 2.3.6, the energy savings is also affected by q_{ref}. The general trend is that energy savings increase as q_{ref} increases. Figure 2.15 gives an illustration of this general trend using the average relative energy savings over the 6 MediaBench benchmarks.

Thus, we believe the results in this subsection and Section 2.3.6 show both analytically and experimentally how the energy performance tradeoff is affected by the q_{ref} setting in

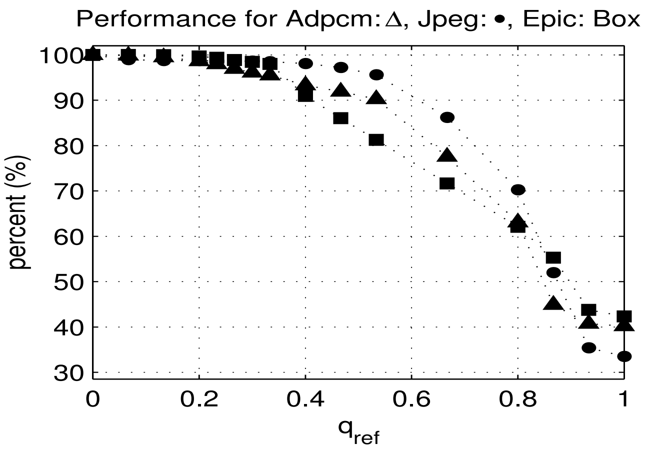

Figure 2.13: Relative performance degradation as a function of q_{ref} for individual benchmarks – adpcm-encode (△), epic-encode (□), and jpeg-encode (•).

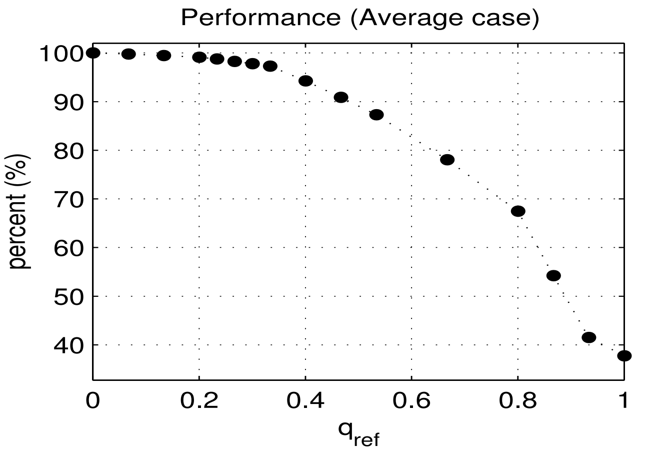

Figure 2.14: Average relative performance degradation over the 6 MediaBench benchmarks as a function of q_{ref}

our DVFS controller.

2.5 Centralized DVFS schemes: A Discussion

The online DVFS scheme studied in previous sections is decentralized. That is, it uses only local queue information and ignores interactions among multiple queues. The decentralized DVFS scheme can work fairly well in an MCD processor where frequency change in one clock domain has negligible or little impact on other domains and queues. For example, for the 4-domain MCD processor studied in Section 2.4 which has a fan-out structure,

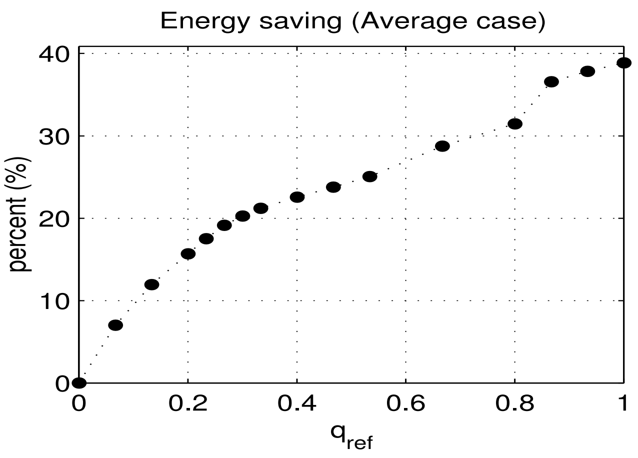

Figure 2.15: Average relative energy savings over the 6 MediaBench benchmarks as a function of q_{ref}

decentralized DVFS works quite well, as shown by the experimental results in Section 2.4.

However, for an MCD processor with more elaborate domain partitions and strong interactions among multiple queues, a centralized online DVFS scheme may be needed in order to make correct and efficient scaling decisions. For example, in a system where three clock domains are in series with two queues in between, the status of the first queue will not only depend on its neighboring domains, but also depend on the status of the downstream queues and domains. So, in this case, online DVFS needs to look at the status of all queues to make correct and efficient control actions. Otherwise, a clock domain can be confused by information from individual local DVFS controllers—one scenario is, when its input and output queues are both full, the local controller for its input queue will suggest a speed increase while the local controller for its output queue will suggest the opposite.

A systematic approach to design a centralized DVFS scheme needs to extend the DVFS framework in Section 2.3 using a global control theory. A new analytic model is first needed for all queues in an MCD processor which interact with each other. We can generalize the single queue-domain model used in previous sections into a queue-domain network. For each clock domain, its input queue can take flows (demand) from multiple sources (either a upstream domain or an external input source) through a *join* operation. Also, a clock domain can send out flows to multiple destinations (either a downstream domain or an external output sink) through a *split* operation. For this system, due to the interactions among multiple queues and domains, the actual execution speed μ and the arriving flow rate (demand) λ for a clock domain will be affected by the status of all related queues. For example, μ for a domain will be zero if its *downstream* queue is full. Similarly, λ for a domain will be zero if its *upstream* queue is empty. To analytically model these interactions, we need to use tools like an *indicate* function $\mathbf{1}()$ and introduce a *flow-matrix* A. The *indicate* function is defined as

$$\mathbf{1}(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases} \tag{2.14}$$

and the *flow-matrix* A defines how the tasks flow from one domain to another for a given queue-domain network.

With an analytical model for this system, control can be achieved by feeding back the state variables (the so-called state feedback with integral control [41]). Intuitively, this means frequency and voltage are controlled using all queue information as feedback signals. Note that, the design of an actual control law may need some linearization techniques in order to eliminate the nonlinearity due to presence of the *indicate* function.

The above extension to centralized DVFS control is one of the topics we are currently exploring. We have obtained a detailed analytic model and a practical global DVFS decision algorithm based on DVFS controllers.

2.6 Summary

We have presented an online scheme for dynamic voltage and frequency scaling (DVFS) in a multiple clock domain (MCD) processor. The proposed scheme takes a rigorous analytic approach and is guided by control theory.

We have described the modeling, analysis, design, and implementation of the proposed DVFS controller. Compared to the best-known prior online DVFS, which is heuristic-based, the proposed DVFS scheme has achieved a 2-3 fold increase in efficiency. In addition, the control theoretic technique is more resilient and complete. For example, we can guarantee stability and achieve significant energy savings even under extreme cases.

We believe the techniques and methodology described in this paper can be generalized for effective energy control in processors other than MCD. For example, there are many tiled CMPs in research now [22], and the ideas here would translate neatly to DVFS for them as well. Furthermore, the formal control techniques described here for DVFS can also serve as examples for applying control theory to other aspects of dynamic execution in high-performance CPUs.

Chapter 3

A Novel Energy Control Scheme With Adaptive Reaction Time

3.1 Introduction

Most existing hardware-based DVFS schemes (including the one in the previous chapter) use a fixed time interval to frame possible voltage/frequency changes. Specifically, during the time interval, certain system metrics or statistics are monitored, such as IPC [24] or average issue queue occupancy [58, 65]. At the end of the interval, the statistics from the current and past intervals are used to compute a new voltage/frequency setting for future intervals. One limitation of the above schemes is that the interval boundaries are predetermined and independent of workload changes. Thus, no matter what severe workload change occurs, the fixed-interval approaches wait and attempt to adjust voltage/frequency at the end of the interval. In addition, they might miss opportunities to respond to large activity swings inside the interval. One simple scenario for this case is that the workload increases dramatically in the first half-interval and decreases in the second half. Average statistics (for example the average queue occupancy) over the entire interval may not be able to capture this workload change.

Compared to the above fixed-interval schemes, in this work we propose an online DVFS scheme in which the reaction time for DVFS is not predetermined and is instead determined by the actual workload variation. In other words, the instant to react is *adaptive* to large, severe workload changes, so it could be more responsive. On the other hand, given no or only minor workload changes, an *adaptive* scheme will stay inactive for an arbitrarily long time, so it could also be more cost-effective.

Similar to the work in Chapter 2, we design our adaptive DVFS scheme in the context of a Multiple Clock Domain (MCD) architecture, which is described in Section 2.2. The triggering condition is based on recent instant queue occupancy for the issue queues. More specifically, two queue signals are monitored at each sampling period. The first one is the relative queue occupancy value with respect to a reference value, while the second one is the difference of queue occupancies between two sampling points. Based on these queue signals, appropriate DVFS decisions are made through a simple process, which also uses deviation window and resettable time delay to handle the noise problem and avoid unnecessary DVFS actions. After an initial design is done, we wish to obtain some analytical insights on whether or how the designed DVFS system will work and how to improve it. So we derive an aggregate continuous model for the designed DVFS controller and use it in a formal stability analysis. The analytic insight obtained is then used to guide and improve the design in terms of stability margin and control effectiveness. Finally, we evaluate our DVFS scheme through a cycle-accurate MCD simulator over a wide set of MediaBench and SPEC benchmarks.

Compared to the prior fixed-interval DVFS schemes for MCD [58, 65], the decision process for the new scheme is much simpler, and this leads to smaller and cheaper hardware. Yet, our scheme has achieved a significant amount of energy savings over all studied benchmarks (about 19% energy savings with 3% performance degradation on average, which is close to the best results from prior online DVFS schemes). In addition, our scheme is potentially more responsive. For a group of applications with fast workload variations, our scheme outperforms existing fixed-interval DVFS schemes significantly.

The rest of the chapter is structured as follows. Section 3.2 describes the detailed design of our adaptive DVFS scheme. In Section 3.3, we model the designed DVFS control and gain insights through a formal stability analysis. This is followed in Section 3.4 by experimental results. In Section 3.5, we highlight related work. Finally, Section 3.6 summarizes this chapter.

3.2 Design of Adaptive DVFS Scheme

One key design issue for adaptive DVFS is how sensitive it should be in responding to workload changes. Another design question is how big the frequency/voltage adjustment should be for a triggered action. So, before proceeding to an actual design, we first give some rationale and discussion of these key design issues.

As mentioned in the introduction, a variable-interval DVFS scheme responds immediately to large, severe workload changes. On the other hand, it will stay inactive for an arbitrarily long time given no or only minor workload change. Therefore, in general, the number of voltage/frequency adjustments for an adaptive DVFS scheme will ultimately depend on the pattern of workload change in a program. (This is different from that for the fixed-interval DVFS schemes where only one adjustment is possible for a fixed time interval.) However, for a given program, the number of adjustments will also depend on the setting of triggering conditions: what size of workload change should be treated as *severe* enough to trigger an action?

The main reason for which we want to respond only to severe workload changes is the DVFS switching cost, which includes both time and energy cost. (The switching cost is typically proportional to the magnitude of the switching.) The energy cost part for the DVFS switching comes from the voltage regulator (VR) [12]. Since the capacitor in VR is relatively small (most VRs are dual-phased so there are two output capacitors), the switch-

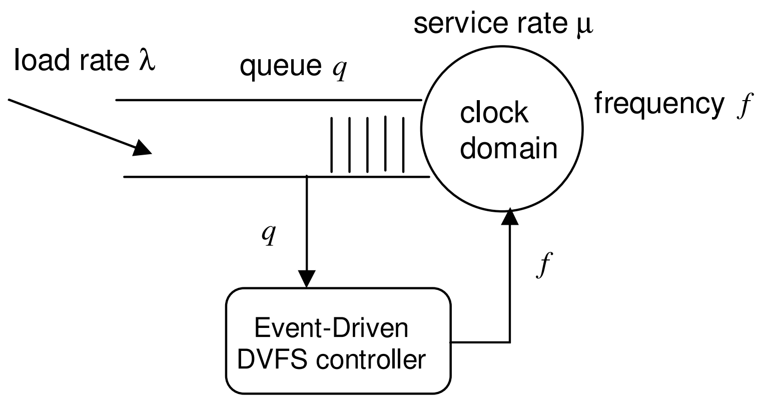

Figure 3.1: A local control model for a clock domain with an input queue.

ing energy cost is small and is ignored in most DVFS studies [19, 48, 58, 65]. The time cost is the main concern in DVFS switching. This is basically the transition/switching time and may include some amount of idle time for the processor waiting for PLL relocking [20].

Because of this DVFS switching cost, the adaptive DVFS action should be triggered only for large workload changes (in terms of magnitude and duration) such that the benefit brought by the DVFS is greater than the switching cost (i.e., there is a net gain in terms of energy-delay product improvement). Therefore, for adaptive DVFS control, the choices of the triggering condition and the amount of adjustment at each action should be based on the DVFS switching cost. For a DVFS implementation with relatively fast transition time and no (or very little) processor idle time (denote this group as XScale-style DVFS [20]), the triggering condition and adjustment step can be chosen as relatively low or small in order to have more frequent and fine-grained DVFS control. On the other hand, for a DVFS implementation with relatively slow transition time and long processor idle time (denote this group as Transmeta-style DVFS [58]), the triggering condition and adjustment step should be chosen as relatively high or big in order to reduce the switching overhead; and we will have less frequent and more coarse-grained DVFS control for this case.

3.2.1 A Design for MCD Processors

Conceptually, the online DVFS problem for an MCD processor is to adapt the frequency (and execution speed) to program phases and workload changes in each clock domain. In this work, similar to that in [36, 58, 65], we utilize interface queues to guide the DVFS control. Recall from Section 2.2 that there are interface queues between clock domains for synchronization. Intuitively, these queues give clues about the speed balance between the sender domain and the receiver domain. Therefore, an online DVFS controller can use this queue information, such as the fullness of queues and the rate of queue changes, to detect workload changes and respond to them by increasing or decreasing voltage and frequency in a clock domain.

In this work, we will only use local queue/domain information to direct DVFS control (a so-called decentralized control scheme). In other words, we assume the interactions between different queues and domains are weak and can be ignored. (This assumption is typically valid for an MCD implementation with relatively simple structure such as that in Figure 2.1.) A centralized DVFS scheme which utilizes all queue/domain information may work better, but is much harder to design, as it is still an open research problem.

Based on the above rationale, we will use the local queue information as trigger signals for the DVFS actions. Figure 3.1 shows such a local control model, where the circle represents a clock domain being controlled (such as a floating point function unit). The clock domain is connected to other domains (such as a decode/issue domain) through an interface queue (input or output) which has a finite size. The queue occupancy is sampled and used as a possible triggering signal by the DVFS controller, which controls the frequency (and hence the voltage) of the clock domain.

More specifically, our DVFS controller monitors two queue signals: the relative queue occupancy (q_i - q_{ref}) and the relative queue difference (q_i - q_{i-1}); where q_i represents the queue occupancy at the i_{th} sampling point, and q_{ref} is the reference queue occupancy, i.e. the target or nominal queue occupancy. A possible DVFS action will be triggered based on

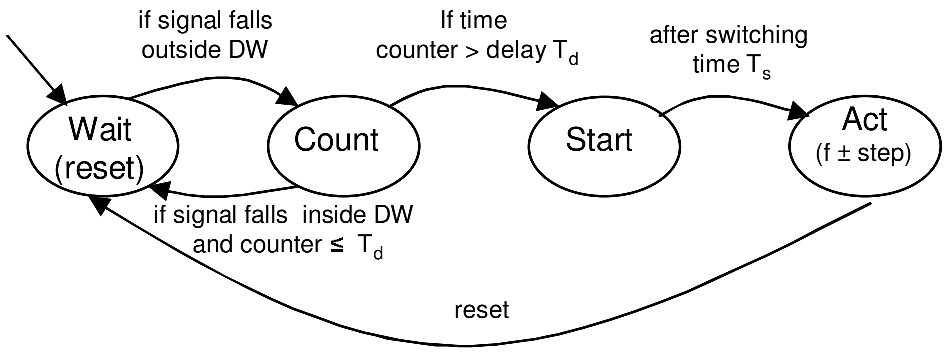
if signal falls outside DW
If time counter > delay T_d
after switching time T_s
Wait (reset)
Count
Start
Act (f ± step)
if signal falls inside DW and counter ≤ T_d
reset

Figure 3.2: The state transition graph for our DVFS control, where the *signal* refers to the queue signal ($q_i - q_{ref}$) or ($q_i - q_{i-1}$); DW is the deviation window.

these two queue signals. To simplify our design, we use a single step of incrementing or decrementing the clock frequency (and voltage accordingly) as the triggered DVFS action. As mentioned earlier, in general, the choice of the step size (and the time delay, which will be defined shortly) depends on the DVFS switching cost. For the MCD processor with an XScale-style DVFS model described in Section 2.2, we will choose relative small step sizes in order to have more fine-grained frequency adjustments. For processors with Transmeta-style DVFS model, the design framework in this section can still be used, but larger values should be chosen for the step size.

To handle the noise or random short-time variation in the queue occupancy and avoid unnecessary DVFS actions, we use a combination of deviation windows and time-delay relay. The deviation window (DW) is a small interval around the origin (denoted as $[-DW, +DW]$), while the time-delay relay is essentially a re-settable time counter. A signal will activate the time counter if the signal falls outside the deviation window. If a pre-set time-delay has passed, a possible DVFS action will be triggered. Note, for the time-delay, there are a number of design options. For example, it can be set as a simple constant (T_{d0}) , or a constant with a scaling factor depending on some system statistics. In our design, we design it as a constant with a scaling factor depending on signal values and current frequency setting.

Figure 3.2 shows the state transition graph for our DVFS design. In the figure, the *signal* refers to a queue signal (either $q_i - q_{ref}$ or $q_i - q_{i-1}$). Initially, the system is in a

Wait state. A transition into *Count* will occur if the queue signal falls outside the deviation window. The system will stay in the *Count* state until either the preset time delay (T_d) has passed and a transition is made to the *Start* state, or the queue signal falls inside the deviation window before the time delay has passed and a transition is made to the initial *Wait* state, which will reset the time counter. The *Start* state represents that a frequency (voltage) increment/decrement has been triggered or scheduled. Since it takes a certain amount of time to physically switch frequency/voltage, the increment/decrement action will be accomplished (in the *Act* state) after a switching time T_s. After that, the system transitions back to the *Wait* state, resets the time counter, and is ready for a new round of operations. The detailed finite state machine (FSM) graph for our DVFS controller is shown in Figure 3.3, where we shown increment and decrement actions separately.

In the above descriptions, for the sake of clarity, we assume that the two finite state machines (FSM), one for the trigger signal $(q_i - q_{ref})$ and one for the signal $(q_i - q_{i-1})$, operate independently. However, in practice, methods are required to reconcile the *Act* operations triggered by different queue signals in different FSMs. So we add a new state called *Schedule* into the above operation state graphs. If, at any time, only one queue signal is triggering the *Act* operation, then the *Scheduler* will initiate and *start* the action in the same way as we described before. On the other hand, if two queue signals are triggering DVFS actions at the same time, the system will schedule the two actions depending on the actions being triggered. Specifically, if two identical actions are being triggered (both are *Up* or both are *Down*), the system will schedule the two actions in sequence (or combine them into one action with a step size twice as big). On the other hand, if two opposite actions are being triggered (one is *Up* while the other one is *Down*), the system will cancel them all and reset both signals to the *Wait* state.

In the above design the reference queue value q_{ref} can be any value which is neither full nor empty. However, similar to the observation in Chapter 2, we notice that the position of q_{ref} specifies the actual tradeoff between performance degradation and energy saving.

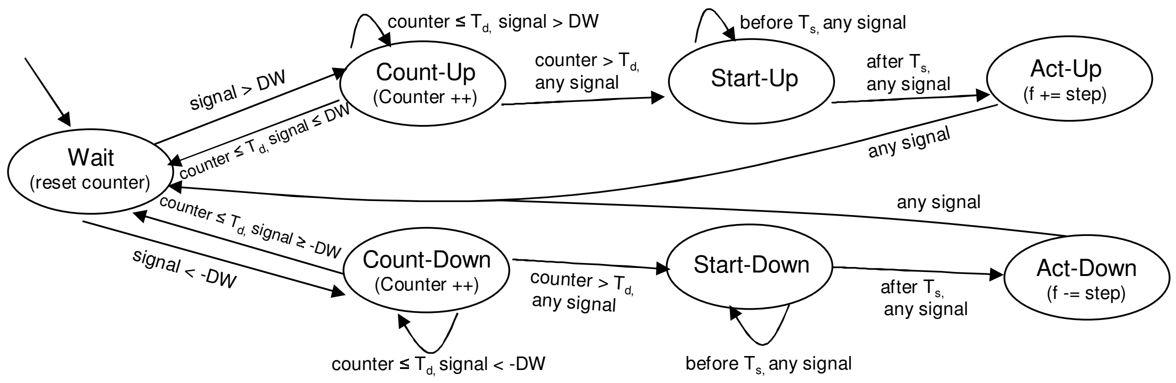

Figure 3.3: The detailed finite state machine (FSM) graph for our DVFS control, where the *signal* refers to the queue signal ($q_i - q_{ref}$) or ($q_i - q_{i-1}$); DW is the deviation window; T_d is the time delay; T_s is the switching time spent for one DVFS action.

We can increase q_{ref} to make the DVFS controller more aggressive in saving energy, or decrease q_{ref} to preserve performance more. The main reason is that the choice of the nominal operating point q_{ref} and its distances to the two queue end-points reflect the relative margin for the queue to tolerate control errors and input noise before the queue becomes full (will lose performance) or empty (will waste energy).

Next, we look at the issue of hardware implementation. From the description of the above design, we see the adaptive DVFS decision logic requires very little hardware in addition to the existing MCD hardware, since the interface queues and the voltage/frequency switching mechanism already exist in current MCD implementations [59]. The main additions are some book-keeping hardware. Figure 3.4 shows the block diagram for a possible hardware implementation of the basic DVFS decision logic. Specifically, an adder is used to compute the trigger signal (either $q_i - q_{ref}$ or $q_i - q_{i-1}$). Since a queue size is around 20 ($\ll 2^6$), a 6-bit adder is sufficient. Then, a 7-bit comparator is used to compare the trigger signal with the deviation window (DW). The rest is a 5-state finite state machine (FSM) and a time-delay counter. The FSM corresponds to the left five states in Figure 3.3. For the counter, if we assume the time delay in Figure 3.3 is ≤ 256, an 8-bit time-delay counter is sufficient. The two output signals of the FSM, *Start-up* and *Start-down*, will be used by the voltage (and frequency) switching mechanism in the voltage regulator. Note the hardware

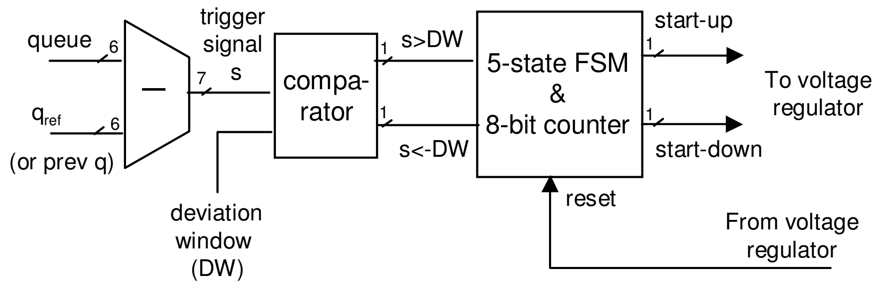

Figure 3.4: Block diagram for a possible hardware implementation of the basic DVFS decision logic.

requirement for this scheme is roughly in the same order as the book-keeping hardware (like counters) required by the fixed-interval DVFS schemes in [58] and Chapter 2. However, in those prior approaches, some extra hardware is required to compute appropriate voltage and frequency settings on a per-interval basis. This extra hardware is more complex than the book-keeping hardware discussed above (for example, multipliers/dividers or lookup tables are required to implement the PID controller in [65]). So, overall, the hardware requirement for the present scheme is much smaller and cheaper than those in the prior fixed-interval schemes.

3.3 Modeling and Stability Analysis of Adaptive DVFS System

In this section, we will look at the adaptive DVFS design from a control system point of view. The modeling and stability analysis are intended to gain insights and answer questions such as

- Will the above design work? Under extreme cases, can it lead to unbounded or unstable results?
- If it works, how well does it work and how can it be improved?
- More specifically, in order to improve the stability margin and control effectiveness, how should we choose the design options/parameters like the basic time delays?

(There are two queue signals, so there are two time delays to pick. Should we use the same amount of delay for both? If not,which one should be bigger or smaller?)

3.3.1 Overview of Modeling and Stability Analysis

This subsection gives an overview of modeling and analysis of the adaptive DVFS system without going through the derivation and stability analysis details, which will follow.

In order to analytically evaluate the DVFS design in the previous section, we need to model the DVFS control operation and the involved queue and clock domain dynamics. As mentioned earlier, we use the local control model in Figure 3.1. The clock domain has frequency f and execution capability (or service rate) μ, which is a function of f. The workload (or arrival rate) is denoted as λ. Note both μ and λ are time varying and we will denote them as $\mu_{(t)}$ and $\lambda_{(t)}$. Similarly, we denote the queue occupancy at time t as $q_{(t)}$. With these notations, we are ready to derive a model.

Recall from Section 3.2 that our DVFS controller for MCD processors has fine-grained *step* control. We can conveniently approximate the DVFS control by a continuous-time model[1]. This model is expressed as

$$\dot{f}_{(t)} = \frac{m}{h_{(f)}} \frac{step}{T_{m0}} (q_{(t)} - q_{ref}) + \frac{l}{h_{(f)}} \frac{step}{T_{l0}} \dot{q}_{(t)} \tag{3.1}$$

Intuitively, the above equation models the aggregate effects of the frequency control operation in Figures 3.2 and 3.3. The first part of the right hand side of the equation corresponds to frequency control operation for the queue signal $(q_i - q_{ref})$, while the second part corresponds to the operation for the queue signal $(q_i - q_{i-1})$. Specifically, in the equation, $\dot{f}$ is the time derivative of normalized frequency f (i.e. $\partial f/\partial t$); $q_{(t)}$ and q_{ref} are the queue and

[1]For processors with Transmeta-style DVFS, which require more coarse-grained *step* control, the modeling and analysis in this section can still be applied but will be less accurate. A similar but more complicated discrete-time model can be derived to get a better and more accurate analysis result. We leave this as possible future work.

reference queue occupancy as in Section 3.2; $\dot{q}$ is the time derivative of queue occupancy; *step* is the step size of the frequency change triggered by the queue signals; T_{m0} and T_{l0} are the basic time delays for the queue signals $(q_i - q_{ref})$ and $(q_i - q_{i-1})$ respectively; m and l are constants which are mainly from the conversion due to the different units used for queue occupancies and frequencies; $h_{(f)}$ is a function of f which is used to take account of possible affects of f on the effective time delay.

We then derive a model for the queue and clock domain dynamics. At an aggregate level, they are modeled as

$$\begin{aligned} \dot{q}_{(t)} &= \gamma\, \lambda_{(t)} - \gamma\, \mu_{(t)} \\ \mu_{(t)} &= \frac{1}{t_1 + \frac{c_2}{f_{(t)}}} \end{aligned} \tag{3.2}$$

Intuitively, the first equation in (3.2) means the queue occupancy change in a sampling time unit is equal to the number of arrived elements minus the number of departed/executed elements in that unit time (this is essentially a continuous-time version of the well-known Lindley equation [44]); γ is a constant which is proportional to the size of sampling period. The second equation models the execution/service rate μ in terms of clock frequency f, which is essentially a continuous-time version of the model used in [68] and Chapter 2. The t_1 and c_2 are constants whose meaning will be explained in detail in Section 3.3.3.

Putting together (3.1) and (3.2), we have a complete model for the involved DVFS controller, queue, and clock domain dynamics. This model is inherently nonlinear. To simplify the stability analysis, we linearize the system model through a standard nonlinear transformation technique [4] (which is essentially done by choosing an appropriate $h_{(f)}$ in (3.1) to compensate for the nonlinear function in (3.2) – details in Section 3.3.3). Then we proceed with some classic stability analysis for the linearized DVFS control system. Through the stability and control performance analysis, we have obtained the following:

- Remark 1: Given any non-zero values of *step* and basic time delays, the DVFS control system in Section 3.2 is stable. So, for any workload inputs, the DVFS controller would not lead to unbounded or unstable results.

- Remark 2: The control effectiveness of our design, in general, is mostly dependent on the values of time delays. A smaller time delay tends to improve the control response and settling time, and thus increase the control effectiveness. On the other hand, a smaller time delay will weaken the system's noise rejection ability, which may lead to more unnecessary/incorrect DVFS actions and thus reduce the overall DVFS efficiency. So there is a tradeoff between the control effectiveness and the system's noise rejection ability.

- Remark 3: In order to have small percent transient overshoot in system response, the values of the time delay for those two queue signals should be constrained by an inequality constraint (details in Section 3.3.3). With a typical system setting, this constraint implies that the time delay for the signal $(q_i - q_{ref})$ should be relatively larger than that for $(q_i - q_{i-1})$, and a setting of 2-8 time larger would typically lead to fairly good results.

In the rest of this section, we will give the derivation and analysis details. People who are already familiar with these subjects or who are not interested in these details may wish to skip them and go directly to the experimental evaluation in Section 3.4.

3.3.2 Modeling the Adaptive DVFS controller

The DVFS controller in Section 3.2 is essentially a discrete device with some inherent deviation windows and adjustable time delays. We will derive a model to capture the aggregate effect of its adaptive operations. We start by doing it for the queue signal $(q_i - q_{ref})$ only. Later, we will give a complete model for DVFS operations with both queue signals.

In aggregate, the frequency control operation with the queue signal $(q - q_{ref})$ in Figure 3.2 and 3.3 can be modeled as

$$f_{i+1} = f_i + step \cdot I(q_i - q_{ref}) \tag{3.3}$$

where

$$I(q_i - q_{ref}) = \begin{cases} 1 & \text{if } (q_i - q_{ref}) > DW \text{ for } (T_d + T_s) \\ -1 & \text{if } (q_i - q_{ref}) < -DW \text{ for } (T_d + T_s) \\ 0 & \text{Otherwise} \end{cases}$$

Intuitively, the above equation means a frequency increment or decrement will occur if the queue signal ($q_i - q_{ref}$) falls outside the deviation window for a consecutive ($T_d + T_s$) amount of time (in sampling period units). Otherwise, the frequency stays unchanged. In the equation, DW stands for Deviation Window; T_d is the time delay; T_s is the switching time; and others are defined the same as before.

In the DVFS design in Section 3.2, the discrete *step* was chosen to be relatively small in order to have more fine-grained frequency adjustments. With a small *step*, the above discrete model can be approximated by a continuous-time model as follows (we use a combined time delay $T_m = T_d + T_s$).

$$\dot{f}_{(t)} = \begin{cases} \frac{step}{T_m} & \text{if } (q_{(t)} - q_{ref}) > DW \\ \frac{-step}{T_m} & \text{if } (q_{(t)} - q_{ref}) < -DW \\ 0 & \text{if } |q_{(t)} - q_{ref}| \leq DW \end{cases} \tag{3.4}$$

The above model captures the aggregate effects of the frequency control operation by approximating the discrete-time step up or down action in (3.3) with a continuous-time linear increment or decrement action as illustrated in Figure 3.5 (the slope of the line is $\frac{step}{T_m}$). In the equation, $\dot{f}_{(t)}$ is the derivative of the frequency f at time t.

As mentioned in the last section, we choose the time delay T_m as a constant with a scaling factor which is negatively dependent on the absolute value of the queue signal.

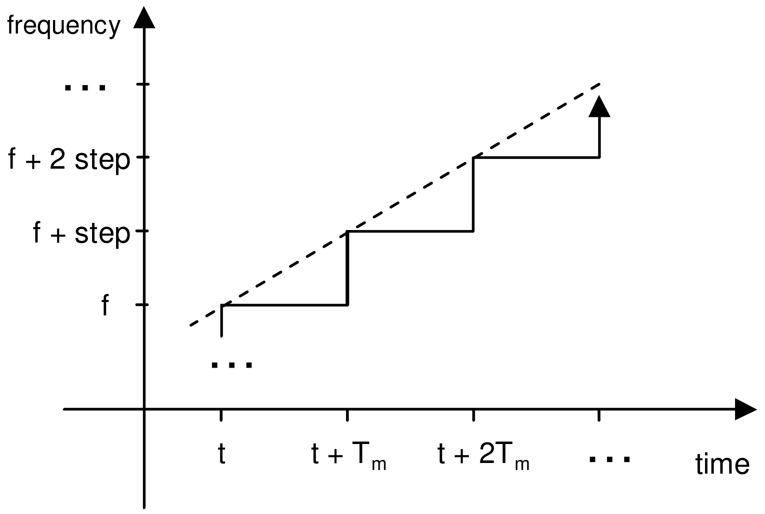

Figure 3.5: The approximation of the Step-up action modeled in (3.3) by a continuous-time linear increment action as modeled in (3.4).

That is, T_m is expressed as

$$T_m = T_{m0} \cdot \frac{1}{|q_{(t)} - q_{ref}| \cdot m} \tag{3.5}$$

The above equation means that when the value of the queue signal is larger, the time delay will be smaller and the DVFS controller will respond more quickly. Specifically, T_{m0} is the basic (constant) time delay; m is a constant which is mainly used to convert the value of queue occupancy to the scaling factor for time delay.

In addition, we augment the right side of equation (3.5) by a function $h_{(f)}$ to model possible impacts of frequency f on the time delay. We set $h_{(f)}$ to 1 if the level of frequency has no effect on the time delay.

Substituting the delay T_m in (3.5) into the continuous-time model in (3.4), assuming $DW \approx 0$, and rearranging algebraically, we have a new model in an ordinary differential equation (ODE) format as

$$\dot{f}_{(t)} = \frac{m}{h_{(f)}} \cdot \frac{step}{T_{m0}} \cdot (q_{(t)} - q_{ref}) \tag{3.6}$$

Like (3.4), the above equation models the aggregate effects of frequency control operation in Figures 3.2 and 3.3. However, compared to (3.4), this model is much simpler and more amenable to analysis.

We follow a similar procedure to derive another similar model for DVFS operations with the other queue signal $(q_i - q_{i-1})$. Since our DVFS controller is driven by both queue signals, we combine these two models to get a complete model for our DVFS controller.

$$\dot{f}_{(t)} = \frac{m}{h_{(f)}} \frac{step}{T_{m0}} (q_{(t)} - q_{ref}) + \frac{l}{h_{(f)}} \frac{step}{T_{l0}} \dot{q}_{(t)} \tag{3.7}$$

The above model captures the control effects of the DVFS operation with both trigger signals, $(q_i - q_{ref})$ and $(q_i - q_{i-1})$. (Note that this equation is the same as (3.1) in the overview sub-section and is repeated here for convenience). In the equation, $\dot{q}_{(t)}$ is derivative of queue value at time t; l and T_{l0} have similar interpretations as the m and T_{m0} in (3.6).

3.3.3 Stability Analysis of the Adaptive DVFS System

As described in the overview subsection, the modeling for the queue and clock domain dynamics is expressed by the following equations.

$$\dot{q}_{(t)} = \gamma\, \lambda_{(t)} - \gamma\, \mu_{(t)} \tag{3.8}$$

$$\mu_{(t)} = \frac{1}{t_1 + \frac{c_2}{f_{(t)}}} \tag{3.9}$$

Intuitively, the first equation means the queue occupancy change in a sampling unit time is equal to the number of arrived elements minus the number of departed/executed elements in that unit time. γ in the first equation is a constant proportional to the size of the sampling period. The second equation models the execution/service rate μ in terms of clock frequency f. This $\mu \sim f$ model is essentially a generalization of the model used in [68] and Chapter 2, which is discrete-time and models the average execution speed of a clock domain as a function of the average frequency in a time interval. The model in the prior work is based on the observation that, in most clock domains, execution time can be separated into two parts, one that is independent of clock frequency and one that is dependent. For example, in a load/store domain, the time spent for accessing asynchronous memory due to a cache miss is independent of domain frequency, while the time for querying and accessing a cache is dependent on frequency. Accordingly, in this model, t_1 is the average amount of unit time per instruction that is independent of frequency, and c_2 is the average number of frequency-dependent cycles per instruction (the value of t_1 and c_2 can be estimated online or offline using methods similar to those in [19, 68]). In our model in (3.9), we generalize the model into a continuous-time model by assuming that, at every sampling time unit, the $\mu_{(t)}$ and $f_{(t)}$ satisfies the same relationship as in the discrete $\mu \sim f$ model.

Putting together (3.7), (3.8) and (3.9), we have a complete model for the involved DVFS controller, queue, and clock domain dynamics. This model is inherently nonlinear. To simplify the stability analysis, we transform the equation (3.7) in terms of a new state variable of $\dot{\mu}_{(t)}$ through the equation

$$\begin{aligned} \dot{\mu}_{(t)} &= \tfrac{\partial \mu}{\partial f} \cdot \tfrac{\partial f}{\partial t} \\ &= \tfrac{c_2}{(t_1 f + c_2)^2} \cdot \dot{f}_{(t)} \end{aligned} \tag{3.10}$$

which is obtained by taking derivatives on both sides of (3.9). After this transformation, equation (3.7) becomes

$$\dot{\mu}_{(t)} = \frac{c_2}{(t_1 f + c_2)^2} \cdot \frac{1}{h_{(f)}} \left[\frac{m \cdot step}{T_{m0}} (q_{(t)} - q_{ref}) + \frac{l \cdot step}{T_{l0}} \dot{q}_{(t)} \right] \tag{3.11}$$

Like (3.7), this equation also models the frequency control in Figure 3.2 and 3.3, but is expressed in a different state variable $\mu_{(t)}$. We see that, in this equation, if we choose the function $h_{(f)}$ proportional to $\frac{c_2}{(t_1 f + c_2)^2}$, the non-linear part in it will be compensated for and the above equation becomes linear. Since the function $\frac{c_2}{(t_1 f + c_2)^2}$ is relatively complex to implement in practice, we will approximate it by a simpler quadratic function $\frac{k}{f^2}$ around the operating point. (Here, k is a constant factor dependent on the $\mu \sim f$ relationship; it can be estimated using t_1 and c_2 values.) Then, we can simply choose $h_{(f)} = \frac{1}{f^2}$ to linearize the equation (3.11).

Therefore, after the above linearization, we have a new system as

$$\begin{aligned} \dot{q}_{(t)} &= \gamma\, \lambda_{(t)} - \gamma\, \mu_{(t)} \\ \dot{\mu}_{(t)} &= \tfrac{m \cdot k \cdot step}{T_{m0}} (q_{(t)} - q_{ref}) + \tfrac{l \cdot k \cdot step}{T_{l0}} \dot{q}_{(t)} \end{aligned} \tag{3.12}$$

The above linearized system model is equivalent to the original system model in (3.7) - (3.9). Therefore, in order to understand the stability and transient behavior of our DVFS control system, we can conveniently analyze this linearized system model. By classic stability theory, the stability behavior of a linear system is decided by its characteristic roots [41]. For the system in (3.12), we can solve the characteristic roots as

$$s_{1,2} = \frac{-K_l}{2} \pm \frac{\sqrt{K_l^2 - 4K_m}}{2} \tag{3.13}$$

where $K_m = \frac{m \cdot \gamma \cdot k \cdot step}{T_{m0}}$ and $K_l = \frac{l \cdot \gamma \cdot k \cdot step}{T_{l0}}$.

Based on the above characteristic roots, we have the following observations and remarks.

Remark 1: With a typical system setting (non-zero parameters), the DVFS control system in Section 3.2 is stable. So, for any kind of workload inputs, our DVFS controller would not lead to unbounded or unstable results.

The above remark comes from the following. A linear system is stable if all its characteristic roots are negative (i.e. on the left side of the s-plane) [41]. With a typical system setting, all system parameters are non-zero and positive. So K_m, K_l will be positive. Then, from (3.13), we see both roots will be on the left side of the s-plane. Thus the DVFS control system is stable.

Remark 2: The control effectiveness of our design, in general, is mostly dependent on the value of time delays. A smaller time delay tends to improve the control response and settling time, and thus increase the control effectiveness. On the other hand, a smaller time delay will weaken the actual system's noise rejection ability (not modeled by the analytical model), which may lead to more unnecessary/incorrect DVFS actions and thus reduce the overall DVFS efficiency. So there is a tradeoff between the control effectiveness and the system's noise rejection ability.

The above remark is based on the following. Though any positive K_m and K_l values would make the system stable, the control effectiveness is dependent on the actual values of K_m and K_l. Among all the parameters in the K_m and K_l definitions, the basic time delays T_{m0} and T_{l0} are the most adjustable parameters in the design space. So, in general, the control effectiveness is mostly dependent on the values of the time delays. More specifically, the control effectiveness is typically characterized by the *settling time* (t_s) and the *rising time* (t_r) of the system unit-step response [41]. For this system, we have $t_s = \frac{8}{K_l}$ and $t_r = \frac{0.8}{\sqrt{K_m}} + \frac{1.25K_l}{K_m}$ using the formulas in [41]. So, smaller time delays T_{m0} and T_{l0} (thus larger K_m and K_l) will improve the rising and settling time, and increase the control effectiveness. On the other hand, in our DVFS system, the time delays are used to handle the noise (i.e. the random short time input variation) and avoid unnecessary DVFS actions. So, smaller time delays will weaken the system's noise rejection ability, and may lead to more unnecessary/incorrect DVFS actions. Therefore, the time delays should be chosen to balance the control effectiveness and the noise rejection.

Remark 3: In order for the system to have relatively small transient overshoot in the system response, the values of the time delay for the two queue signals should be constrained by an inequality. With a typical system setting, this constraint implies that the time delay for the signal ($q_i - q_{ref}$) should be relatively larger than that for ($q_i - q_{i-1}$), and a setting of 2-8 time larger would typically lead to fairly good results.

The above remark is based on the following. In this system, the maximum percent transient overshoot is decided by a parameter called damping ratio $\xi = \frac{K_l}{2\sqrt{K_m}}$ [41]. To have a small percent overshoot (say $\leq 15\%$) and a good rising time also, we have an inequality constraint of $0.5 \leq \xi \leq 1$ for this system. Substituting the ξ in the above inequality with K_l and K_m, we have $\frac{K_l^2}{4} \leq K_m \leq K_l^2$. With a typical system setting, we have $K_l < 1$. Combining the above two inequalities, we have $K_m < K_l$. Assuming all other parameters (such as *step*) are the same for the two queue signals, the above inequality implies $T_{m0} > T_{l0}$. In other words, the time delay for the signal $(q_i - q_{ref})$ should be relatively larger than that for $(q_i - q_{i-1})$. For example, if $K_l = \frac{1}{2}$, we have $\frac{1}{16} \leq K_m \leq \frac{1}{4}$. So the time delay T_{m0} is 2-8 times larger than T_{l0}, which would typically lead to fairly good transient control response. Inside this range, we can take a value close to the upper end (8 times slower) if overshoot is the major concern, or take a value close to the low end (2 times slower) if rising time is the major concern.

So far, we have analytically evaluated the adaptive DVFS design in terms of stability and control effectiveness. In the next section, we will experimentally evaluate the DVFS design and check how well it works in practice.

3.4 Experimental Results

In this section, we will show experimental results for evaluation. We will compare our results to those from the conventional synchronous voltage/frequency scaling and to those from two existing fixed-interval DVFS schemes for MCD processors (the work in [58] and the work in Chapter 2, which is also in [65]). At the end of this section, we will also compare our results to Chapter 2 with different and shorter interval lengths.

Table 3.1: Summary of All Simulation Parameters

Simulation Parameters	**Value**
Domain frequency range	250MHz – 1.0GHz
Domain voltage range	0.65V - 1.20V
Frequency/voltage change speed	73.3 ns/MHz, 171 ns/2.86mV
Signal sampling rate	250MHz
Time delays (sampling)	$T_{l0} = 10$, $T_{m0} = 50$
Step size (f/v)	2.3MHz/2.86mv
Reference queue point	7 INT, 4 FP, 4 LS
Deviation window (DW)	$\pm$1 or 0
Domain clock jitter	$\pm$110ps, normally distributed
Inter-domain synchro window	300ps
Branch predictor:	
2-level	L1 1024, hist 10, L2 1024
Bimodal	size 1024, BTB 4096 sets, 2-way
Combined	size 4096
Decode/Issue/Retire width	4/6/11
L1 data cache	64KB, 2-way
L1 instr cache	64KB, 2-way
L2 unified cache	1MB, direct mapped
Cache access time	2 cycle L1, 12 cycles L2
Memory access latency	80 first chunk, 2 inters
Integer ALUs	4 + 1 mult/div unit
Floating-point ALUs	2 + 1 mult/div/sqrt unit
Issue queue size	20 INT, 16 FP, 16 LS
Reorder buffer size	80
LS retire buffer size	64
Physical Register file size	72 INT, 72 FP

3.4.1 Simulation Methodology and Setup

Our simulation environment is based on that in Chapter 2, which is in turn based on the SimpleScalar toolset [13] with the Wattch [10] power estimation extension and the MCD processor extension [48, 59]. The MCD extension by Semeraro *et al.* in [48, 59] has 4 clock domains as shown in Figure 2.1.

We implemented our DVFS controller for local queues and domains, following the design and analysis in Sections 3.2 and 3.3 . Also, as in previous work on DVFS in MCD [48, 58, 65], we made the front end domain run at a fixed maximum speed, and allowed the INT, FP, and LS domains to be controlled by the DVFS controller. Since we are assuming an aggressive XScale-like DVFS model as described in Section 2.2, we choose a fine-grained step size for the frequency increment or decrement (2.4MHz/step, so it takes 320 steps to traverse the total frequency/voltage range). We choose a sampling rate of 250MHz for the queue signals, which corresponds to the lower bound of the frequency range $250MHz - 1GHz$. Based on the remarks 2 and 3 in Section 3.3, we choose the basic time delays for the queue signals $(q_i - q_{i-1})$ and $(q_i - q_{ref})$ as $T_{l0} = 8$ and $T_{m0} = 50$ respectively (in units of sampling period). We emulate the signal-dependent time delay by having larger time-counter increments in Figures 3.2 and 3.3 for larger signal values. The time delay for *counting-down* is also scaled by a factor of $(\frac{1}{f^2})$, where f is the relative frequency using $f_{max} = 1.0GHz$ as the base. So, for *counting-down* in a clock domain, the time delay would be larger with a lower frequency level and the system would be more cautious in further scaling down the clock frequency. We choose a q_{ref} of 6 for the INT clock domain which is roughly $\frac{1}{3}$ of its total queue size; and a q_{ref} of 4 for the FP and LS clock domains which are $\frac{1}{4}$ of their total queue sizes. These numbers are chosen to make the overall performance degradation around 5%. For the deviation window DW, we choose $DW = \pm 1$ for the queue signal $q_i - q_{ref}$, and $DW = 0$ for the signal $q_i - q_{i-1}$. Also, we assume an aggressive clock gating that is applied whenever the unit is not used. All other architecture parameters are chosen to have the same values as those in [48, 58, 65]. A summary of all simulation parameters is shown in Table 3.1.

As an illustrative example, Figure 3.6 shows the frequency settings in the FP domain obtained from our DVFS controller for the MediaBench benchmark Epic-decode. We chose this benchmark because its FP issue queue has a very simple workload pattern, in which the queue is emptying except for two distinct phases, as observed in [58, 65]. From this figure, we see, in the beginning stage, the adaptive DVFS controller detected the queue emptiness and quickly reduced the FP frequency to $f_{min} = 0.25GHz$. The first non-empty workload phase occurred around the moment at 2500k instructions when a modest workload increase was detected and the frequency recovered from $f_{min} = 0.25$ to a value around 0.6. After

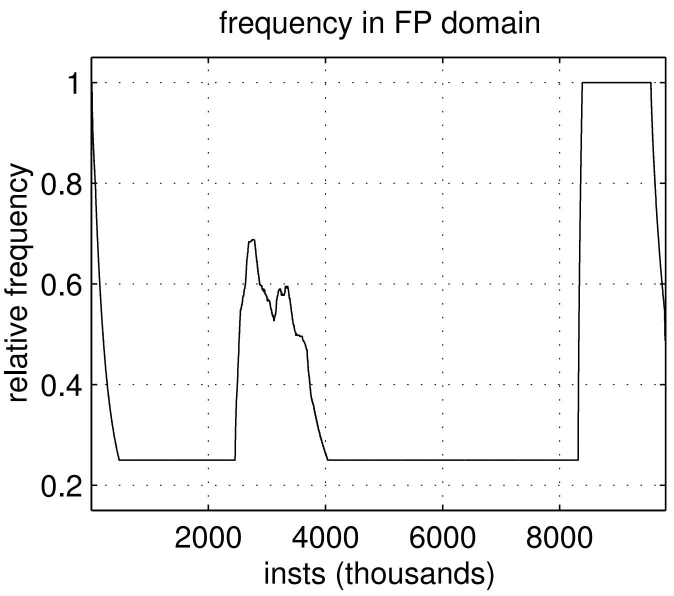

Figure 3.6: Frequency settings obtained from our adaptive DVFS in the FP clock domain for the benchmark of Epic-decode

.

this recovery, the frequency gradually dropped to $f_{min} = 0.25$ again as the queue decreased and became complete empty again around the moment at 4000k instructions. The queue stayed empty until a dramatic increase occurred around the moment at 8200k instructions. The adaptive DVFS controller detected this dramatic increase of queue entries and quickly adjusted the clock frequency to $f_{max} = 1.0GHz$.

3.4.2 Benchmark Classifications

We want to evaluate our adaptive DVFS scheme with a broad set of applications. To show variety, we will present results for 6 MediaBench, 6 SPEC2000int, and 5 SPEC2000fp applications as shown in Table 3.2. We chose roughly the same subset of SPECint and SPECfp as those used in [48, 58, 65]. For MediaBench benchmarks, we use the official data input set in the MediaBench web site and the whole program as the simulation window; for SPEC2000 benchmarks, we use the reference input set and choose the simulation window using the published Early SimPoint numbers [56].

Before we start to show the energy/performance results from our DVFS scheme and

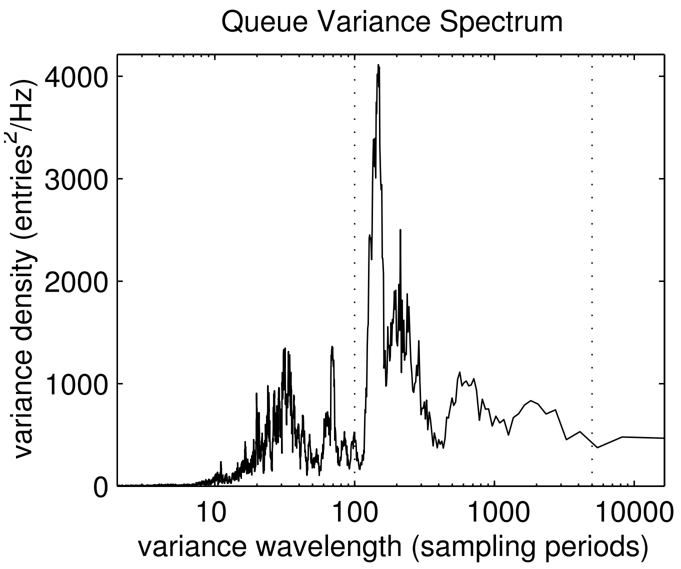

Figure 3.7: Variance spectrum for queue entries in the INT domain for the benchmark Epic-decode. The dotted line marks the interesting frequency/wavelength range used to identify *fast* workload variations.

compare them to those from prior fixed-interval DVFS schemes, we want to first look at some benchmark characteristics which affect the performance of the current and prior online DVFS schemes. As mentioned in Section 3.1, the current scheme has potentially better responsiveness due to its adaptive nature and is more suitable for applications with fast workload changes. Therefore, one benchmark characteristic we might wish to look at is the workload variability. In the rest of this subsection, we will study this characteristic and identify applications with relatively fast workload variations.

We use the queue occupancies to characterize the program workload. A metric which we can use to justify the application workload as varied is the overall variance observed in the queue occupancy. However, there is a problem with this metric: it only reflects the total workload variations, and not necessarily the *fast* variation. To overcome this problem, we make use of spectrum or spectral analysis [55].

The spectrum of a time series is the distribution of variance as a function of variance frequency [55]. We will denote the workload variance frequency as ω to distinguish it from the clock frequency f. Note the basic components for spectrum are the sinusoidal waves of different frequency or wavelengths. Spectral analysis estimates and computes the variance associated with each frequency component. (The method we use is the Multi-taper method [55] which utilizes the fast Fourier transform during the estimation process.)

With the spectrum or spectral density function (spectral density is in terms of variance per unit frequency), we can get the variance associated with any range of frequencies by integrating the spectral density over the increment of frequency ω. Therefore, in order to identify fast workload variations, we can compute and inspect the queue variance associated with high frequencies or short wavelengths only. The question is how to quantify *fast*, *high* or *short*.

Since we are studying adaptive DVFS as compared to fixed-interval DVFS (which is assumed to have a fixed interval of length N, $N = 5 - 10k$ sampling periods normally), we define any queue variance component with a wavelength of N or less as *fast* or *short*. This is because any queue swings inside the control interval (i.e. with a wavelength of N or less) will not be captured by the fixed-interval schemes (as the swings offset by the averaging in the fixed-interval schemes). In addition, queue variance components with extremely short wavelengths are considered noise and are ignored (i.e. they are too fast to be captured by either approach). We choose 100 sampling periods as the noise drop-off line because the basic time-delay for our DVFS scheme is about 50.

Therefore, to identify fast workload variations, we compute and inspect queue variance with wavelengths in the range of $[100, N]$, where N is the interval length for the fixed-interval schemes. As an illustrative example, Figure 3.7 shows the variance spectrum of the queue entries in the INT domain for the benchmark Epic-decode. For convenience, we show the spectrum as a function of the wavelength, rather than the frequency ω. The queue variance in the interesting frequency range (marked by the dotted line in Figure 3.7) can then be computed.

In Table 3.2, we have computed the queue variance associated with the defined interesting frequency/wavelength range for all the benchmarks. Based on these variance numbers

Table 3.2: Classification of benchmarks based on workload variation characteristics (numbers are in units of queue entry square)

Benchmarks	**Queue variance**			**Classification**
	INT	**FP**	**LS**	
epic-decode	17.2	0.1	7.3	
jpeg-decode	12.4	0.0	8.0	
jpeg-encode	14.4	0.0	10.1	*Group1*
172.mgrid	1.7	14.5	5.7	(fast
173.applu	0.8	16.8	8.3	workload
176.gcc	19.1	0.0	7.1	variation)
183.equake	16.7	0.0	5.1	
186.crafty	14.4	0.0	7.0	
197.parser	13.6	0.0	6.6	
adpcm-decode	2.9	0.0	0.5	
adpcm-encode	7.6	0.0	0.8	
epic-encode	4.1	3.1	1.6	*Group2*
164.gzip	9.4	0.0	4.5	(slow or
171.swim	1.0	1.8	1.3	negligible
179.art	4.2	4.6	2.9	variation)
181.mcf	5.1	0.0	4.1	
256.bzip2	13.3	0.0	2.1	

(larger number means more workload variation), we classify the benchmark set into two groups with roughly same number of applications: *group1* with relatively large workload variations and *group2* with relatively small or negligible variations in the defined frequency range. (Specifically, in Table 3.2, INT and LS numbers are used to classify integer benchmarks; while FP and LS numbers are used to classify floating-pointing benchmarks.)

The above benchmark characterization and classification will let us have a better and deeper understanding of the experimental results in the next subsection.

3.4.3 Energy and Performance Results

We will present energy and performance results from our new adaptive-reaction DVFS scheme (denoted as *adaptive*), and compare them to existing fixed-interval DVFS schemes for MCD processors [58, 65]. The work in [58] is one of the best-known DVFS schemes for MCD, and is based on a heuristic called *AttackDecay* (denoted as *fixed-heuristic*). The

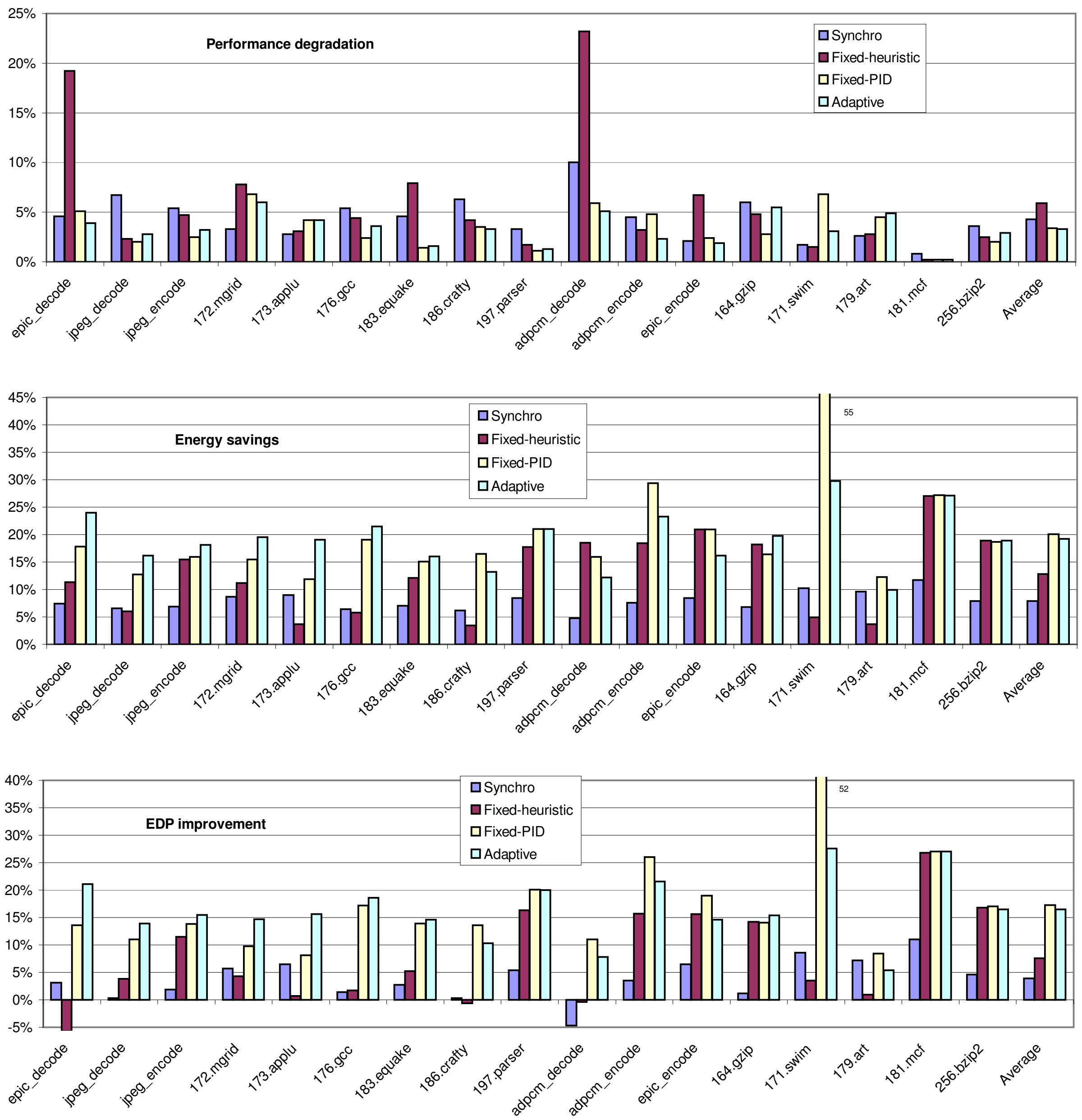

Figure 3.8: Performance degradation, energy savings, and energy-delay product (EDP) improvement for each benchmark; different schemes are *Synchronous* voltage scaling, two fixed-interval DVFS schemes (*fixed-heuristic* and *fixed-PID*), and the DVFS scheme with *adaptive* reaction time.

parameters for *fixed-heuristic* here are taken from [58], with a 5% performance degradation target because this target value will lead to an actual performance degradation similar to *adaptive*. The other scheme to compare with is the previous work in Chapter 2 (also in [65]). Since this work uses a Proportional Integral Derivative (PID) based DVFS con-

Table 3.3: Average results over *group1* benchmarks

Schemes	Performance degradation	Energy savings	Energy-Delay product improvement
Synchro	4.7%	7.0%	2.6%
fixed-heuristic	5.9%	9.6%	4.3%
fixed-PID	3.2%	16.2%	13.5%
adaptive	3.3%	18.7%	16.0%

troller, we denote it as *fixed-PID*. Similarly, the parameters for *fixed-PID* are taken from Chapter 2 and [65]. For the sake of completeness, we also compare our results to conventional (fully) synchronous voltage/frequency scaling (denoted as *Synchro*) which scales the frequency/voltage for the whole processor[2].

The performance degradation, energy savings, and energy-delay product (EDP) improvement for each benchmark are shown in Figure 3.8. (The benchmarks are arranged with *group1* on the left, and *group2* on the right.) The results are relative to the conventional (fully) synchronous processor without voltage scaling. So all results for MCD include about 1.5% percent inherited performance and energy overhead from the baseline MCD processor, as discussed in [58]. The average energy savings and performance loss for *adaptive* over all 17 benchmarks are 19.2% and 3.3% respectively. This gives an EDP improvment of 16.5% for *adaptive*, as compared to 3.9% for *Synchro*, 7.6% for *fixed-heuristic* and 17.3% for *fixed-PID*. So we see, despite its simpler design and hardware requirement, the *adaptive* results on average are quite close to the best from existing approaches (16.5% vs. 17.3%)

Furthermore, if we look at the energy/performance results for individual benchmarks, we have more interesting observations. As we expected, nearly all benchmarks in *group1* favor *adaptive* over other schemes. Table 3.3 shows the average results over the benchmarks in *group1* for different approaches. From Table 3.3, we observe that, for *group1*

[2]We were not able to implement synchronous dynamic voltage scaling, so the *Synchro* results are for static scaling which may under-represent the benefit of synchronous voltage scaling slightly.

Table 3.4: Average results over *group2* benchmarks

Schemes	Performance degradation	Energy savings	Energy-Delay product improvement
Synchro	3.9%	8.4%	4.8%
fixed-heuristic	5.6%	16.3%	11.6%
fixed-PID	3.7%	24.5%	21.7%
adaptive	3.2%	19.7%	17.1%

benchmarks, *adaptive* has achieved significantly better energy efficiency than fixed-interval DVFS schemes. The EDP result from *adaptive* is about 18% better than that from *fixed-PID* (16.0% vs. 13.5%), and nearly 3-fold better than that from *fixed-heuristic* (16.0% vs 4.3%). This shows the efficiency of our DVFS design and its self-tuning reactive advantages.

Next, we look at energy/performance results for the *group2* benchmarks, which have relatively slow or negligible workload variations. Table 3.4 show the average results over the *group2* benchmarks for different approaches. We observe that the *adaptive* results are still significantly better than those from the *Synchro* and *fixed-heuristic* (we attribute this to the MCD advantage and/or the effective design in Section 3.2 which is guided by formal stability analysis). However, compared to the *fixed-PID* result, the *adaptive* result lags by about 21% (17.1% vs. 21.7%). To understand the reason, we see that, for the *group2* benchmarks, the *adaptive* scheme gets little or no useful advantages over *fixed-PID* from responsiveness. On the other hand, *fixed-PID* still has the advantage over *adaptive* in terms of more accurate/intelligent DVFS decisions. Recall that, for *adaptive*, to keep the design simple and reduce the overhead of the decision process, we used simple time delays and single frequency increment or decrement for choosing possible DVFS actions. In general, DVFS actions picked by this mechanism are not as accurate/intelligent as those chosen by the *fixed-PID*, which may utilize system information from the current and past time-intervals and the PID controller to figure out more accurate voltage and frequency settings (of course, this requires more hardware). Therefore, for the *group2* benchmarks, the advantages associated with *fixed-PID* prevail and lead to better results for these than *adaptive*.

3.4.4 Comparison to *Fixed-PID* with Shorter Intervals

In previous sections, all *fixed-PID* results were obtained using the default interval-length in [65] ($10k$ instructions, which is also the interval length for *fixed-heuristic* in [58]). The question is what the results would be like for *fixed-PID* if a very short interval were chosen? Also, how do the *adaptive* results compare to those from *fixed-PID* with shorter intervals?

To answer the above questions, we first look at the pros and cons of reducing the interval-length for *fixed-PID*. On one hand, smaller interval length tends to make DVFS control in *fixed-PID* more fine-grained, which may lead to more energy savings. On the other hand, the energy overhead associated with the DVFS decision process and trigger logic in *fixed-PID* also increases significantly with shorter intervals. As mentioned in Section 3.2, the decision logic in the *fixed-PID* can be separated into two parts. The first part is the logic being executed on a per-cycle basis (such as the counters and other book-keeping logic). The second part is the logic being executed on a per-interval basis; this is the part to implement the PID controller, and is doing more complicated computations such as multiplication and division. Therefore, if the interval is relatively long, the amortized decision logic overhead will be small and negligible. However, if the interval becomes shorter and shorter, the amortized overhead will become larger and more significant.

Furthermore, even if we do not count the above decision overhead, the control efficiency for *fixed-PID* may still diminish with extremely short intervals. This is because the noise rejection ability of *fixed-PID*, which works like a low-pass filter using the average queue occupancy over the entire interval, begins to deteriorate with very short intervals.

Therefore, because of the above pros and cons, reducing the interval length may not lead to increased energy efficiency for *fixed-PID*. In general, there exists a range of medium interval-length values for which the *fixed-PID* has the best overall control performance. (The above discussion is also true for *fixed-heuristic*. The author of [57] showed that their

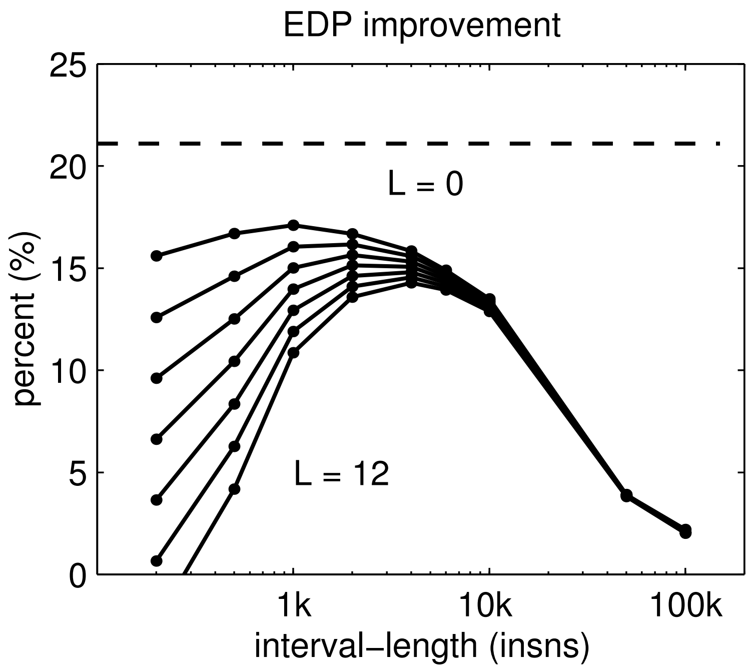

Figure 3.9: The energy delay product improvement (EDPI) for the benchmark Epic-decode obtained from *fixed-PID* (solid lines) as a function of interval lengths, and the increasing hardware complexity level L. We also show the EDPI from *adaptive* (the dashed line) as a comparison

DVFS algorithm works best for a range of interval lengths around 10k instructions).

Figure 3.9 shows the EDP improvement from the *fixed-PID* controller as a function of different interval lengths, and different decision logic overhead, for the benchmark Epic-decode. L in the figure is the scaling factor accounting for different level of hardware complexity of the *fixed-PID* decision logic[3]. ($L = 0$ is for the ideal and unrealistic case where the PID controller has zero energy cost). For comparison, we also show the EDP improvement from the *adaptive* for Epic-decode (the dashed line).

From Figure 3.9, we see the best control performance did not occur at very short intervals (like 1000 instructions or shorter). Rather, it occurred with an interval length somewhere in the medium range depending on the actual hardware complexity level.

[3]Specifically, L is used to scale the unit energy cost for *fixed-PID* decision logic in each time interval. The unit energy cost is estimated as $3 \cdot \alpha \cdot L \cdot C_0 \cdot V^2 \cdot k$, where *3* accounts for 3 clock domains; $\alpha = 0.5$ is the active factor, $L \cdot C_0$ is the total capacitance for the PID controller with C_0 chosen as 1% of the total capacitance of the integer ALU; V is the voltage in the front end; k is the total number of cycles the PID controller takes to finish the computation.

We have also measured the efficiency of the *fixed-PID* with shorter intervals for other benchmarks, and compared them to results from *adaptive*. Overall, the observations and findings above hold for them as well.

3.5 Related Work

In this section, we highlight important related work that we have not discussed in previous sections.

As mentioned earlier, most existing hardware-based online intra-task DVFS work uses a fixed window or time-interval. There is little prior work in the direction of adaptive DVFS design. One pioneering work in that direction is the *Mode-switching* algorithm proposed by Iyer and Marculescu [36]. Their algorithm detects the *High* or *Low* queue occupancies (note they only look at the static queue occupancies, and do not use information on queue occupancy changes). If a *High* or *Low* queue condition has been detected for some consecutive cycles, a possible switch from *Fast-mode* to *Slow-mode* (or vice versa) will be made. We see, though the reaction time in their work is essentially adaptive, the simple two-mode switching algorithm has not been able to fully utilize the power/benefits brought by this paper's approach. For the purpose of comparison, we have re-implemented their *Mode-switching* DVFS algorithm and incorporated it into our simulation infrastructure. Our experimental results show that, in general, an energy-delay product improvement of $6 - 7\%$ has been achieved (over the base case without DVFS). In addition, without more detailed analysis, it is not analytically clear how to further improve [36].

Two other DVFS algorithms for MCD architectures which we have not discussed are the *Shaker* algorithm in [59] and the profile-based algorithm in [48]. They both use offline analysis to obtain optimal DVFS settings for a program. In constrast, the focus of this work is on architectural level hardware-based online DVFS schemes.

Recently, there have been increasing research efforts in applying control theory or other system theories in CPU design and control [62, 65]. One example is the applying of control theory in thermal control [62]. Another example is the previous work described in Chapter 2, which is on applying control theory to DVFS control for MCD processors (also in [65]). The theoretical analysis in [65] is close to this work since the same architecture and the same system dynamics are being considered. However, the focus of the theoretical analysis in [65] is on the design, that is, how to model the controlled system and use control theory as guidances in designing a standard PID-based controller. On the other hand, the theoretical part of this work is focused on modeling and stability analysis of an existing DVFS design. So in this work we derive a mathematical model for an existing design to be used in the analysis. Then, the obtained analytic insight is used to tune and improve the original design.

3.6 Summary

This chapter presents a novel DVFS scheme for multiple-clock-domain (MCD) processors. Compared to existing control schemes, which use fixed control intervals, this new interval-less scheme has reaction time that is not predetermined, but self-tuned and adaptive to application and workload changes.

Overall, we feel the proposed adaptive DVFS scheme is a promising alternative to the existing fixed-interval DVFS schemes. Designers may choose the new scheme for processors with limited hardware budget, or if the type of application behavior is known in advance to have high workload variability. In addition, the modeling and analysis techniques in this work serve as examples of using stability analysis in other aspects of high-performance CPU design and control.

Chapter 4

A Dynamic Compilation Framework for Controlling Microprocessor Energy and Performance

4.1 Introduction

The previous chapters have discussed architectural level hardware-based energy control techniques. In addition to hardware, the compiler and system software can also play an important role in energy and performance management. While existing software-driven energy control techniques are primarily based on OS time-interrupt [19, 64] or static compiler techniques [31, 68], very little has been done to explore DVFS control opportunities in a dynamic compilation or optimization environment. In this chapter, we consider dynamic compiler DVFS techniques, which optimize the application binary code and insert DVFS control instructions at program execution time.

A dynamic compiler is a run-time software system that compiles, modifies, and optimizes a program's instruction sequence as it runs. In recent years, dynamic compilation is becoming increasingly important as a foundation for run-time optimization, binary translation, and information security. Examples of dynamic compiler based infrastructures include HP Dynamo [5], IBM DAISY [21], Intel IA32EL [6], and Intel PIN [47]. Since most DVFS implementations allow direct software control via mode set instructions (by accessing special mode set registers), a dynamic compiler can be used to insert DVFS mode set instructions into application binary code at run time. If there exists CPU execution slack (i.e., CPU idle cycles waiting for memory), these instructions will scale down the CPU voltage and frequency to save energy with no or little impact on performance.

Using dynamic compiler driven DVFS offers some unique features and advantages not present in other approaches. Most importantly, it is more fine-grained and more code-aware than hardware or OS interrupt based schemes. Also it is more adaptive to the run-time environment than static compiler DVFS. In Section 4.2, we will give statistical results to further motivate dynamic compiler driven DVFS, and discuss its advantages and disadvantages as compared to existing techniques.

This work presents a design framework of the run-time DVFS optimizer (RDO) in a dynamic compilation environment. Key design issues that have been considered include code region selection, DVFS decision, and code insertion/transformation. In particular, we propose a new DVFS decision algorithm based on an analytic DVFS decision model. A prototype of the RDO is implemented and integrated into an industrial-strength dynamic optimization system (a variant of the Intel PIN system [47]). The obtained optimization system is deployed into a real hardware platform (an Intel development board with a Pentium-M processor), that allows us to directly measure CPU current and voltage for accurate power and energy readings. The evaluation is based on experiments with physical measurements for over 40 SPEC or Olden benchmarks. Evaluation results show that significant energy efficiency is achieved. For example, up to 70% energy savings (with 0.5% performance loss) is accomplished for SPEC benchmarks. On average, the technique leads to energy delay product (EDP) improvements of 22.4% for SPEC95 FP, 21.5% for SPEC2K FP, 6.0% for SPEC2K INT, and 22.7% for Olden benchmarks. These average results are 3X-5X better

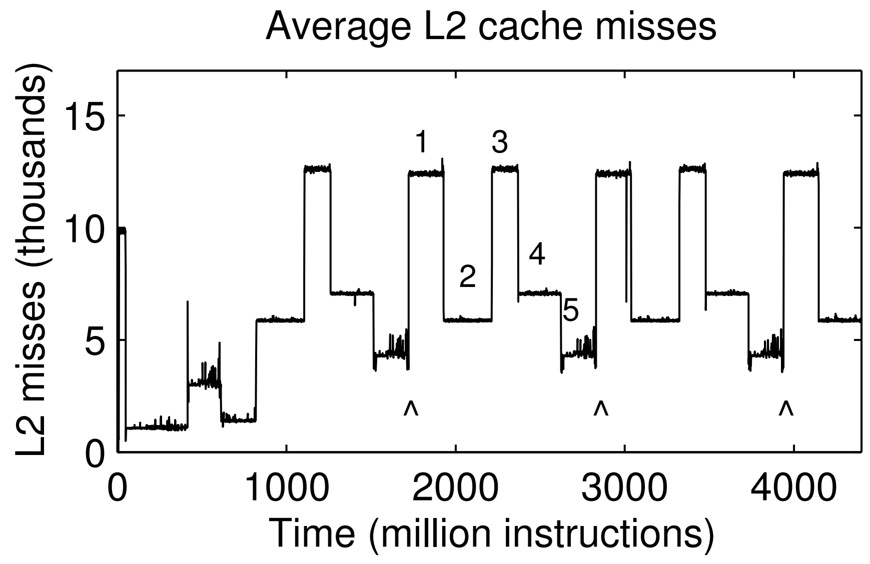

Figure 4.1: Number of L2 cache misses for every million instructions retired during the execution of SPEC2000 benchmark 173.applu. Numbers 1 to 5 mark five memory characteristic phases. The symbol ˆ in the figure marks the recurring point of the program phases.

than those from static voltage scaling (22.4% vs. 5.6%, 21.5% vs. 6.8%, 6.0% vs. -0.3%, 22.7% vs.6.3%), and are more than 2X better (21.5% vs. 9%) than those reported by a static compiler DVFS scheme [31].

The structure for the rest of the chapter is as follows. Section 4.2 further motivates dynamic compiler DVFS. Section 4.3 presents the design framework of the RDO. Section 4.4 describes the implementation and deployment of the RDO system. This is followed by the experimental results in Section 4.5. Section 4.6 highlights related work. Section 4.7 discusses possible future work. Finally, Section 4.8 summarizes this chapter.

4.2 Why Dynamic Compiler Driven DVFS?

In this section, we discuss in more detail the unique features, advantages, and disadvantages of dynamic compiler driven DVFS, as compared to existing DVFS techniques.

4.2.1 Advantages over hardware or OS DVFS

Existing hardware or OS time-interrupt based DVFS techniques typically monitor some system statistics (such as issue queue occupancy [58]) in fixed time intervals, and decide DVFS settings for future time intervals [19, 49, 58, 64]. Since the time intervals are predetermined and independent of program structure, DVFS control by these methods may not be efficient in adapting to program phase changes. One reason is that program phase changes are generally caused by the invocation of different code regions, as observed in [33]. Thus, the hardware or OS techniques may not be able to infer enough about the application code attributes and find the most effective adaptation points. Another reason is that program phase changes are often recurrent (i.e., loops). In this case, the hardware or OS schemes would need to detect and adapt to the recurring phase changes repeatedly.

To illustrate the above reasoning, Figure 4.1 shows an example trace of program phase changes for the SPEC2000 benchmark 173.applu. (The trace is from a part of the program with about 4.5 billion instructions.) The y-axis is the number of L2 cache misses for every 1M instructions during the program execution. (All results in the section were obtained using hardware performance counters in a processor setup described in Section 4.5.) From the figure, we see that there are about 5 distinct memory phases characterized by different L2 cache miss values and duration. As will be shown in Section 4.5, these phases actually correspond to 5 distinct code regions (functions) in the program. Also, we see the phase changes are recurrent, as shown by the marked points in the figure.

Compiler driven DVFS schemes (static or dynamic) can apply DVFS to fine-grained code regions so as to adapt naturally to program phase changes. Hardware or OS-based DVFS schemes with fixed intervals lack this fine-grained, code-aware adaptation.

4.2.2 Advantages over static compiler DVFS

Existing compiler DVFS work is primarily based on static compiler techniques [31, 68]. Typically profiling is used to learn about program behavior. Then some offline analysis

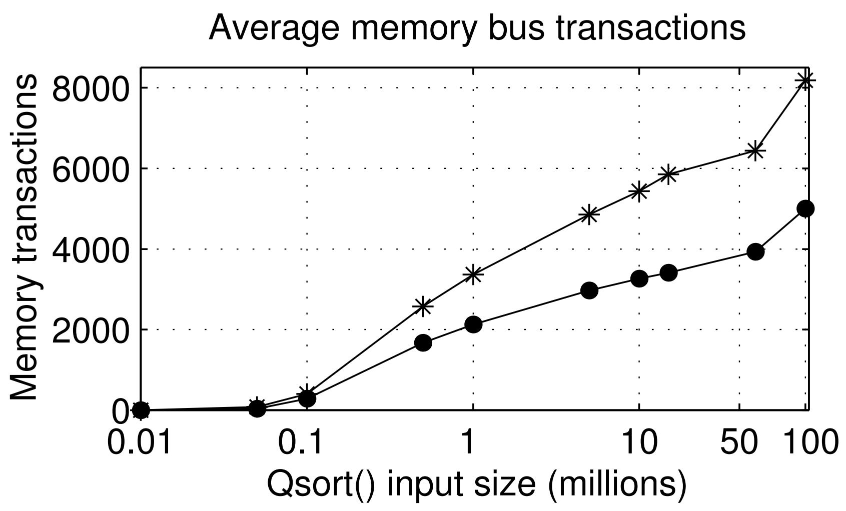

Figure 4.2: Average number of memory bus transactions (per 1M instructions) for the function qsort() (as in stdlib.h) with different input sizes and different input patterns (random input:• versus pre-sorted input: $*$).

techniques (such as linear programming [31]) are used to decide DVFS settings for some code regions.

One limitation to static compiler DVFS is that, due to different runtime environments for the profiler and the actual program, the DVFS setting obtained at static compile time may not be appropriate for the program at runtime. The reasoning is that DVFS decisions are dependent on the program's memory boundedness (i.e., the CPU can be slowed down if it is waiting for memory operation completion). Then, the program behavior in term of memory boundedness is in turn dependent on run-time system settings such as machine/architecture configuration, program input size and patterns. For example, machine/architecture settings such as cache configuration or memory bus speed may affect how much CPU slack or idle time exists. Also, different program input sizes or patterns may affect how much memory is to be used and how it is going to be used.

As an illustration, Figure 4.2 shows the average number of memory bus transactions (per 1M instructions) for the function qsort() (as in the stdlib). The curve with • is for random input elements with different input sizes, while the curve with $*$ is for pre-sorted input elements. Figure 4.2 shows that the average numbers of memory bus transactions vary significantly for different input sizes and input patterns. (Not surprisingly, larger input sizes lead to more L2 cache misses and thus more memory bus transactions.)

While the above example is from a small program for illustration, Table 4.1 shows examples of different memory behavior from the SPEC programs with reference or train inputs. We show the average number of L2 cache misses and the average number of memory bus transactions (per 1M instructions) for some example code regions. Similarly, we see these numbers may become very different if input is changed from reference to train, or vice versa.

Based on the above observation, we see, for different input sizes or patterns, different DVFS might be needed to have the best energy/performance results. For the qsort() example in Figure 4.2, a more aggressive (i.e., lower) DVFS setting should be used for a large input size (like 100M) to take advantage of the memory boundedness in the program and save more energy. Conversely, a more conservative (i.e., higher) DVFS setting should be used for a small input size (like 10K) to avoid excessive performance losses. For the SPEC benchmark 103.su2cor in Table 4.1, the code region in the function sweep() might need different DVFS settings for different input sets. Our empirical experience on the Intel Pentium-M processor shows that, assuming a performance loss constraint of 4%, this code region can be clocked at 1.0Ghz with the reference input, while it has to be clocked at the maximum 1.6Ghz for the train input.

While it is inherently difficult for a static compiler to make DVFS decisions adaptive to the above factors, dynamic compiler DVFS can utilize run-time system information and make input-adaptive and architecture-adaptive decisions.

4.2.3 Disadvantages and challenges

Having discussed its advantages, we would like to point out that dynamic compiler DVFS also has its disadvantages. The most significant one is that, just as for any dynamic optimization technique, every cycle spent for optimization might be a cycle lost to execution

Table 4.1: Examples of different memory behavior for SPEC programs with reference and train inputs.

L2: Average Num of L2 cache misses per 1M instructions
Mem: Average Num of memory bus transactions per 1M inst
4L sweep(): means 4th loop in function sweep()

Benchmark	Code region	reference		train	
		L2	Mem	L2	Mem
103.su2cor	corr()	3.9K	13.6K	1.4K	5.7K
103.su2cor	4L sweep()	4.3K	14.4K	1.8K	6.1K
107.mgrid	mg3p()	2.6K	9.5K	0.6K	2.3K
189.lucas	fftSquare()	6.8K	18.1K	0.06K	0.1K
256.bzip2	DoRevers-Transform()	7.8K	11.9K	1.3K	3.1K

(unless optimizations are performed as side-line optimizations on a chip multi-processor [11]). Therefore, one challenge to dynamic compiler driven DVFS is to design simple and inexpensive analysis and decision algorithms in order to minimize the run-time optimization cost.

4.3 Design Framework and DVFS Decision Algorithms

In this section, we present a design framework for the run-time DVFS optimizer (RDO) in a dynamic compilation and optimization environment. We start by considering some key design issues in general. Then we give the detailed design of a new DVFS decision algorithm.

4.3.1 Key design issues

Candidate code region selection: Like other dynamic optimization techniques, we want only to optimize those frequently executed code regions (so-called hot code regions), in order to be cost effective. In addition, since DVFS is a relative slow process (the voltage transition rate is typically around $1mv/1\mu s$), we also want to only optimize long-running code regions. In our design, we choose functions and loops as candidate code regions.

Since most dynamic optimization systems are already equipped with some light-weight profiling mechanism to identify hot code regions (for example DynamoRio profiles every possible loop target [11]), we will extend the existing profiling infrastructure to monitor and identify hot functions or loops.

DVFS decisions: For each candidate code region, an important step is to decide whether it is beneficial to apply DVFS to it (i.e., whether it can operate at a lower voltage and frequency to save energy without significant impact on the overall performance) and what the appropriate DVFS setting is. As we mentioned earlier, for a dynamic optimization system, the analysis or decision algorithm needs to be simple and fast to minimize overhead. Thus, the offline analysis techniques used by static compiler DVFS [31] are typically too time-consuming and are not appropriate here. For our work, we have designed a fast DVFS decision algorithm, which is based on an analytical decision model and uses hardware feedback information.

DVFS code insertion and transformation: If a candidate code region is found beneficial for DVFS, DVFS mode set instructions will be inserted at every entry point of the code region to start the DVFS, and at every exit point of the code region to restore the voltage level. One design question is how many adjusted regions we want to have in a program. Some existing static compiler algorithms choose only a single DVFS code region for a program [31] (to avoid the excessively long analysis time). In our design, we will allow/identify multiple DVFS regions to provide more energy saving opportunities. But things become complicated when two DVFS regions are nested. (If a child region is nested in a parent region, then a child may not know the DVFS setting for the parent at its exit points.) We provide two design solutions. One is to maintain a relation graph at run time, and only allow the parent region to be scaled. The other one is to have a DVFS-setting stacked so that both the parent and the child regions can be scaled. In addition to the code insertion, the dynamic compiler can also perform code transformation to create more energy saving opportunities. One example is to merge two separate (small) memory bound

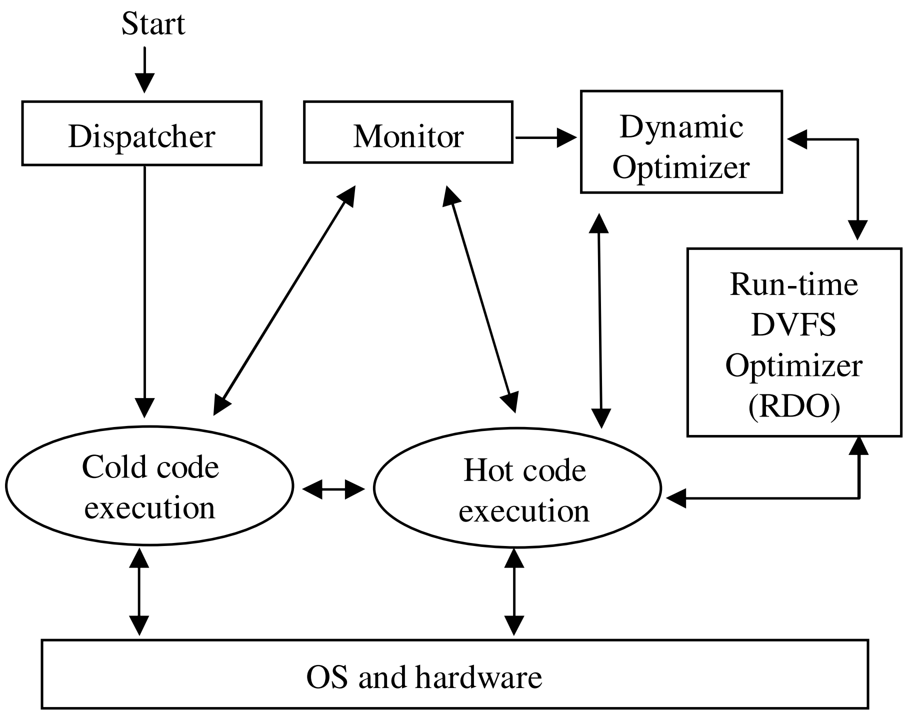

Figure 4.3: The overall block diagram showing the operation and interactions among different components of a dynamic compiler DVFS optimization system.

code regions into one big one. Of course, we need to check that this code merging does not hurt the performance (or the correctness) of the program. So there will exist interactions among the DVFS optimizer and the conventional performance optimizer.

Overall operation block diagram: The block diagram in Figure 4.3 shows the overall operation and interactions between different components of a dynamic compiler DVFS optimization system. At the start, the dynamic optimizer dispatches or patches original binary code and delivers the code to execution by the hardware. At this moment, the system is in a *cold-code* execution mode. While the cold code is being executed, the dynamic optimization system monitors and identifies the frequently executed or hot code regions. Then, the RDO optimization is applied to the hot code regions, either before or after the conventional performance optimizations have been conducted. (For a hot code region, the RDO optimization can be applied once per program execution, or multiple times — which we call periodic re-optimization.) Lastly, if a code transformation is desirable, the RDO will query the regular performance optimizer to check the feasibility of the code transformation.

Next, we describe in detail a key design component: the DVFS decision algorithm.

4.3.2 DVFS decision algorithms

To make DVFS decisions, RDO first inserts some testing and decision code at the entry and exit points of a candidate code region. (A candidate region can be viewed as a single entry, multiple exits code region.) The testing and decision code collects some run-time information (such as number of cache misses or memory bus transactions for this code region). If enough information has been collected, RDO decides the appropriate DVFS setting for a candidate code region based on the collected information and the RDO setup. After a decision is made, RDO removes the testing and decision code and prepares for possible DVFS code insertion and transformation.

The above testing steps assume that a candidate code region has relatively stable or slowly-varying run-time characteristics for a given input. Therefore, the obtained decision based on the testing information will be valid for the rest of the program execution, or valid until the next re-optimization point if we choose periodic re-optimizations. Note that this assumption has been shown reasonable or valid in practice by studies such as [66].

The key testing step in the above is the DVFS decision making. As we mentioned earlier, in order to be beneficial for DVFS, a code region first needs to be long-running, which can be easily checked. The harder question is, for a long running code region, how to decide whether it is beneficial to have DVFS and what an appropriate DVFS setting is. To answer this question, we first look at an analytical decision model for DVFS.

An analytical decision model for DVFS

The discussion and analysis model in this section assume that the goal of our energy control is to minimize the energy consumption, *subject to some performance constraints*. (Note that the analytical model for a different objective, such as thermal control, might be differ-

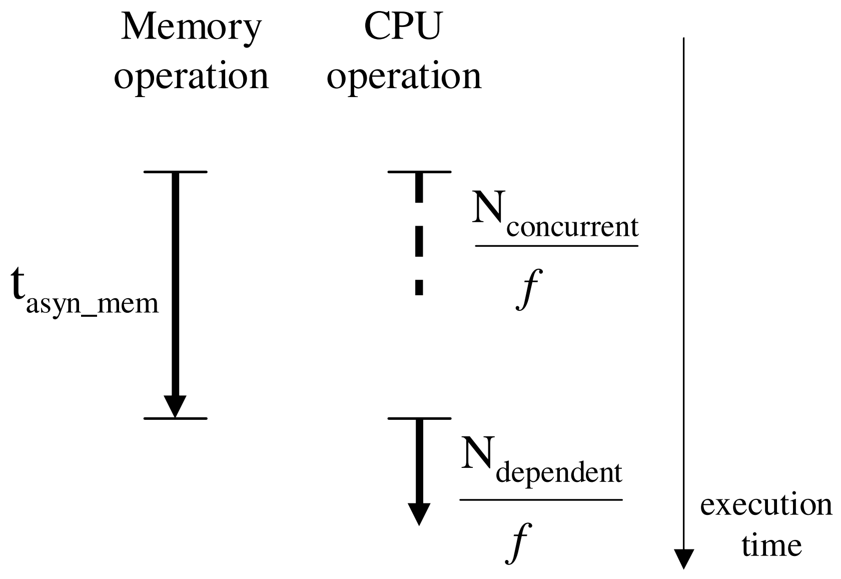

Figure 4.4: An analytical decision model for DVFS. t_{asyn_mem} is the asynchronous memory access time, $N_{concurrent}$ is the number of execution cycles for the concurrent CPU operation, $N_{dependent}$ is the number of cycles for the dependent CPU operation, f is the CPU frequency.

ent.)

In general, scaling down the CPU voltage and frequency will certainly reduce processor power consumption, but it will also slow down the CPU execution speed (and the resulting energy delay product improvement might be low or even negative). The key insight to a beneficial DVFS (which saves energy but with no or little performance impact) is that there exists an asynchronous memory system, which is independent of the CPU clock and is many times slower than the CPU. Therefore, if we can identify the CPU execution slack (i.e., CPU stall or idle cycles waiting for the completion of memory operations), we can scale down the CPU voltage and frequency to save energy without much performance impact.

Based on the above rationale, Figure 4.4 shows our analytical decision model for DVFS, which is an extension of the analytical model proposed in [68]. As in Figure 4.4, the processor operations are categorized into two groups: memory operation and CPU operation. Since memory is asynchronous with respect to the CPU frequency f, we denote the time for memory operation as t_{asyn_mem}. The CPU operation time can be further separated into two parts: part 1 is those CPU operations that can run concurrently with memory operations, and part 2 is those CPU operations that depend on the final results of the pending memory operations. Since the CPU operation time is dependent on the CPU frequency f, we denote the concurrent CPU operation time as $N_{concurrent}/f$, where $N_{concurrent}$ is the number of clock cycles for the concurrent CPU operation. Similarly, we denote the dependent CPU operation time as $N_{dependent}/f$. (In actual program execution, the memory operation and the CPU operation, either concurrent or dependent, will be interleaved somehow. However, for an analytical model, we abstract the execution model by lumping all the occurrences of each category together. This is the same treatment as in [68].)

From Figure 4.4, we see if the overlap period is memory bound, i.e $t_{asyn\_mem} > \frac{N_{concurrent}}{f}$, there exists a CPU slack time defined as

$$\text{CPU slack time} \;=\; t_{asyn\_mem} - \frac{N_{concurrent}}{f} \qquad (4.1)$$

Ideally, the concurrent CPU operation can be slowed down to consume the CPU slack time.

With the above model, we want to decide the frequency scaling factor β for a candidate code region. (So, if the original clock frequency is f, the new clock frequency will be $\beta\ f$; and the voltage will be scaled accordingly.) We assume the execution time for a candidate code region can be categorized according to Figure 4.4. So frequency scaling will have two effects on the CPU operation. First, it will increase the concurrent CPU operation time and reduce the CPU slack time (if any). Second, it will dilate the dependent CPU operation time, which will cause performance loss unless $N_{dependent} = 0$.

Next we will give a detailed method to select or compute the scaling factor β.

DVFS selection method

We introduce a new concept called relative CPU slack time. Based on the definition of CPU slack time in (4.1), we define

$$\text{relative CPU slack time} \;=\; \frac{t_{asyn\_mem} - N_{concurrent}/f}{total\_time} \qquad (4.2)$$

where the $total\_time$ is the total execution time in Figure 4.4. For a memory bound case, $total\_time = t_{asyn\_mem} + N_{dependent}/f$. From Figure 4.4, we see the larger the relative CPU slack, the more frequency reduction the system can have without affecting the overall performance. So the frequency reduction (i.e., $1 - \beta$) is proportional to the relative CPU slack time. We have

$$(1-\beta) \;=\; k_0 \left(\frac{t_{asyn\_mem}}{total\_time} - \frac{N_{concurrent}/f}{total\_time} \right) \tag{4.3}$$

where k_0 is a constant coefficient. Note that the value of k_0 can be chosen to be either relatively large to have more aggressive energy reduction, or relatively small to preserve performance more. Therefore, to take into account the effect of the maximum allowed performance loss P_{loss}, we replace k_0 in (4.3) by $k_0\ P_{loss}$, and we have

$$\beta \;=\; 1 - P_{loss}\, k_0 \frac{t_{asyn\_mem}}{total\_time} + P_{loss}\, k_0 \frac{N_{concurrent}/f}{total\_time} \tag{4.4}$$

Intuitively, the above equation means the scaling factor is negatively proportional to the memory intensity level (the term with t_{asyn_mem}), and positively proportional to the CPU intensity level (the term with $N_{concurrent}$). The time ratios in the above equation can be estimated using hardware feedback information such as hardware performance counter (HPC) events. For example, for an x86 processor, the two time ratios in the above equation can be estimated by ratios of some HPC events [25].

$$\frac{t_{asyn\_mem}}{total\_time} \;\simeq\; k_1 \frac{\text{Num\_of\_mem\_bus\_transactions}}{\text{Num\_of\_}\mu ops\text{\_retired}} \tag{4.5}$$

$$\frac{N_{concurrent}/f}{total\_time} \;\simeq\; k_2 \frac{\text{Num\_of\_FP\_INT\_instructions}}{\text{Num\_of\_}\mu ops\text{\_retired}} \tag{4.6}$$

where in (4.5) the first HPC event is the number of memory bus transaction, which is what we have used in Section 4.2 to measure memory busy-ness. The second HPC event is the total number of μops retired. The ratio of these two events is used to estimate the relative memory busy-ness. Similarly, in (4.6), the first HPC event is the number of FP/INT instruction retired (while there is an outstanding memory operation). The second event is also the number of μops retired. The ratio of these two HPC event is used to estimate the concurrent CPU busy-ness. Like k_0 in (4.3), k_1 and k_2 in the above are constant coefficients which depend on machine configurations and can be estimated empirically and reset at the installation time of a dynamic compiler.

Because the above method computes β directly from some run-time hardware information, it is simple and fast. The downside is that the formulation is relatively ad-hoc, especially the way it considers the constraint P_{loss}. We have also developed an alternative method which is more precise in handling the performance constraint P_{loss}, but is more complicated. We see this alternative method as a complement to the above method.

4.4 Implementation and Deployment: Methodology and Experience

We have implemented a prototype of the proposed run-time DVFS optimizer (RDO), and integrated the RDO into a real dynamic compilation system. To evaluate it, our results present live-system physical power measurements.

4.4.1 Implementation

We use the Intel PIN system [47] as the basic software platform to implement our DVFS algorithm and develop the RDO. PIN is a dynamic instrumentation and compilation system developed at Intel and is publicly available. The PIN system which we use is based on the regular PIN but has been modified to be more suited and more convenient for dynamic optimizations. (For convenience, we refer it as O-PIN, i.e., Optimization PIN.) Compared to the standard PIN package, O-PIN has added more features to support dynamic optimizations, such as adaptive code replacement (i.e., the instrumented code can update and

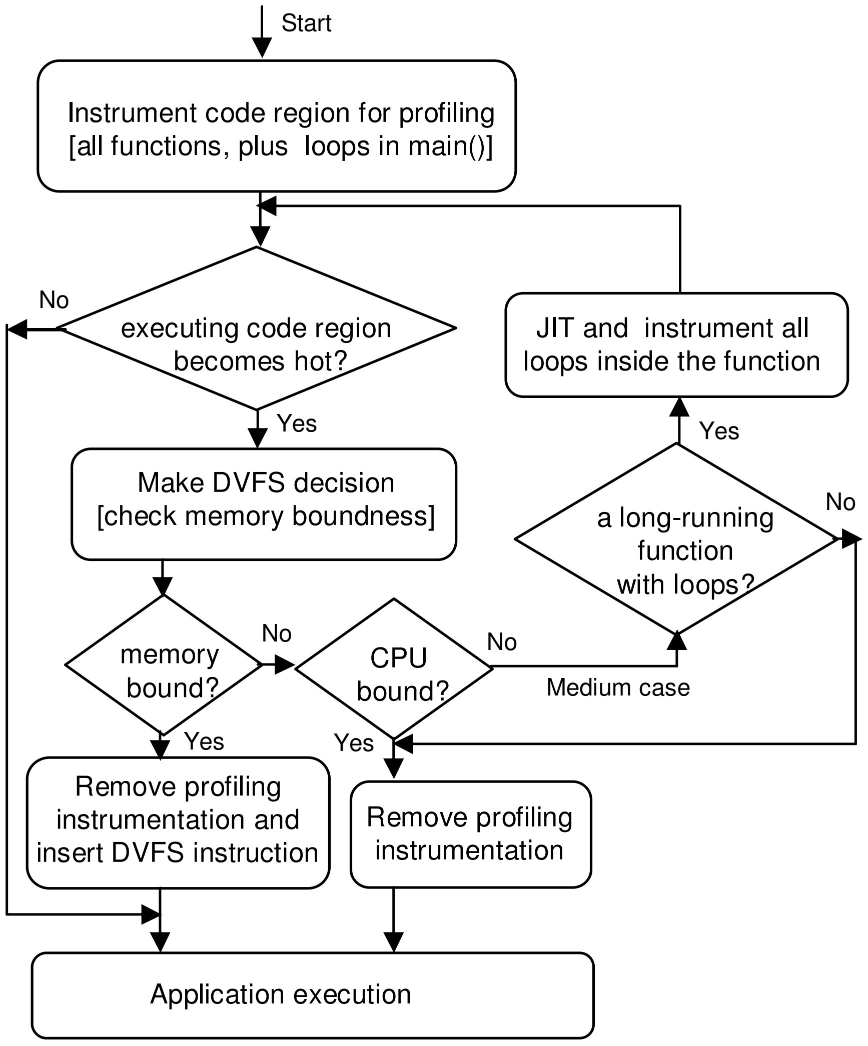
Start
Instrument code region for profiling [all functions, plus loops in main()]
executing code region becomes hot?
No
Yes
JIT and instrument all loops inside the function
Make DVFS decision [check memory boundness]
a long-running function with loops?
Yes
No
memory bound?
No
CPU bound?
No
Medium case
Yes
Yes
Remove profiling instrumentation and insert DVFS instruction
Remove profiling instrumentation
Application execution

Figure 4.5: The operation flow diagram for our prototype implementation of the RDO.

replace itself at run time) and customized trace or code region selection. In addition, unlike the basic PIN which is JIT-based and executes the generated code only [47], O-PIN takes a partial-JIT approach and executes a mix of the original code and the generated code. For example, O-PIN can be configured to first patch, instrument, and profile the original code at a coarse granularity (such as function calls only). Then, at run time, it selectively generates (JIT) code and does more fine-grained profiling and optimization of the dynamically compiled code (such as all loops inside a function). Therefore, O-PIN has less operation overhead, compared to regular PIN [1].

Figure 4.5 shows the operation flow graph for our prototype implementation of the

RDO system. At the start, RDO instruments all function calls in the program, and all loops in the main() function, in order to monitor and identify the frequently executed code regions. (Strongly connected components in the call graph are treated as single nodes.) If a candidate code region is found hot (i.e. the execution count is greater than a *hot threshold*), DVFS testing and decision code will be started to collect run-time information and decide how memory bound the code region is. If the code region is found to be memory bound, RDO will remove the instrumentation code, insert DVFS mode set instructions, and resume the program execution. On the other hand, if a code region is found CPU bound, no DVFS instructions will be inserted. There is still a medium case where the candidate code region may exhibit mixed memory behavior (likely because it contains both memory-bound and CPU-bound sub-regions). For this case, RDO will check if it is a long-running function containing loops. If it is, a copy of this function will be dynamically generated and all loops inside this function will be identified[1] and instrumented. The process will then continue at the loop granularity.

The DVFS selection method in Section 4.3.2 is used to check the memory boundedness of a code region and select a DVFS setting. The required HPC events in equation (4.5), Number of memory bus transactions and Number of μops retired, are among the roughly 100 countable HPC events provided by a Pentium-M processor [35]. (This will be the core of our hardware platform, as to be discussed in the next subsection.) However, the HPC event in equation (4.6) to estimate the time ratio $\frac{N_{concurrent}/f}{total\_time}$ is not available for Pentium-M processors. Instead, we approximate the estimation in (4.6) by a new ratio obtained from available but less-related HPC events. There are several ways to choose the HPC events for this. For our implementation, we used the ratio of Number of μops retired over Number of instructions retired. Based on our empirical experience, we found that the larger this ratio is, the more concurrent CPU instructions there are for a code region. (Note, in Section 4.5,

[1]To identify loops, some linear-time loop analysis techniques such as that in [52] can be used. For our implementation, to reduce the run-time analysis overhead, a simple and fast loop-identification heuristic is used. A likely loop is identified if a conditional branch is going from a higher address to a lower address. Our experience shows this heuristic works quite well for most applications in practice.

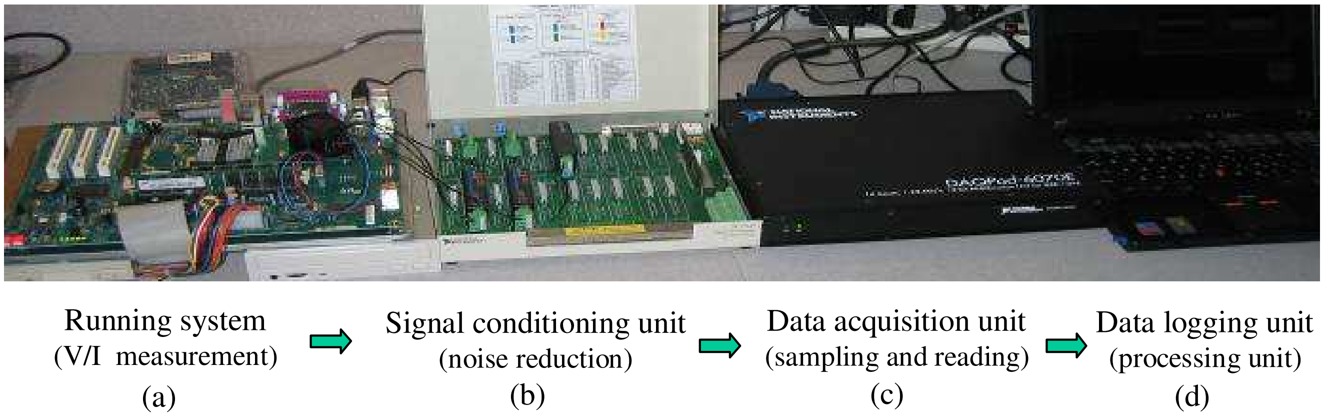

Figure 4.6: Processor power measurement setup. This setup consists of four components.

we will actually show the inverse of this ratio, i.e., the average number of instructions per 1M μops retired.)

Based on the scaling factor β obtained from the DVFS decision algorithm, we choose the actual DVFS setting for a code region (i.e., new_$f = \beta$ old_f). Since most existing processors have only a limited number of DVFS setting points (e.g., six to eight), we pick the setting point close to the desired DVFS setting. (If the desired setting is between two available setting points, we pick the more conservative or higher one. Of course, the more fine-grained DVFS settings available, the better the control effectiveness.)

4.4.2 Deployment in a real system

We have deployed our RDO system in a real running system. The hardware platform we use is an Intel development board with a Pentium-M processor (855GME, FW82801DB), which is shown in Figure 4.6a. The Pentium-M processor we use has a maximum clock frequency of 1.6GHz, two 32K L1 caches, and one unified 1M L2 cache. The board has a 400MHz FSB bus and 512M DDR RAM.

There are 6 DVFS settings or so-called SpeedSteps for Pentium-M (expressed in frequency/voltage pairs): 1.6GHz/1.48v, 1.4GHz/1.42v, 1.2GHz/1.27v, 1.0GHz/1.16v, 800MHz/1.04v, and 600MHz/ 0.96v. The voltage transition rate for DVFS is about $1mv/1\mu s$ (based on our own measurements).

The OS is Linux kernel 2.4.18 (with gcc updated to 3.3.2). We have implemented two loadable kernel modules (LKM) to provide user level support for DVFS control and HPC reading in the form of system calls.

The above system allows accurate power measurements. The overall procedure for power measurements is that we first collect sampling points of CPU voltage and current values. We then compute the power trace and the total energy from these sampling points. Figure 4.6 shows the processor power measurement setup, which includes four components, as detailed below.

Running system voltage/current measurement unit: This unit isolates and measures CPU voltage and current signals. The reason for isolating and measuring the CPU power (instead of power for the whole board) is that we want to have more deterministic and accurate results, not affected by other random factors on the board. Figure 4.7 is a system diagram showing the CPU voltage and current measurement points (marked with $+$ and $-$) on the 855GME development board. As seen in the figure, we use the output sense resistors of the main voltage regulator (precision resistors of $2m\Omega$ each) to measure the current going to the CPU (i.e. measure the voltage drop, then use $I_{CPU} = V_{drop}/R_{sense}$), and use the bulk capacitor to measure the CPU voltage. Note that, as shown in Figure 4.7, if we simply measure the power supply line going to the voltage regulator, the obtained power reading will undesirably include power consumed by components other than the CPU (such as the I/O Hub).

Signal conditioning unit: This unit reduces the measurement noise to get more accurate readings. Measurement noise is inevitable because of the noise sources like the CPU board itself. In particular, since the voltage drop across the sense resistor in Figure 4.7 is on the order of $1mv$ while the noise is on the order of $10mv$ in practice, the noise for our system is 10 times larger than the measured signal. Because noise typically has much higher frequency than the measured signals, we use a two-layer low-pass filter to reduce the measurement noise, which includes a National Instrument (NI) signal conditioning module

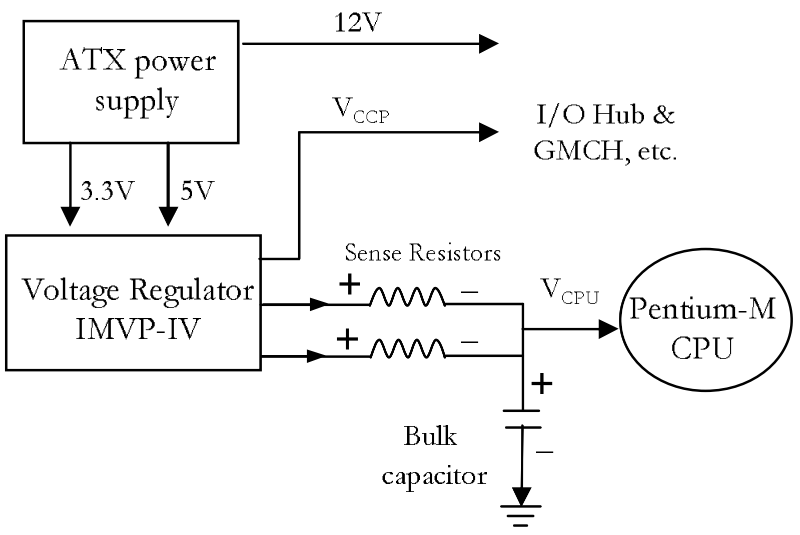

Figure 4.7: A system diagram showing the CPU voltage and current measurement points (marked by $+$ and $-$) on the development board.

AI05 and a simple RC filter as shown in Figure 4.6b. With these filters, we are able to reduce the relative noise error to less than 1%.

Data acquisition (DAQ) unit: This unit samples and reads the voltage and current signals. In order to capture the program behavior variations (especially with DVFS), a fast sampling rate is required. We use the NI data acquisition system DAQPad-6070E [53], which has a maximum sampling rate of 1.2M/s (aggregate), as shown in Figure 4.6c. Since three measurement channels are needed – two for the CPU current and one for the CPU voltage, we set a sampling rate of 200K/s for each channel (so a total of 600K/s is used). This gives a $5\mu s$ sample length for each channel. Given that the minimum voltage transition time is $20 - 100\mu s$ [25], the $5\mu s$ sampling length is adequate.

Data logging and processing unit: This is the host logging machine which processes the sampling data. Every 0.1 seconds, the DAQ unit sends collected data to the host logging machine via a high-speed fire-wire cable. (For each channel, 20K samples are first stored in an internal buffer in the DAQ unit before they are sent out.) The logging machine then processes the received data. We use a regular laptop running NI Labview DAQ software to process the data. We have configured Labview for various tasks: monitoring, raw data recording, and power/energy computation.

4.5 Experimental Results

4.5.1 Experimental setup

For all experiments, we use the software and hardware platforms described in previous sections. Our run-time DVFS optimization system is set to have a performance loss constraint P_{loss} of 5%. (If a larger P_{loss} were used, the resulting frequency settings would be lower, allowing more aggressive energy savings. Conversely, a smaller P_{loss} would lead to larger and more conservative DVFS settings.) For a candidate code region, the *hot threshold* is chosen to be 4 (i.e., a code region is hot if it has executed at least 4 times). But we found our results are not sensitive to this value when it is varied from $3 - 20$. Since the voltage transition time between different SpeedSteps is about $100\mu s - 500\mu s$ for our machine [25], we set the long-running threshold for a code region (as described in our DVFS algorithm in Section 4.3) to be 1.5ms (or 2.4M cycles for a 1.6GHz processor) to make it as least 3X bigger than the voltage transition time. For nested functions, we handle them using a relation graph as described in Section 4.3

For evaluation, we use all SPEC2K FP and SPEC2K INT benchmarks. Since previous static compiler DVFS work in [31] used SPEC95 FP benchmarks, we also include them in our benchmark suites. In addition, we include some Olden benchmarks [15] as they are popular integer benchmarks for studying program memory behavior.[2] For each benchmark, the Intel C++/Fortran compiler V8.1 is used to get the application binary (compiled with -O2). We test each benchmark with the largest ref input set (running to completion). The power and performance results reported here are average results obtained from three separate runs.

To illustrate and give insight for RDO operation, Table 4.2 shows some statistical results obtained from the RDO system for some SPEC benchmarks. In the table, we give total number of hot code regions in the program and total number of DVFS regions iden-

[2] Olden manipulates 7 different kinds of data organizations and structures ranging from linked list to heterogeneous OcTree. We choose the first 7 benchmarks which cover all 7 kinds of data organizations being studied.

Table 4.2: Statistical results obtained for some SPEC benchmarks.
Average number is for per 1M μops; 4L main() means 4th loop in main()

Benchmark	total hot regions	total DVFS regions	region name	total μops	Average L2 cache misses	Average Memory trans	Average Inst	DVFS setting (Hz)
101.tomcatv	63	4	4L main()	23M	3.9K	14.4K	0.99M	1.0G
			11L main()	5M	9.1K	44.7K	0.99M	0.6G
			14L main()	3M	10.0K	49.4K	0.98M	0.6G
			16Lmain()	2M	12.7K	69.1K	0.97M	0.6G
104.hydro2d	184	3	tistep()	11M	4.5K	18.6K	0.91M	1.0G
			advnce()	284M	4.9K	21.5K	0.87M	1.2G
			check()	14M	5.3K	22.9K	0.85M	1.2G
173.applu	72	5	jacld()	208M	12.4K	24.8K	0.99M	0.8G
			blts()	286M	5.9K	11.5K	0.99M	1.2G
			jacu()	156M	12.7K	25.6K	0.99M	0.8G
			buts()	254M	7.0K	12.9K	0.99M	1.2G
			rhs()	188M	4.2K	8.2K	1.0M	1.4G
176.gcc	5673	0	Mem: 0.01K - 1.0K for candidate regions; No DVFS					
181.mcf	34	2	name^1()	3644M	21.0K	83.0K	0.85M	1.0G
			flowcost()	20M	32.0K	112K	0.94M	0.6G
186.crafty	588	0	Mem: 0.00K - 0.01K for candidate regions; No DVF S					
187.facerec	207	2	9L name^2()	22M	3.4K	11.0K	0.95M	1.2G
			16L name^2()	11M	3.4K	10.5K	0.96M	1.2G
254.gap	823	1	name^3()	315M	6.6K	17.0K	0.86M	1.4G

Name 1: primal_net_simplex(); 2: gaborroutines_mp_gabortra to(); 3:collectGaib()

tified. For each DVFS code region, we show the total number of μops retired for the code region (in a single invocation), average L2 cache misses, average number of memory bus transactions, average number of instructions retired (per 1M μops), and the obtained DVFS settings. The DVFS settings are based on the average number of memory bus transactions and the average number of instructions retired. In general, the higher these two numbers are, the lower the DVFS setting. Taking the benchmark 104.hydro2d as an example, we see both numbers contributed to the final DVFS settings. The quantitative relationship between those numbers and the DVFS setting is based on the formulas in Section 4.3. Since there are only 6 available frequency/voltage settings for our system, the obtained β needs to be rounded up to an available frequency point. Overall, we see the number of DVFS opportunities identified by RDO ranges from large (e.g. as low as 0.6Ghz for 101.tomcatv) to small (e.g. no DVFS for 176.gcc).

In order to look more closely at RDO operation, we next examine in detail one partic-

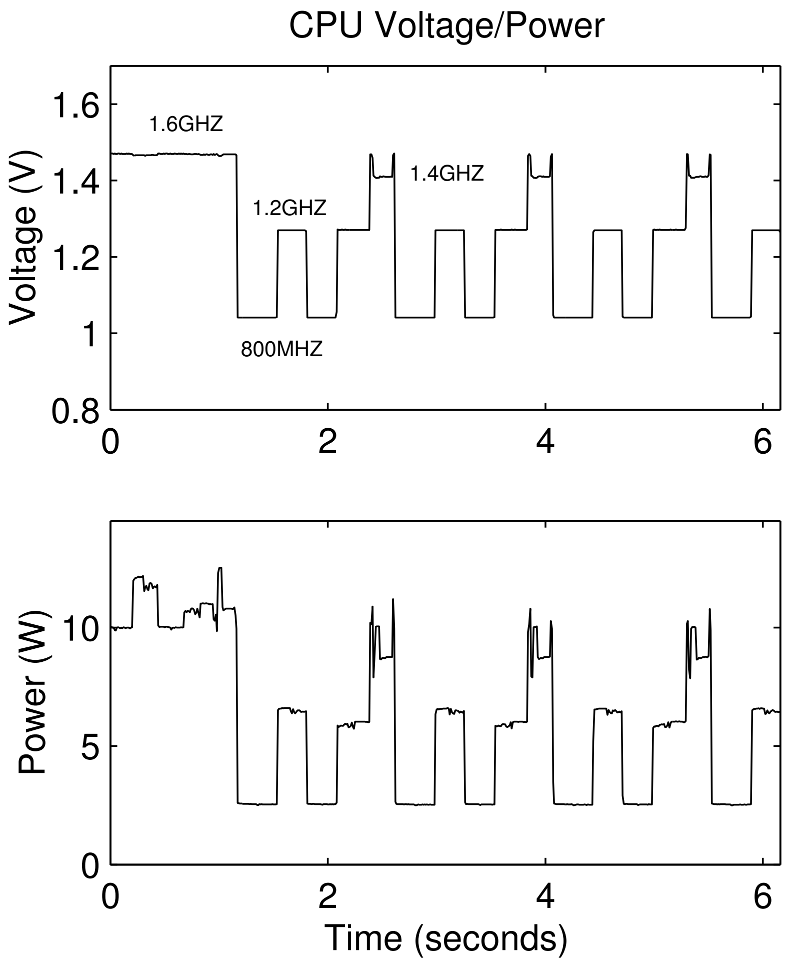

Figure 4.8: A partial trace of the CPU voltage and power for SPEC benchmark 173.applu running with the RDO

ular benchmark: 173.applu. Recall in Section 4.2 we observed recurring memory phase behavior for 173.applu. Analysis by RDO further reveals that those phase changes are mainly caused by invocations of the 5 different functions shown in Table 4.2. The 5 functions have different memory behavior in terms of average number of L2 cache misses and average number of memory bus transactions. This observation is consistent with the behavior shown in Figure 4.1. By inserting DVFS instruction directly in the code regions, RDO adjusts the CPU voltage and frequency to adapt to the program phase changes (with frequency settings of 0.8GHz for two regions, 1.2GHz for two other, and 1.4GHz for the last region). Figure 4.8 shows a part of the CPU voltage and power trace for 173.applu running with RDO. If we compare this figure with the figures in Section 4.2, it is interesting to see the CPU voltage/frequency are being adjusted to adapt to the recurring phase changes shown in Figure 4.1 (with lower clock frequencies corresponding to higher L2 cache misses). The power trace is also interesting. Initially it fluctuates around the value of $11W$ (due to different system switching activities). After the program execution enters into the DVFS code regions, the power drops dramatically to a level as low as $2.5W$. As will be shown by the experimental results in Section 4.5.2, the DVFS optimization applied to the code regions in 173.applu has led to considerable energy savings ($\sim$35%) with little performance loss ($\sim$5%).

4.5.2 Energy and performance results

We view the run-time DVFS optimizer (RDO) as an addition to the regular dynamic (performance) optimization system as shown in Figure 4.3. So, to isolate the contribution of the DVFS optimization, we will first report the energy and performance results relative to the O-PIN system *without* DVFS (i.e., we do not want to mix the effect of our DVFS optimization and that of the underlying dynamic compilation and optimization system, which is being developed heavily by researchers at Intel and U. of Colorado [1]). In addition, as a comparison, we will also report the energy results from a static voltage scaling, which sim-

Table 4.3: Average results for each benchmark suite: *RDO* versus *Static*Scale.

Benchmark Suite	Performance degradation		Energy savings		Energy-Delay product improvement	
	RDO	*Static*	*RDO*	*Static*	*RDO*	*Static*
SPEC95 FP	2.1%	7.9%	24.1%	13.0%	22.4%	5.6%
SPEC2K FP	3.3%	7.0%	24.0%	13.5%	21.5%	6.8%
SPEC2K INT	0.7%	11.6%	6.5%	11.5%	6.0%	-0.3%
Olden	3.7%	7.8%	25.3%	13.7%	22.7%	6.3%

ply scales the supply voltage and frequency statically for all benchmarks to get roughly the same amount of average performance loss as those in our results. (We chose $f = 1.4Ghz$ for static voltage scaling, which is the only voltage setting point in our system to get an average performance loss around 5%.)

Figures 4.9 and 4.10 show the performance loss, energy savings, and energy delay product (EDP) improvement results for all SPEC95 FP, SPEC2K FP/INT, and Olden benchmarks. Note that these results have taken into account all DVFS optimization overhead, such as the time cost to check memory boundedness of a code region. For convenience, we refer to the result from our runtime DVFS optimizer as *RDO*, and refer to results by static voltage scaling as *StaticScale*. There are several interesting observations.

First, in terms of EDP improvement, *RDO* outperforms *StaticScale* by a big margin for nearly all benchmarks. This shows the efficiency of our design with fast and effective DVFS decisions.

Second, the energy and performance results for individual benchmarks in each benchmark suite vary significantly. On the high end, we have achieved up to 64% energy savings (4.9% performance loss) for SPEC95 FP (101.tomcatv), up to 70% energy savings (0.5% performance loss) for SPEC2K FP (171.swim), up to 44% energy savings (with 5% performance loss) for SPEC2K INT (181.mcf), and up to 61% energy savings (4.5% performance loss) for Olden benchmarks (Health). On the low end, we see close-to-zero (or even slightly negative) EDP improvement for some benchmarks in each benchmark suite. To understand

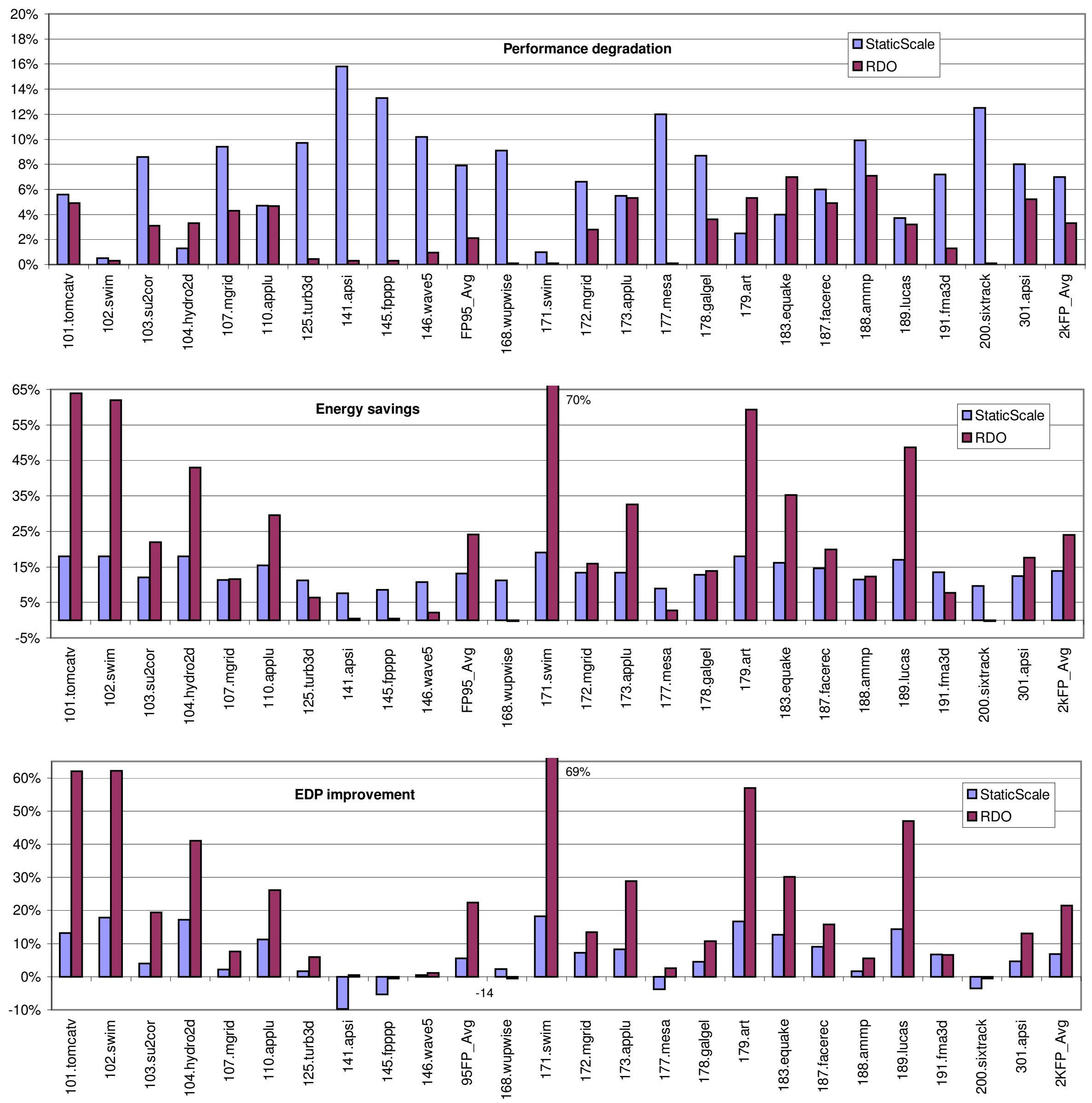

Figure 4.9: Performance degradation, energy savings, and energy-delay product (EDP) improvement for SPEC95 FP benchmarks (on the left) and SPEC2K FP benchmarks (on the right). We show results for both our runtime DVFS optimizer (*RDO*) and the *StaticScale* voltage scaling.

the reasons, we see that the efficiency of a DVFS control is largely constrained by the memory boundedness of an application. The more memory bound an application is, the more opportunities and energy saving potentials there are for DVFS. Relative to our experimental system in Figure 4.6 (with a large 1M L2 cache), these benchmarks show a variety of memory boundedness which leads to a variety of the EDP results. Overall, the distribution

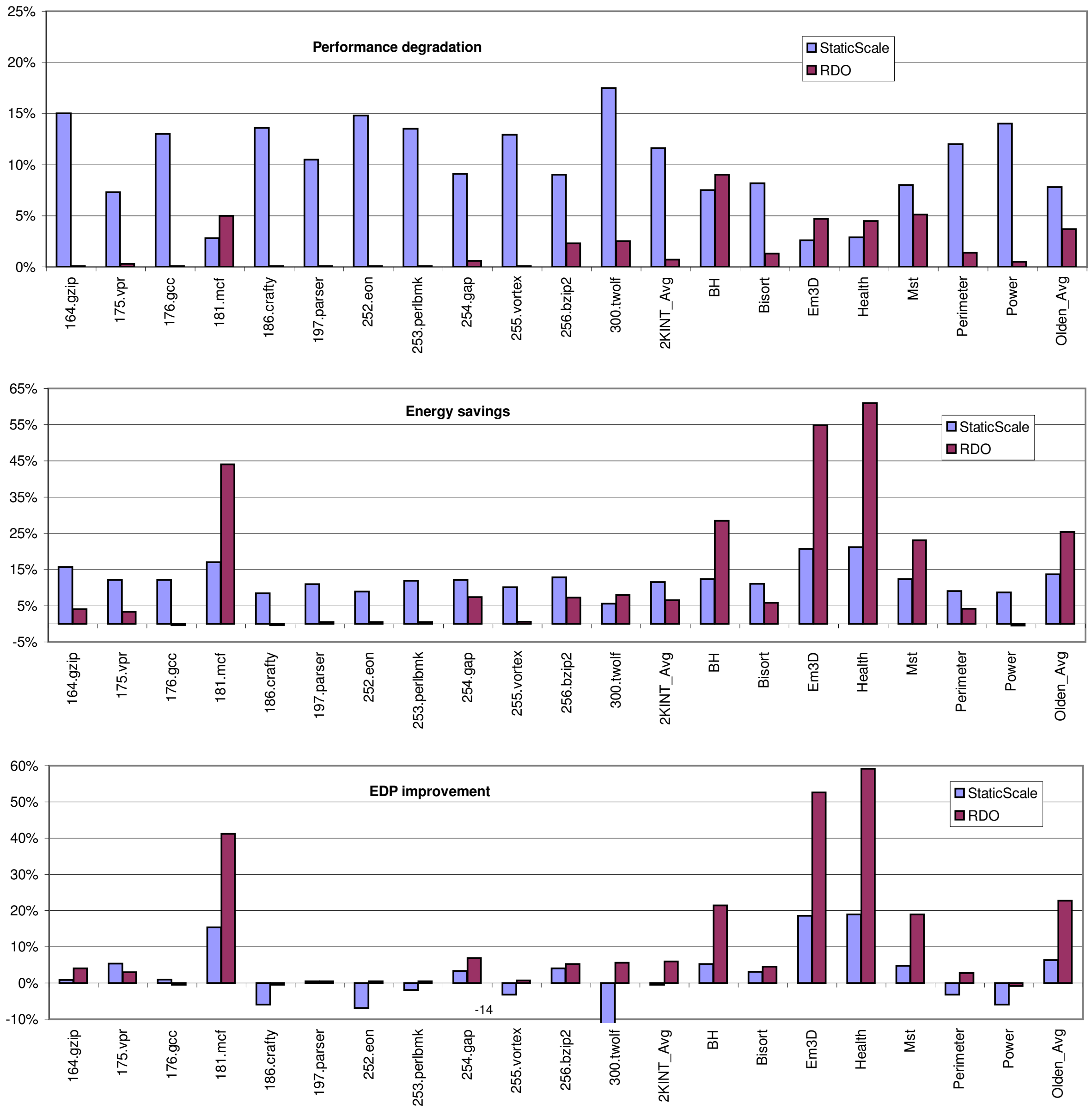

Figure 4.10: Performance degradation, energy savings, and energy-delay product (EDP) improvement for SPEC2K INT benchmarks (on the left) and Olden benchmarks (on the right). We show results for both our runtime DVFS optimizer (*RDO*) and the *StaticScale* voltage scaling.

of the SPEC2K-INT EDP results is concentrated and close to the low end, while the overall distribution of the Olden and SPEC-FP EDP results is very spread out between the high end and the low end.

The average results for each benchmark suite are summarized in Table 4.3. We show both the results from our techniques and the *StaticScale* results. On average, we have achieved an EDP improvement of 22.4% for SPEC95 FP, 21.5% for SPEC2K FP, 6.0% for SPEC2K INT, and 22.7% for Olden benchmarks. These represent $3-5$ fold better results as compared to the *StaticScale* EDP improvement: 5.6% for SPEC95 FP, 6.8% for SPEC2K FP, -0.3% for SPEC2K INT, and 6.3% for Olden benchmarks. (The average SPEC2K INT EDP result is relatively lower compared to the other three benchmark suites. This is because SPEC2K INT benchmarks are dominantly CPU bound as shown by previous studies [31]. There is nothing intrinsic about floating point versus integer data. It is just about the amount of memory traffic.)

We also want to have a rough comparison with the static compiler DVFS results in [31] based on the reported energy performance numbers in that paper. (We were not able to re-implement their optimizer and replicate the experiments in [31].) Compared to the reported results for SPEC95 FP benchmarks in [31] (on average: 2.1% performance loss, 11.0% energy savings, 9.0% EDP improvement), we have achieved on average twice as much energy savings for the same amount of performance loss. Apart from the dynamic versus static benefits described in Section 4.2, there are two other key factors contributing to the different results. First, the static compiler DVFS algorithm in [31] picks only a single DVFS code region for a program (to avoid the excessive offline analysis time), while our online DVFS design can identify multiple DVFS code regions in a program as long as they are beneficial, as illustrated by examples in Table 4.2. Second, the decision algorithm in [31] is based on (offline) timing profiling for each code region, while our algorithm is more microarchitecture oriented and directly uses information about run-time environment, such as hardware performance counts.

Overall, the results in Figures 4.9 and 4.10 and Table 4.3 show the proposed technique is promising in addressing the energy and performance control problem in microprocessors. We attribute the promising results to the efficiency of our design and to the advantages of the dynamic compiler driven approach.

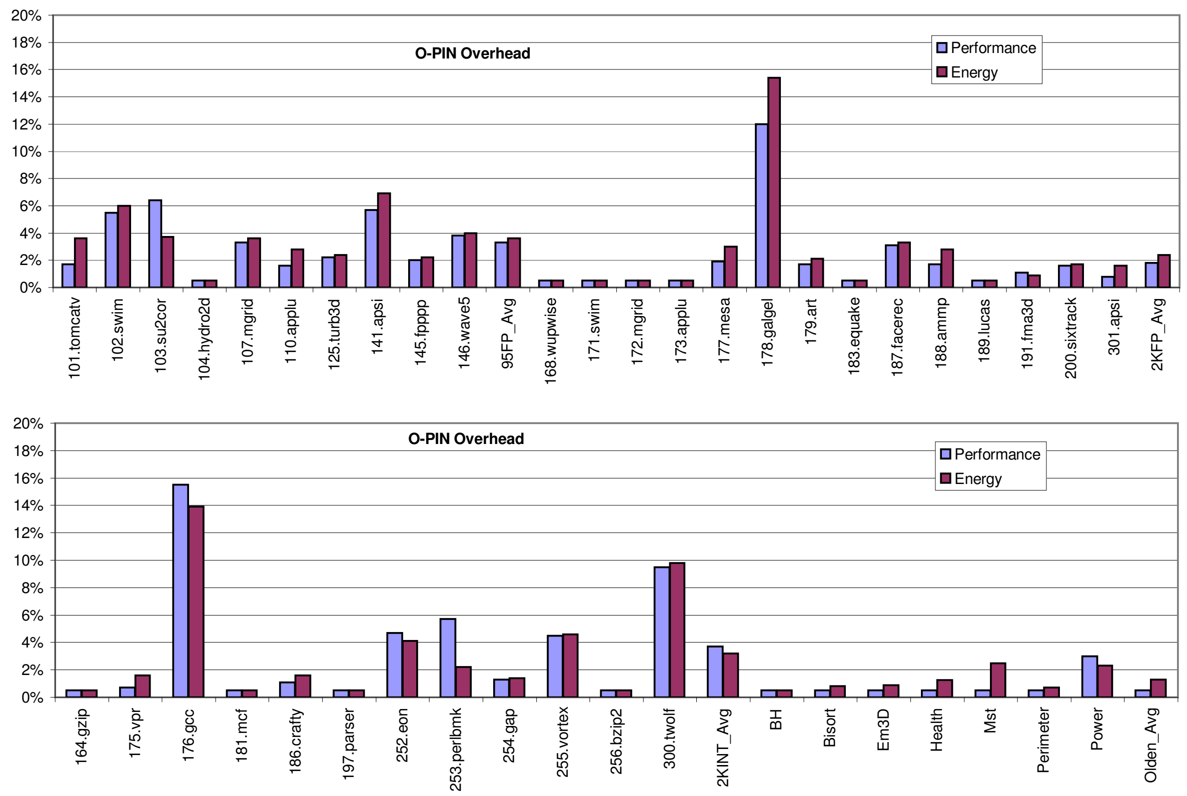

Figure 4.11: Performance and Energy overhead for the basic O-PIN infrastructure without applying optimizations.

4.5.3 Basic O-PIN overhead

A dynamic optimization system has basic setup/operation overhead (i.e., the time spent to do basic setup, to monitor/identify frequently executed code regions, etc). This overhead must be offset or amortized by the subsequent performance optimization gain before we can see any net performance improvement. It has been shown that a dynamic optimizer with aggressive performance optimizations will have significant performance gains for various benchmarks [5, 11].

The O-PIN system we use is a dynamic-optimization infrastructure and does not include implemented performance optimizations. (Users can use this infrastructure to implement their own performance optimizations like loop unrolling and data prefetching, but this is beyond the scope of this paper.) So compared to the native application, the basic O-PIN system has a negative performance gain. In other words, there is a performance and energy overhead associated with the basic O-PIN infrastructure. Next, to give a complete picture of this work, we will also show results for O-PIN and DVFS relative to the native.

Figure 4.11 shows the performance and energy overhead for the basic O-PIN infrastructure (computed relative to the native). For individual benchmarks, the performance overhead is as low as 0.5% for benchmarks like 164.gzip or 171.swim, and is as high as 15% for benchmarks like 176.gcc. On average, the performance overhead for O-PIN is about 3.3% for SPEC95 FP, 1.8% for SPEC2K FP, 3.7% for SPEC2K INT, and 0.5% for Olden benchmarks. The energy overhead values are similar. These values are significantly lower than the basic overhead for a regular PIN system [47], because of the low-overhead implementation in O-PIN as described in Section 4.4.

If we look at the DVFS results from our scheme when computed with the inherited infrastructure overhead, the EDP numbers will be lower than those in the last subsection, as we expected. On average, the EDP improvement with the inherited overhead is about 16.7% for SPEC95 FP, 17.9% for SPEC2K FP, -1.4% for SPEC2K INT, and 20.9% for Olden benchmarks. In general, these are clearly still promising results.

4.5.4 Discussion and micro-architectural suggestions

For experimental results in this section, it would be desirable to have some potential or upper-bound DVFS numbers, and show how far our results are from these upper-bound numbers. However, it is still an open research question how to effectively and accurately compute the upper-bound DVFS results. One possible way to estimate the upper bound is to extend the mathematical formulation in [67] from optimizing multiple sequential scaling points to optimizing multiple DVFS code regions. We leave this for possible future exploration.

While the experimental results in this section are promising, they could be further improved if more micro-architectural support were available. One possible support could be logic to identify and predict CPU execution slack such as that proposed in [23]. This would make the DVFS computation easier and more accurate. Another possible support could be some power-aware hardware monitoring counters and events to monitor the power consumption in a processor unit and the voltage variations. In addition, more fine-grained DVFS settings could make the intra-task DVFS design more effective. Our experience shows that, for many code regions in the benchmarks, RDO was forced to select an unnecessarily high voltage/frequency setting due to a lack of enough intermediate settings between the current six SpeedSteps in Pentium-M processors.

4.6 Related Work

As we mentioned in the introduction, nearly all existing intra-task DVFS schemes are based on hardware [49, 58], OS time-interrupt [19, 64], or static compiler [31, 68] techniques. Very little has been done in the direction of dynamic compiler driven DVFS.

One piece of related work along that direction is the Java virtual machine DVFS presented in [27]. Their work is similar to ours in the sense that both use run-time software to decide DVFS settings for the application. However, their work differs from ours in the following aspect. First, they use the Java virtual machine to target Java applications at the granularity of Java method, while we use a general dynamic optimization system to target general applications at a more fine-grained granularity including code regions like loops. Second, their DVFS algorithm does not take into account the memory boundedness of a code region (they assume the execution time of a code region always scales linearly with the frequency, no matter how memory bound it is). Also, their DVFS algorithm assumes some sort of time budget (so called *projected time*), and compares the current execution time with the time-budget to decide how much to scale. This treatment might be suitable for multimedia applications which have a pre-determined time-budget for each frame, but might not be as suitable to general applications. In contrast, our DVFS algorithm considers the memory boundedness of a code region, and works well for general applications.

Third, the power evaluation in [27] is based on simulation, while our evaluation is based on live-system physical power measurements.

4.7 Future Work

There are several possible avenues for future work. The focus of this work is on the new concept of dynamic compiler driven DVFS and the overall design framework. A direct follow-up work would look at specific design issues and techniques in more depth, such as code transformation and periodic re-optimization for DVFS. Also deeper analysis could be done for the experimental results, such as a breakdown of the results/benefits by regions or by different contributing factors. However, since we use a real system as opposed to simulation, it will be challenging to break down the results/benefits in an effective way.

Another possible future work is to implement some conventional performance optimizations (like loop unrolling and data prefetching), and study the interactions between energy optimizations and performance optimizations in a run-time system. In addition, some new processors allow DVFS for the memory bus as well. A possible future direction is to generalize the analytical decision model and the DVFS algorithm in this paper for the case where both CPU and memory can have DVFS.

4.8 Summary

This Chapter has given reasoning and statistical results to highlight the unique features and advantages of dynamic compiler driven DVFS over existing techniques. We have presented a design framework of the run-time DVFS optimizer in a general dynamic compilation system. We have described the methodology and reported our experiences in implementing and deploying a real DVFS optimization system. Experimental results based on physical measurements show that our technique leads to significant energy delay product (EDP) improvement in a real computer system.

Overall, we feel the dynamic compiler driven control technique is a good complement to existing techniques, and offers great promise for active power and performance management in modern processors.

Chapter 5

Conclusions and Future Directions

5.1 Concluding Remarks

The work in this thesis studies both architectural level and compiler level techniques for controlling power and performance in microprocessors. The overall contributions of this work are the proposed new concepts, methods, and framework for intelligent power and performance management.

This research has shown that formal control techniques for architectural level power management are applicable and effective on modern processors. It has investigated both fixed-interval and adaptive-interval control schemes, which are both hardware-efficient. This research has also shown dynamic compiler driven energy control can lead to significant energy efficiency in real computer systems. Because of its orthogonal features and advantages, dynamic compiler driven techniques can be an effective complement to existing hardware-based techniques, and allow us to work toward a multi-layer (hardware and software) collaborative control scheme.

While this work focuses on power and performance management only, the methodology and design framework described in this thesis are more broadly applicable, and can be generalized for other emerging issues such as thermal control, DI/dt, and system reliability.

5.2 Future Directions

There are several future directions, which are directly or indirectly related to this work.

One direction is to look at multi-modal software/hardware techniques for power-efficient computing. Nearly all existing work in this area uses a single-loop local control mechanism, which looks at one control device and one metric at a time and ignores interactions among different control loops. A multi-loop global control mechanism is needed to achieve the best control effectiveness. The formal control techniques in this thesis can be applied to study the interacting effects of multiple control loops. Also, Chip multi-processors (CMPs) constitute a new complexity-efficient and power-efficient architecture. Many CMP design and control issues can be studied, especially those issues that link architecture, circuits and clocks. For example, there are questions about how to design the interface circuit between different cores, what the best energy control strategy is for each clocking style, and how to use an analytical or hybrid (analytical-simulation) approach to evaluate thread level parallelism potential in applications.

Another direction is to look at the analysis and design techniques for variation-tolerant architecture. As technology scales, process variation will increase dramatically for future processors and lead to critical variability in terms of path delay, leakage current, threshold voltage, etc. In addition, operating variations of supply voltage and temperature will also cause increased variability at run-time. There are two main issues with the increasing parameter variations. First, there will be a widening gap between the target performance (or yield) and the actual performance (or yield). Second, extra guard bands are needed to critical path timing to account for variability, which may lead to lower target performance and sub-optimal operations.

One important question for computer architects and system software writers is how to design a variation-tolerant system that maximizes performance, power efficiency, and yield even with the increasing variability. This is an open and challenging research problem. Many ideas can be explored. One idea is to use multiple voltage levels and multiple clock islands to accommodate process variations, especially within-die variations. Another idea is to design the processor not constrained by the worst case delay, but by the nominal or expected delay of critical paths. Some kind of self-adjusting mechanism might then be used to test and detect the actual delay (or even errors) at run-time, and adjust design parameters like supply voltage or body bias voltage to ensure correctness, performance, and power efficiency. This is essentially a feedback control problem. Some formal control-theoretic techniques can be used here for an effective and robust design. Besides the design techniques, new modeling and analysis techniques may be needed. For example, probabilistic techniques may be necessary to account for variability, as opposed to conventional deterministic techniques.

Overall, I feel the techniques and schemes outlined in this thesis have great applicability potential in addressing many of the upcoming challenges in the design of future computing systems.

Bibliography

[1] PIN manuals and APIs. In *Website http://rogue.colorado.edu/Pin/index.html*, August 2005.

[2] D.H. Albonesi. Selective cache ways: On-demand cache resource allocation. In *Proceedings of MICRO*, Dec. 1999.

[3] C. Anderson. Tuning and optimization of a 170m transistor microprocessor. In *Proceedings of the IEEE/ACM International Workshop on Timing Issue in the Specification and Synthesis of Digital System (TAU2000)*, Dec 2000.

[4] K.J. Astrom and B. Wittenmark. *Adaptive Control*. Addison-Wesley, 1995.

[5] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of PLDI*, June 2000.

[6] L. Baraz, T. Devor, O. Etzion, S. Gondenberg, A. Skaletsky, Y. Wang, and Y. Zemach. IA-32 execution layer: a two-phase dynamic translator designed to support IA-32 applications on itanium-based systems. In *Proc. of the 36th Micro*, Dec 2003.

[7] L.A. Barroso. The price of performance. *ACM Queue*, 3(7), September 2005.

[8] S. K. Bose. *An Introduction to Queueing Systems*. Kluwer Academic, 2002.

[9] D. Brooks and M. Martonosi. Dynamic thermal management for high performance processors. In *Proceedings of HPCA*, Feb 2001.

[10] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimization. In *Proc. of the ISCA-27*, June 2000.

[11] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of CGO'03*, March 2003.

[12] T. Burd and R. Brodersen. Design issues for dynamic voltage scaling. In *Proceedings of ISLPED*, August 2000.

[13] D. Burger and T. M. Austin. The SimpleScalar tool set version 2.0. Technical Report 97-1342, Department of Computer Science, University of Wisconsin-Madison, June 1997.

[14] A. Buyuktosunoglu, T. Karkhanis, D.H. Albonesi, and P. Bose. Energy efficient co-adaptive instruction fetch and issue. In *Proceedings of ISCA*, pages 147–156, June 2003.

[15] M.C. Carlisle, A. Rogers, J. H. Reppy, and L. J. Hendren. Early experiences with Olden. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, August 1993.

[16] A.P. Chandrakasan, S. Sheng, and R.W. Brodersen. Low-power cmos digital design. *IEEE Journal of Solid-State Circuits*, 27(4):473–484, April 1992.

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

[18] T. Chelcea and S. M. Nowick. Robust interfaces for mixed-timing systems with application to latency-insensitive protocols. In *Proc. of DAC-2001*, pages 21–26, 2001.

[19] K. Choi, R. Soma, and M. Pedram. Fine-grained dynamic voltage and frequency scaling for precise energy and performance trade-off based on the ratio of off-chip access to on-chip computation times. In *Proceedings of DATE*, Feb 2004.

[20] L.T. Clark. Circuit design of XScale microprocessors. In *Proceedings of the 2001 Symposium on VLSI Circuits*, June 2001.

[21] K. Ebcioglu and E. R. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *Proceedings of ISCA*, June 1997.

[22] M. Taylor et al. The RAW processor - a scalable 32-bit fabric for embedded and general purpose computing. In *Proceedings of Hot Chips XIII*, August 2001.

[23] B. Fields, R. Bodik, and M. D. Hill. Slack: Maximizing performance under technological constraints. In *Proceedings of ISCA*, May 2002.

[24] S. Ghiasi, J. Casmira, and D. Grunwald. Using IPC variation in workloads with externally specified rates to reduce power consumption. In *Workshop on Complexity Effective Design, Vancouver, Canada, June 2000.*, June 2000.

[25] S. Gochman, R. Ronen, et al. The Intel Pentium M processor: Microarchitecture and performance. *Intel Technology Journal*, 07(2), 2003.

[26] S. Gurumurthi, A. Sivasubramaniam, M. Kandemir, and H. Franke. DRPM: Dynamic speed control for power management in server class disks. In *Proceedings of ISCA*, June 2003.

[27] V. Haldar, C. Probst, V. Venkatachalam, and M. Franz. Virtual machine driven dynamic voltage scaling. Technical Report CS-03-21, University of California, Irvine, CA, Oct 2003.

[28] T. Heath, E. Pinheiro, J. Hom, U. Kremer, and R. Bianchini. Code transformations for energy-efficient device management. *IEEE Transactions on Computers*, 53(8), August 2004.

[29] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 3rd Edition, 2003.

[30] R.V. Hogg and A.T. Craig. *Introduction to Mathematical Statistics, Fifth edition*. Prentice Hall, 1995.

[31] C-H Hsu and U. Kremer. The design, implementation, and evaluation of a compiler algorithm for CPU energy reduction. In *Proc. of PLDI-2003*, pages 38–48, June 2003.

[32] S. Hu, M. Valluri, and L. John. Effective adaptive computing environment management via dynamic optimization. In *Proceedings of CGO'05*, March 2005.

[33] M.C. Huang, J. Renau, and J. Torrellas. Positional adaptation of processors: Application to energy reduction. In *Proceedings of ISCA*, June 2003.

[34] C.J. Hughes, J. Srinivasan, and S. V. Adve. Saving energy with architectural and frequency adaptations for multimedia applications. In *Proceedings of Micro-34*, Dec. 2001.

[35] Intel Corporation, Santa Clara, CA. *Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide*, 2005.

[36] A. Iyer and D. Marculescu. Power efficiency of multiple clock multiple voltage cores. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2002.

[37] A. Iyer and D. Marculescu. Power-performance evaluation of globally asynchronous, locally synchronous processors. In *Proc. of the 26th ISCA*, May 2002.

[38] R. Joseph, D. Brooks, and M. Martonosi. Control techniques to eliminate voltage emergencies in high performance processors. In *Proceedings of HPCA*, Feb 2003.

[39] I. Kadayif, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and A Sivasubramaniam. EAC: a compiler framework for high-level energy estimation and optimization. In *Proceedings of DATE*, March 2002.

[40] A. Kansal and M.B. Srivastava. An environmental energy harvesting framework for sensor networks. In *Proceedings of ISLPED*, August 2003.

[41] B.C. Kuo. *Automatic Control Systems. , 7th edition*. Prentice Hall, 1995.

[42] A.R. Lebeck, X. Fan, H. Zeng, and C.S. Ellis. Power aware page allocation. In *Proceedings of ASPLOS-IX*, pages 105–116, Nov. 2000.

[43] X. Li, Z. Li, F. M. David, P. Zhou, Y.Y. Zhou, S.V. Adve, and S. Kumar. Performance directed energy management for main memory and disks. In *Proceedings of ASPLOS-XI*, Oct. 2004.

[44] D.V. Lindley. The theory of queues with a single server. In *Proceedings of the Cambridge Philosophical Society*, pages 277–289, 1952.

[45] J. R. Lorch and A. J. Smith. Improving dynamic voltage scaling algorithm with PACE. In *Proceedings of the SIGMETRICS-2001*, pages 50–61, June 2001.

[46] Z. Lu, J. Hein, M. Stan, J. Lach, and K. Skadron. Control-theoretic dynamic frequency and voltage scaling. In *Proc. of the Intl. Conference on Compiler, Architecture, and Synchesis for Embedded Systems (CASES)*, October 2002.

[47] C-K Luk, R. Cohn, R. Muth, R. Muth, H. Patil, A. Kaluser, G. Lowney, S. Wallace, V.J. Reddi, and K. Hazelwood. PIN: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of PLDI'05*, June 2005.

[48] G. Magklis, M.L. Scott, G. Semeraro, D.H. Albonesi, and S. Dropsho. Profile-based dynamic voltage and frequency scaling for a multiple clock domain microprocessor. In *Proc. of the 30th ISCA*, June 2003.

[49] D. Marculescu. On the use of microarchitecture-driven dynamic voltage scaling. In *In Workshop on Complexity Effective Design, Vancouver, Canada, June 2000.*, June 2000.

[50] D. Marculescu, D.H. Albonesi, A. Buyuktosunoglu, and P. Bose. Partially asynchronous microprocessors (PAMs). In *ISCA 2003 Tutorial*, June 2003.

[51] D. Matzke. Will physical scalability sabotage performance gains? *IEEE Computer*, pages 37–39, Sep 1997.

[52] Steven Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufman Publishers, 1997.

[53] National Instruments. *Data Acquisition (DAQ) Hardware, http://www.ni.com/dataacquisition*, 2005.

[54] K. Olukotun and L. Hammond. The future of microprocessors. *ACM Queue*, 3(7):27–34, September 2005.

[55] D.B. Percival and A.T. Walden. *Spectral Analysis for Physical Applications*. Cambridge University Press, 1993.

[56] E. Perelman, G. Hamerly, and B. Calder. Picking statistically valid and early simulation points. In *Proc. of the PACT-2003*, September 2003.

[57] G. Semeraro. Multiple clock domain microarchitecture design and analysis. In *Ph.D Thesis, University of Rochester*, August 2003.

[58] G. Semeraro, D.H. Albonesi, S.G. Dropsho, G. Magklis, S. Dwarkadas, and M.L. Scott. Dynamic frequency and voltage control for a multiple clock domain microarchitecture. In *Proc. of the 35th Micro*, pages 356–367, November 2002.

[59] G. Semeraro, G. Magklis, R. Balasubramonian, D.H. Albonesi, S. Dwarkadas, and M.L. Scott. Energy efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. In *Proc. of the 8th HPCA*, pages 29–40, February 2002.

[60] G. Semeraro, G. Magklis, and Y. Zhu. Personal communications. December 2003.

[61] A.E. Sjogren and C.J. Myers. Interfacing synchronous and asynchronous modules within a high-speeed pipeline. In *Proceedings of the 17th International Conference on Advanced Research in VLSI*, pages 47–61, Sept 1997.

[62] K. Skadron, T. Abdelzaher, and M. Stan. Control-theoretic techniques and thermal-RC modeling for accurate and localized dynamic thermal management. In *Proc. of the 8th HPCA*, February 2002.

[63] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *Proceedings of the 1st USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, Nov. 1994.

[64] A. Weissel and F. Bellosa. Process cruise control: Event-driven clock scaling for dynamic power management. In *Proceedings of CASE'02*, Oct 2002.

[65] Q. Wu, P. Juang, M. Martonosi, and D.W. Clark. Formal online methods for voltage/frequency control in multiple clock domain microprocessors. In *Proceedings of the 11th ASPLOS*, October 2004.

[66] Y. Wu, M. Breternitz, J.Quek, O. Etzion, and J. Fang. The accuracy of initial prediction in two-phase dynamic binary translators. In *Proceedings of CGO'04*, March 2004.

[67] F. Xie, M. Martonosi, and S. Malik. Bounds on power savings using runtime dynamic voltage/frequency scaling: An exact algorithm and a linear-time heuristic approximation. In *Proc. of ISLPED*, August 2005.

[68] Fen Xie, Margaret Martonosi, and Sharad Malik. Compile-time dynamic voltage scaling settings: Opportunities and limits. In *Proc. of 2003 PLDI*, June 2003.

[69] K.Y. Yun and A. E. Dooply. Pausible clocking based heterogeneous systems. *IEEE Transactions on VLSI Systems*, 7(4):482–487, December 1999.