SOFTWARE APPROACHES FOR ENERGY EFFICIENT SYSTEM DESIGN

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Eui-Young Chung
June 2002

© Copyright by Eui-Young Chung 2002
All Rights Reserved

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Giovanni De Micheli
(Principal Adviser)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Oyekunle Olukotun

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Luca Benini

Approved for the University Committee on Graduate Studies:

Abstract

Energy efficiency of electronic devices has become a critical issue in modern system design due to three major reasons. First, energy saving increases the operational time of battery-powered portable systems by extending the battery lifetime. Second, heat dissipation can be reduced by energy reduction. Excessive heat dissipation increases cooling and packaging cost. Also, it disturbs continuous performance improvement and system reliability. Third, energy reduction alleviates the demand for electricity as well as the impact on our environment.

As design complexity increases, processor-based embedded system design has become widely used to meet the time-to-market constraint and increase design flexibility. Typically, processor-based systems can be structured into three layers - hardware, operating system (OS), and application program layers. Even though two software layers do not consume the power directly, they control the behavior of the hardware and have strong impact on the energy consumption of the hardware layer. However, most previous research has focused on the energy opimization of the hardware itself.

In this thesis, I will discuss how to restructure two software layers for the improvement of overall system energy consumption. As an OS-level technique, I will present *dynamic power management* (DPM) technique which allows OS to change the power state of hardware components. Also, I will describe an application-program-level technique which restructures the software so that hardware consumes less power.

DPM is a design methodology for reducing power consumption of electronic systems by performing selective shutdown of idle system resources. The effectiveness of a power management algorithm depends critically on accurate modeling and prediction of user behavior and the computation of the control policy. In this thesis, I present

two different on-line adaptive DPM algorithms. The first approach models systems as finite-state Markov chains and applies on-line adaptation technique to deal with initially unknown or non-stationary user behavior, which are very common in real-life systems. The proposed approach extends policy optimization techniques in a stationary stochastic environment to handle the non-stationary stochastic environment for practical applications using *sliding windows*. The effectiveness of the proposed approaches is demonstrated by a complete DPM implementation on a laptop computer with a power-manageable hard disk that outperforms existing DPM schemes.

The second approach is a heuristic event-driven DPM algorithm based on an *adaptive learning tree* which is the representation of recent past user history. This approach controls the power state of the system based on the future user behavior predicted using the learning tree.

Next, I discuss a framework and a tool for low energy software optimization technique including the algorithms and a tool flow to reduce the computational effort of programs, using value profiling and partial evaluation. Such a reduction translates into both energy savings and average case performance improvement, with tolerable increase of worst case performance and code size. The tool reduces the computational effort by specializing frequently executed procedures for the most common values of their parameters. The most effective specializations are automatically searched and identified, and the code is transformed through partial evaluation. Experimental results show that the proposed technique greatly improves both energy consumption and performance of the source code. Also, the proposed automatic search engine greatly reduces code optimization time with respect to exhaustive search.

Acknowledgments

This thesis would not be possible without the inspiration and patient guidance from my advisor, Professor Giovanni De Micheli. I am very grateful to my reading committee, Professor Oyekunle Olukotun and Professor Luca Benini, for their encouragement and suggestions during my Ph.D. study.

The support from Stanford CAD group is also invaluable. I learned a lot from my officemate, Yung-Hsiang Lu, he gave immediate feedback on my ideas. I also want to thank current and past members in our group: Armita Peymandoust, Terry Ye, Chaiyasit Manovit, Davide Bertozzi, Tajana Šimunić, Luc Semeria, Jim Smith, Vincent Mooney and Alessandro Bogliolo. The administrative staff made Computer Systems Laboratory an enjoyable working environment. Credits belong to Evelyn Ubhoff, Kathleen DiTommaso.

Last but not least, I am deeply in debt to my family, especially my wife Sung-Eun Kim. I want to thank my friends in Korea and USA; they have made my life full of fun.

This research project was sponsored by MARCO/DARPA Gigascale Silicon Research Center, NSF under contract CCR-9901190 and ST Microelectronics. Their financial support made this work possible.

Contents

Abstract **iv**

Acknowledgments **vi**

1 Introduction **1**

1.1 Motivation 1
1.2 Energy-Centric System Abstraction 3
1.3 Research Objectives 4
1.4 Dynamic Power Management 5
1.4.1 Overview 5
1.4.2 Related Work 8
1.5 Low Energy Software Optimization 13
1.5.1 Overview 13
1.5.2 Related Work 16
1.6 Thesis Contributions 19
1.6.1 DPM for Non-Stationary Service Requests 19
1.6.2 Low Energy Software Optimization Framework 20
1.6.3 Limitations and Future Work 21
1.7 Thesis Organization 22

2 Sliding Window Technique for DPM **23**

2.1 DPM in Known Stationary Environment 24
2.2 DPM in Unknown Stationary Environment 26
2.2.1 Estimation of Stationary SR 28

2.2.2 Decision Policy 29
2.3 DPM in Non-Stationary Environment 33
2.3.1 Non-Stationary Service Requestor 33
2.3.2 Single Window Approach 35
2.3.3 Multi-Window Approach 38
2.4 Experimental Results 40
2.4.1 Experimental Setting 40
2.4.2 Estimation Error 43
2.4.3 Interpolation Error 46
2.4.4 Overall Quality of Estimation and Control 48
2.5 Chapter Summary 52

3 Adaptive Learning Tree for DPM 53
3.1 Idle Period Grouping 54
3.2 Adaptive Learning Tree 57
3.2.1 Basic Structure 57
3.2.2 Decision 58
3.2.3 Learning 59
3.3 Power Manager 61
3.4 Experimental Results 63
3.5 Chapter Summary 69

4 Comparison of DPM Policies 70
4.1 Compared Policies 70
4.1.1 Adaptive Timout Policies 72
4.2 Policy Implementation 72
4.3 Experimental Results 74
4.4 Chapter Summary 79

5 Low Energy Software Optimization 81
5.1 Basic Idea and Overall Flow 82
5.1.1 Basic Idea and Problem Description 82

5.1.2 Framework Configuration and Transformation Flow 85
5.2 Profiling 87
5.2.1 The Structure of Profiler 87
5.2.2 Computational-Effort Estimation 88
5.2.3 Value Profiling 90
5.3 Common-Case Selection 93
5.3.1 Common Case Representation 93
5.3.2 Pruning Trivial Cases 95
5.4 Common-Case Specialization 99
5.4.1 Overview 99
5.4.2 Semi-exhaustive approach 100
5.4.3 One-shot approach 102
5.5 Global Effective-Case Selection 103
5.6 Experimental Results 108
5.6.1 Experimental Setting 108
5.6.2 Search space reduction 109
5.6.3 Code quality improvement 111
5.6.4 Input data sensitivity analysis 114
5.7 Chapter Summary 115

6 Conclusion **117**
6.1 Thesis Summary 118
6.1.1 Dynamic Power Management 118
6.1.2 Low Energy Software Optimization 119
6.2 Future Work 120

Bibliography **122**

List of Tables

1.1 Device parameters for energy consumption computation 10

2.1 Power states of commercial HDDs from Fujitsu and IBM 41

3.1 HDD specifications 63
3.2 Comparisons of the various policies 65
3.3 Comparisons for design guide 67

4.1 Compared policies 71
4.2 Algorithm Comparison 77
4.3 Experimental results for window-size sensitivity analysis on desktop PC 78

5.1 Notations for a hierarchical tree 94
5.2 Profiling information for the hierarchical tree shown in Figure 5.6 . . 95
5.3 Quality of the code transformed with different approaches (normalized to original code) 110
5.4 Improvement ratio of floating-point and fixed-point versions (semi-exhaustive) 113

List of Figures

1.1 The structural view of processor-based systems 2
1.2 An example of energy perspective system abstraction 3
1.3 Research objectives corresponding to each software layer 4
1.4 Energy perspective system abstraction with a sleep state 6
1.5 System power consumption level variation with DPM 7
1.6 An example of user behavior change 7
1.7 compute the break-even time of a device 9

2.1 Overall system model for DPM 24
2.2 An example of a Markov chain for stationary SR 25
2.3 State transition matrix and observed transition table of SR 28
2.4 An example of a 2D policy table ($NS_0 = NS_1 = 5$) 31
2.5 Decision Table and Rows Selection from Policy Table 31
2.6 Visualized 2-dimensional Interpolation Example 32
2.7 2-dimensional Interpolation 33
2.8 An example of a Markov Chain for non-stationary SR 34
2.9 $R_i(u_s)$ of non-stationary SR .vs. stationary SR 34
2.10 Single window operation for two-state user requests 36
2.11 Estimation error source 37
2.12 Multi-window operation for two-state user requests 38
2.13 Hardware setup for HDD power measurement 41
2.14 The logical diagram of Figure 2.13 42
2.15 Ideal and Estimated curve at $f = 0.001$ 44
2.16 Attenuation and phase shift of window-based estimates 45

2.17 Sub-optimality of interpolated policies. 47
2.18 Power and Average Waiting Time Comparison for Various Window Size ($L_p = 0.05$, $W_p = 1$) 49
2.19 Local power consumption and average waiting time provided by the PM policies for each sub-trace u_s of U^{20} ($L_p = 0.05, W_p = 1$). 51

3.1 An example of system shutdown with multiple sleep states 55
3.2 An adaptive learning tree (with two sleep states) 58
3.3 PCL operation 60
3.4 An example of learning for a prediction miss 60
3.5 Power Manager Configuration 61
3.6 Distribution of idle intervals 66
3.7 Cumulative hit ratio for IBM HDD 68

4.1 ACPI Interface and PC Platform 73

5.1 Example of source code transformation using the proposed technique 83
5.2 The configuration of the proposed framework 86
5.3 Overall source code transformation flow 86
5.4 An example of abstract syntax tree and instruction cost table 90
5.5 Internal data structure of value profiling 91
5.6 Hierarchical tree representation of common cases 94
5.7 An example of loop tree 101
5.8 A more compex example for global effective-case selection 104
5.9 An example of binary tree for M 104
5.10 Search procedure for the given binary tree 106
5.11 Search space reduction using common-case selection 108
5.12 Search space reduction ratio in common case and global effective-case selection step 109

Chapter 1

Introduction

1.1 Motivation

Design methodologies and tools for energy-efficient system-level design are receiving increasing attention [21, 69, 70, 85] because of the widespread use of portable electronic appliances (*e.g.*, cellular phones, laptop computers, *etc.*) and of the concerns about environmental impact of electronic systems.

Battery life time in portable systems can be prolonged in two ways - by increasing battery capacity per unit weight and by reducing power consumption with minimal performance loss. Between these two alternatives, the latter has been the major concern of designers because battery capacity (Watt-hours / kg) has only improved a factor 2 to 4 over the last 30 years, while the computational power of digital IC's has increased by more than 4 orders of magnitude [85].

Energy-efficient design requires the development of new computer-aided design (CAD) techniques to help exploring the trade-off between power and conventional design constraints, *i.e.*, performance and area. Numerous CAD techniques [70] have been researched and implemented at all levels of the design hierarchy to reduce power consumption, and many of these techniques target VLSI digital components. Modern portable systems are heterogeneous [61]. For example, the power breakdown for a well-known laptop computer [107] shows that the power consumed by VLSI digital components is only 21%, while the display, hard disk drive, and wireless LAN card

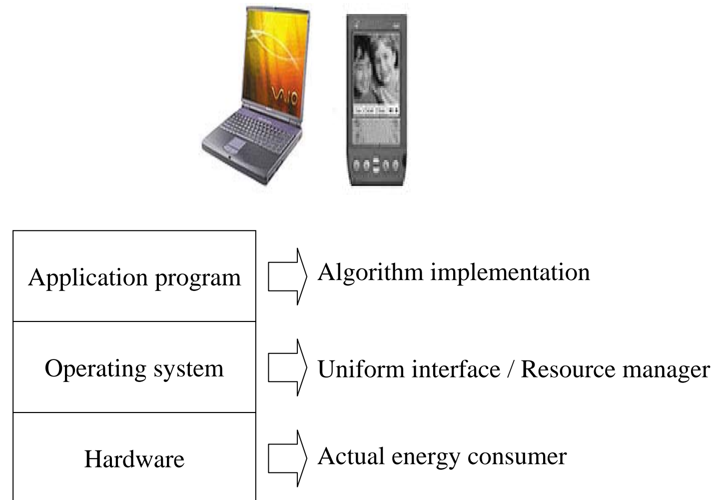


Figure 1.1: The structural view of processor-based systems

consume 36%, 18%, and 18%, respectively.

Recently, processor-based system architectures are adoped in many modern application domains such as telecommunications, consumer electronics, and multimedia [39, 75]. The major driving force to move from ASIC design to processor-based architectures is programmability, which increases flexibility and reduces the design time. Cost is also reduced, because the design is based on high-volume commodity parts (processor and memory), whereas ASIC solutions require low-volume custom components [56, 111].

Typically, processor-based portable systems (*e.g.* laptop computers and PalmPilots) can be structured into three layers as shown in Figure 1.1. The hardware resides at the botторm layer, the operating system (OS) layer is on top of the hardware layer, and application programs are the top layer. Application-program layer consists of a set of programs and each of them is executed for its specific application domain. On the other hand, the OS layer provides the uniform interface to application programs and manages hardware resources, and the hardware layer is the actual energy consumer. In these systems, two software layers running on the processor control the behavior of hardware. For this reason, the overall performance and energy consumption of processor-based design critically depends on software organization and quality.

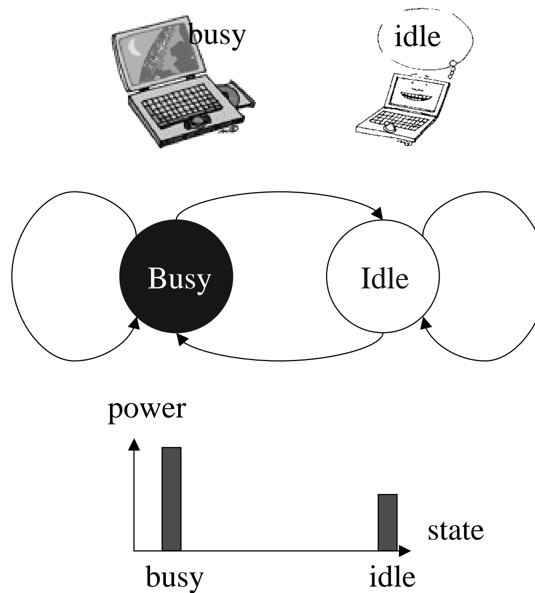


Figure 1.2: An example of energy perspective system abstraction

Software optimization is one of the most important issues in modern processor-based embedded system design [61, 108, 101, 85].

In this thesis, I will discuss how to enhance the two software layers to facilitate energy reduction. I will also describe how these enhancements affect the system performance.

1.2 Energy-Centric System Abstraction

From an energy point of view, processor-based systems can be represented by finite-state models. Consider a laptop computer as shown in Figure 1.2. While a user is working on the computer, the computer is in busy state. On the other hand, while a user is taking a break, and there is no application program running on the computer, the computer is in idle state. In other words, whenever a user changes his or her behavior, the state model changes its state. Usually, the system consumes

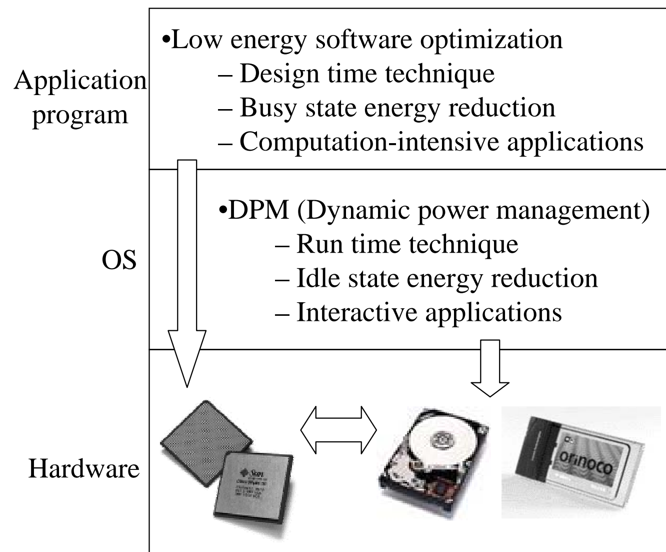


Figure 1.3: Research objectives corresponding to each software layer

different power levels depending on the state. For this reason, total energy consumption of the system is the sum of busy-state energy consumption and idle-state energy consumption.

Notice that Figure 1.2 is the simplest example of system abstraction from an energy perspective. More complex system abstraction may be required when a system has multiple busy and/or sleep states which consume different power levels. For example, a laptop computer consists of several components such as processor, hard disk, memory, and so on. Depending on the programs running on the computer, each system component shows large variation of its utilization and such utilization variation translates into different busy states.

1.3 Research Objectives

The major goal of this research is to restructure the software layers to reduce the energy consumption of the hardware layer. Each software layer requires different techniques due to its different role.

Figure 1.3 shows the research objectives in each software layer. *Dynamic Power*

Management (DPM) is an OS-level run-time technique aiming at reducing the idle-state energy consumption. DPM is a design methodology to reduce the energy consumption at the system level by selectively placing components into low-power states. The challenging areas of DPM are interactive applications because the system goes into idle state frequently.

On the other hand, the major objective at the application-program layer is to develop low-energy software optimization techniques. These techniques are software design time techniques for the busy-state energy reduction by decreasing the overall computations and appropriate for computation-intensive applications.

Typically, a system consists of several hardware components. For instance, a system consists of a microprocessor and some peripheral devices such as hard disk and network card. For computation-intensive programs such as image compression and expansion programs, the major energy consumer is the processor. On the other hand, for interactive programs such as web browser and editors, the peripheral devices play an important role in overall energy consumption. For this reason, I will show the effectiveness of DPM by applying it to hard disk drives while interactive programs are running on the processor. Also, the effectiveness of low energy optimization techniques will be shown by applying them to computation-intensive programs for processor energy reduction.

1.4 Dynamic Power Management

1.4.1 Overview

A system does not always need to operate at its peak performance because its workload changes continuously. In particular, this situation is commonly observed while interactive applications are running on the system. Suppose that a user is executing a web browser on a laptop computer. While the data is transmitted from the network, the system will be in busy state. However, the system with the exception of the LCD display will be in idle state while the user is reading the data displayed by the web browser. Therefore, while the user is reading the data, the system wastes energy and

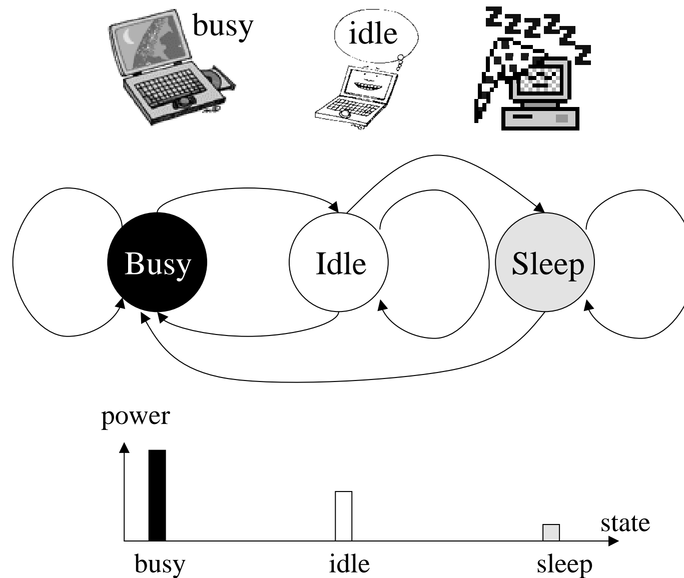


Figure 1.4: Energy perspective system abstraction with a sleep state

does not have to run at its peak performance.

DPM is a flexible and general design methodology aiming at controlling performance and power levels of electronic systems, by exploiting the idleness of their components. For this purpose, a system is provided with sleep states in addition to the states (busy state and idle state) in Figure 1.2 and abstracted as shown in Figure 1.4. In sleep state, a system is shutdown and cannot serve any requests from the user while consuming minimal amount of power.

Also, a system may need a *power manager* (PM) that monitors the overall system and component states and controls the power state of each component. This control procedure is called *power management policy.* The problem in DPM is that changing power state (*e.g.* spin up and down a disk drive) imposes a penalty in terms of both power and performance. Generally, the power state transition from idle state into the sleep state is called shutdown and the power state transition from sleep state to busy state is called wakeup.

Figure 1.5 shows (a) the system power consumption level over time without DPM,

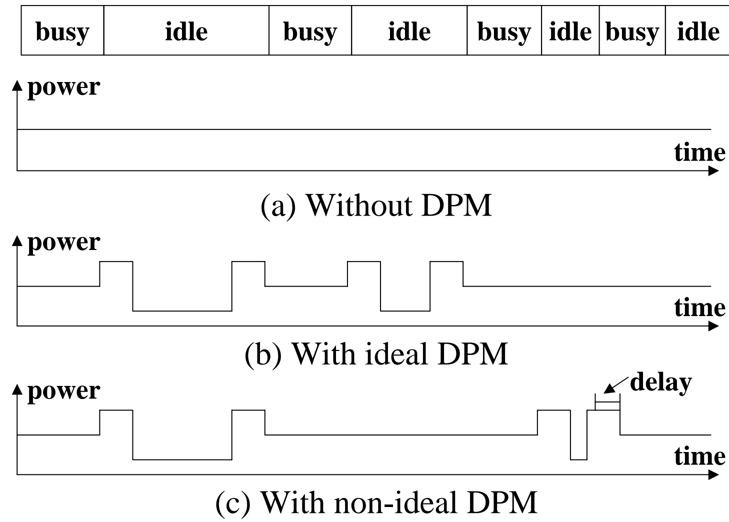


Figure 1.5: System power consumption level variation with *DPM*

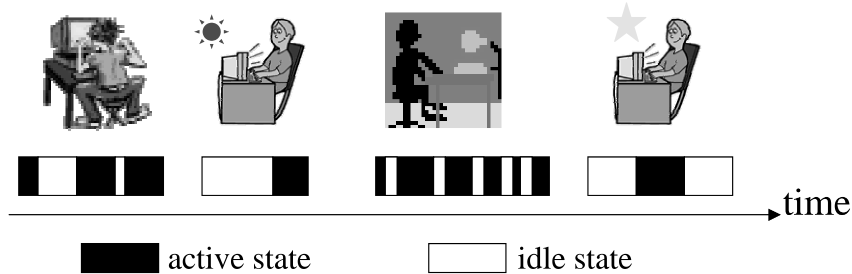


Figure 1.6: An example of user behavior change

(b) the case when the ideal *DPM* is applied, and (c) the case when non-ideal *DPM* is applied. Non-ideal *DPM* wastes the idle interval at the second idle period and pays performance penalty at the third idle period because the idle period is not long enough to amortize the shutdown and wakeup penalty. These inefficiencies come from the inaccurate prediction of the duration of the idle period, or, equivalently, the arrival time of the next request for an idle component.

An example for the inaccurate prediction is shown in Figure 1.6. Suppose that a laptop computer is shared by multiple users. Whenever a user generates a request, the system goes into the busy state to serve it and after finishing the service, the

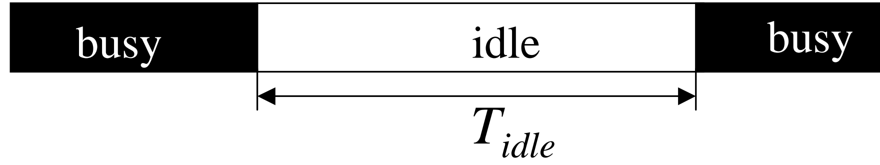
system goes into the idle state and waits for the next request. Each user will show a different behavior (request rate) in using the laptop computer. In other words, the length of busy periods and idle periods will vary significantly depending on the user. Furthermore, the same user can change his or her behavior from time to time. For instance, the user behavior while using a web browser will be different from the user behavior while using an text editor. Therefore, there is unavoidable uncertaintiy on future user requests. More precisely, the user request rate changes over the time and such time-variant property is called *non-stationary* property.

An ideal PM assumes that it has the complete knowledge of present, past and future workloads. In other words, an ideal PM is not affected by the non-stationarity due to the complete knowledge of workloads. However, in practice, it is impossible to have the complete knowledge of future workloads. Therefore, practical *DPM* algorithms should be adaptive to the non-stationarity to increase their efficiency. In some cases, some partial knowledge can be provided by applications that provide hints on the future requirements of system resources [47, 35]. Unfortunately, such application-driven approach requires the modification of the applications, and thus partial knowledge cannot be provided when the applications cannot be modified.

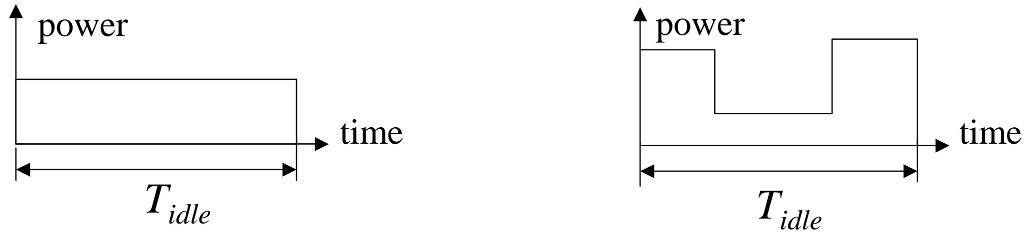
In this thesis, I take an application-independent viewpoint and present two new adaptive DPM approaches that heuristically predict the idle period length to cope with the *non-stationary* property of user behavior.

1.4.2 Related Work

Previous DPM policies can be classified into three major categories: timeout, predictive, and stochastic. Although these approaches proposed different methods to exploit the idleness, they have the common rationale to make a decision for shutdown. For this reason, I will first discuss the shutdown criteria for DPM and then the previous work in each category will be discussed in order.



(a) An example of workload for break-even time



(b) Energy consumption without shutdown (c) Energy consumption with shutdown

Figure 1.7: compute the break-even time of a device

Shutdown Criteria

The first mission of DPM techniques is to identify the idle period long enough to shut down the system for amortizing the shutdown and wakeup penalty. The *break-even time* (t_{be}) is the criterion to determine whether the given idle period is long enough to shut down the system.

Figure 1.7 illustrates the concept of break-even time. Figure 1.7 (b) graphically represents the energy consumption for the idle period shown in Figure 1.7 (a) when the system is not shut down. Using the parameters introduced in Table 1.1. The energy consumption over the time interval t_{idle}, can be expressed as:

$$E_{no-shutdown} = t_{idle} \cdot P_{idle} \tag{1.1}$$

On the other hand, Figure 1.7 (c) graphically represents the energy consumption for the same idle period when the system is shut down. Its corresponding expression is:

t_{be}	break-even time
$t_{shutdown}$	shutdown delay
t_{wakeup}	wakeup delay
t_{idle}	idle period length
$P_{shutdown}$	power consumption during shutdown
P_{wakeup}	power consumption during wakeup
P_{sleep}	power consumption in sleep state
P_{idle}	power consumption in idle state
$E_{no-shutdown}$	energy consumption without shutdown for t_{idle}
$E_{shutdown}$	energy consumption with shutdown for t_{idle}

Table 1.1: Device parameters for energy consumption computation

$$\begin{aligned} E_{shutdown} = {} & t_{shutdown} \cdot P_{shutdown} + t_{wakeup} \cdot P_{wakeup} \\ & + \max((t_{idle} - (t_{shutdown} + t_{wakeup})), 0) \cdot P_{sleep} \end{aligned} \tag{1.2}$$

Note that we assume that it is impossible for the system to stay in sleep state when t_{idle} is shorter than the sum of $t_{shutdown}$ and t_{wakeup}.

The break-even time is the period such that $E_{no-shutdown}$ is exactly the same as $E_{shutdown}$. In other words, there is no difference in energy consumption regardles of power state changes. The break-even time is expressed as Equation 1.3, if the energy overhead of shutdown and wakeup is larger than the energy wasted in idle state, *i.e.* $t_{shutdown} \cdot P_{shutdown} + t_{wakeup} \cdot P_{wakeup} > t_{idle} \cdot P_{idle}$.

$$t_{be} = \frac{t_{shutdown} \cdot (P_{shutdown} - P_{sleep}) + t_{wakeup} \cdot (P_{wakeup} - P_{sleep})}{P_{idle} - P_{sleep}} \tag{1.3}$$

If the idle period length is longer than the break-even time, the system should be shut down for energy reduction. Otherwise, the shutdown should be avoided because it causes more energy consumption due to the shutdown and wakeup overhead.

To summarize, the effectiveness of *DPM* policies critically depends on how to predict accurately the upcoming idle period, whether it is longer than the *break-even*

time or not.

Timeout Policy

The timeout policy [40] is the most widely used in many applications such as microprocessors, monitors, hard disk drives, *etc.* because of its simplicity. The value of the timeout can be fixed (static timeout) or it may be changed over time (adaptive timeout). An effective static timeout policy sets the timeout to the *break-even time* of the power-managed device [46]. Roughly speaking, t_{be} is the minimum idle time for which it is convenient to shut down the device, because the power savings in the *sleep* state should compensate for the shutdown and wakeup costs [12]. It can be shown [46] that setting the timeout to t_{be} produces a *2-competitive* policy, which is guaranteed to be within a factor of two from the power savings that could be achieved by an ideal policy with perfect knowledge of the future (*i.e.*, an *oracle* policy).

Static timeout policies use a fixed timeout value. Several adaptive timeout policies have been introduced to reduce wasted idle time by changing the timeout threshold depending on previous idle period history [37, 38, 30, 62].

To summarize, timeout policy is widely used due to its simplicity, but the main shortcoming of timeout policies is that they waste power waiting for the timeout to expire. This inefficiency motivates the search for more effective techniques.

Predictive Policy

Srivastava et al. [94] proposed heuristic policies to shut down a system selectively as soon as an idle period begins. The basic idea in [94] is to predict the length of idle periods and shut down the system when the predicted idle period is long enough to amortize the cost (in latency and power) of shutting down and later re-activating the system. A shortcoming of the predictive shutdown approach proposed by Srivastava is that it is based on offline analysis of usage traces, hence it is not suitable for non-stationary request streams whose statistical properties are not known a priori. This shortcoming is addressed by Hwang and Wu [44]. They proposed online adaptive methods that predict the duration of an idle period with an exponentially weighted

average of previous idle periods.

However, all predictive shutdown techniques share a few limitations. First, they do not deal with resources with multiple sleep states. Second, they cannot accurately trade-off performance losses (caused by transition delays between states of operation) with power savings. Third, they do not deal with general system models where multiple incoming requests can be queued while waiting for service

To summarize, predictive policies are more aggressive than timout policies by making a decision for the shutdown at the beginning of idle period. But they still need to be improved to overcome the three major limitations mentioned above.

Stochastic Policy

Stochastic policies solve the limitations of predictive approaches by modeling the unavoidable uncertainty of future requests (or workload / user behavior) and the behavior of system components as a stochastic process. Component behavior is also modeled as a stochastic process. Even if the realization of a stochastic process is not known in advance, its properties can be studied and characterized. This assumption is at the basis of many stochastic optimal control approaches that are routinely applied to real-life systems [96, 78, 114]. *DPM* has been formulated and solved as a discrete-time stochastic optimal control problem by Benini et al. [11]. Also, continuous-time stochastic approaches were proposed in [79, 80, 82, 90].

In [11], general systems (with multiple states and queuing) and user request are modeled as discrete-time Markov decision processes. The discrete-time Markov model enables a rigorous formulation of the search for optimal power management policies as a constrained *stochastic optimization* problem whose exact solution can be found in polynomial time. Also, it provides a flexible way to control the trade-off between power consumption and performance penalty depending on the optimization constraints. A few extensions to the discrete-time Markov model have been proposed in the recent past.

To reduce the power cost of the power manager which observes and issues command at regular time-discretization intervals, continuous-time (event-based) formulations have been proposed [79, 80, 82, 90, 92]. The technique in [79] was further

extended to handle more complex systems (multiple devices) in [81], or to control the power states of the system with the consideration of a side metric - *quality of service* [113]. For systems where running the power manager at regular time-discretization intervals imposes an unacceptable power cost, continuous-time (event-based) formulations have been proposed [79, 90]. Also, the authors in [92] proposed to use the distributions other than exponential to model the system behavior more precisely. For example, they modeled the user behavior as a Pareto distribution and the wakeup and shutdown time penalties were modeled as a uniform distribution.

Unfortunately, a common basic assumption in [11, 79, 80, 82, 92] is that the Markov model is stationary and known in advance. Such an assumption clearly does not hold if the system experiences non-stationary workloads. Furthermore, for each idle interval, only a single shutdown decision is allowed in continuous-time methods, thus a wrong decision may critically degrade their effectiveness. In contrast, the discrete-time methods naturally have multiple chances to change its decision. In [92], the continuous-time methods were extended to overcome this limitation based on the renewal theory and the time-indexed semi-Markov decision process model.

To summarize, stochastic policies overcome the limitations of predictive policies, but they are not practical in many real-life applications due to the assumption that the workloads are stationary.

1.5 Low Energy Software Optimization

1.5.1 Overview

DPM is an effective technique for interactive applications by exploiting idleness, but it is not appropriate for the computation intensive applications because these applications rarely allow a system to be in idle state due to a large amount of computations and tight timing constraints.

Processor-based embedded systems are pervasive in many modern application domains such as telecommunications, consumer electronics, and multimedia [39, 75]. The major driving forces to move from ASIC to processor-based architectures are

the following. First, processor-based embedded systems provide the programmability which increases flexibility and reduces the design time. Second, design cost is also reduced, because the design is based on high-volume commodity parts (processor and memory), whereas ASIC solutions require low-volume custom components [56, 111].

The overall performance of processor-based design critically depends on software quality because the processor on which software is running is the major energy consumer for computation-intensive applications. For this reason, software optimization is one of the most important issues in modern embedded system design [61, 108, 101, 85].

Embedded software can be optimized more aggressively than general-purpose software for several reasons. First, embedded software can be often characterized by a few well-known workloads, thus profiling-driven optimization can be effectively used for embedded software development. Second, embedded software only needs to be optimized for the specific target hardware, while general-purpose software should be optimized with the consideration of the various hardware platforms. Third, embedded software optimization takes the advantage of the reduced compilation speed requiremenet with respect to the general-purpose software compilers, therefore embedded software development tools can adopt more complex and aggressive approaches which are not allowed in general purpose software development. Such optimization is often a critical step for striking design targets under tight cost constraints, which are typical of embedded systems.

A traditional quality metric for embedded software is compactness: the most compact code for a program uses the least instruction memory. Moreover, if such program represents a pure data flow (*i.e.* no branching and iteration is involved), it executes in the shortest time and consumes the least energy under the assumption that the cost of each instruction is roughly constant. However, in a real situation, as algorithm complexity grows, the control dependency of a program increases and specific architectural features of a processor may favor some instructions over others in terms of performance and energy consumption. Thus, two additional metrics, namely performance and energy, are considered in embedded software design. It is also very important to distinguish between average- and worst-case performance because many

embedded systems are targeting real-time applications [56].

Average-case performance is tightly related to energy efficiency, because short execution time can be directly translated into reduced energy by slowing down the system's clock (or by gating the clock) and/or by lowering the voltage supply [10, 13, 85]. At the same time, however, worst-case performance should not be adversely affected when optimizing for average case. In other words, while minimizing the expected value of program execution time, variance should remain under control. In this context, I propose an automatic source-code transformation framework aiming at reducing the computational effort (*the average number of executed instructions*) with tightly controlled worst-case performance and code size degradation.

According to Amdahl's law, the most effective way to improve the average case performance is to make the common case fast. Many code transformation techniques adopt execution frequency profiling to identify the most frequently-executed code blocks [77, 9], or *computational kernels.* Then, the kernels can be optimized either by eliminating redundant operations or by matching computation and memory transfer to the characteristics of the hardware platform (e.g., parallelizing computation, improving locality of memory transfers) [13, 8, 112].

Execution frequencies of program fragments are not the only profiling information that can be used for code optimization. Recently, *value profiling* has been proposed as a technique for identifying a new class of common cases for a given program [16, 17, 36]. The common cases identified by value profiling are code fragments which frequently execute operations with the same operand values. In this case, the identified code fragments can be specialized for the commonly observed operand values to eliminate redundant computations.

Profiling-driven optimization is often very effective for embedded systems because embedded software shows value locality and can be characterized by a few well-known workloads, unlike software running on a general purpose system. For example, many DSP programs execute filter operations and the filter coefficients are rarely changed.

Partial evaluation is an appropriate technique to exploit the value locality by specializing a procedure with respect to a subset of its parameters, where these parameters are held constant [45, 27]. In partial evaluation, procedure calls, which are

frequently executed with rarely varying parameter values, are defined as common cases. Such common cases are identified by value profiling and specialized by *partial evaluation*. Even though partial evaluation is a well-developed field, there are several issues in its application that have not been fully addressed in the past. First, the procedures to be specialized, their parameters, and parameter values for specialization are assumed to be specified by the user. Second, partial evaluation sometimes leads to code size blowup. If applied in an uncontrolled fashion, it can actually make performance and energy consumption worse. Third, when multiple procedures within a program are specialized, the interplay among various specialized calls is rarely taken into consideration (refer to the example in Figure 5.1). Because of these limitations, program specialization based on partial evaluation is not widely applied.

In this thesis, I will discuss a source-code transformation framework and tool based on profiling (both execution frequency and value profiling) and partial evaluation to overcome the limitations of partial evaluation in an automated fashion. The framework integrates execution frequency and value profiling, candidate computational kernel selection, partial evaluation, performance and energy estimation within a single optimization engine. Its input is a target program (C source code) with typical inputs. The output is optimized source code, and estimates of average execution time and energy for the original and optimized version of the target program. The impact of optimization is assessed by instruction level simulation on the target hardware architecture [68, 33, 55].

1.5.2 Related Work

Whereas the objective of most software optimizations for general purpose computers is average case performance, the requirements for embedded software are more articulated [66]. Code compactness is often a high-priority objective [57, 29, 52]. Also, energy efficiency as well as performance are becoming important issues in embedded software design [102].

Retargetability is another key requirement for embedded software optimization tools, because of the wide variability of target hardware platforms [99, 74, 53, 58].

Also, compiler development for specific application domains such as digital signal processing was researched to exploit special features of application-specific processor architectures [86, 5]. Most research on optimizing compilers for embedded processors has focused on fairly low-level optimizations, such as instruction scheduling, register assignment, etc. Embedded software optimization takes advantage of the reduced compilation speed requirement with respect to general-purpose software compilers. Therefore embedded software development tools can adopt more complex and aggressive approaches which are not allowed in general purpose software development.

Both compactness and retargetability basically require the instruction-level program analysis. Due to this effect, performance and energy efficiency improvement were also studied in instruction level in the past [101, 102, 104, 98].

Recently, high-level approaches (based on source to source transformations) to improve code quality were proposed. Memory-oriented code transformation techniques were proposed in [18, 72] and other classical high-level loop transformations for general purpose software were applied to embedded software optimization [68, 33, 55]. Source-to-source techniques are more aggressive in modifying the target program, and they can be applied together with more traditional optimizing compilers in the backend. One of the major concerns in the adoption of high-level optimizations is that they are hard to control, and they are often meant to be used in a semi-automated flow that requires programmer's guidance.

Value locality is a promising property for general purpose software optimization, but it has not been studied in depth for embedded software. Value locality is defined as the likelihood of a previously-seen value recurring repeatedly within a physical or logical storage location [59]. Value locality enables to reduce the computational cost of a program by reusing previous computations.

Previous work shows that value locality can be exploited in various ways depending on the target system architecture. In [49], common-case specialization was proposed for hardware synthesis using loop unrolling and algebraic reduction techniques. In [59, 51], value prediction was proposed to reduce the load/store operations with the modification of general purpose microprocessor. Also, in [87], redundant computation (an operation performs the same computation for the same operand

value) was defined and *result cache* was proposed to avoid redundant computations by reusing the result from the *result cache*. Unfortunately, these techniques are not appropriate for our case, because they are architecture dependent. For this reason, I will focus on pure software oriented approaches exploiting value locality (*i.e.* partial evaluation) in this thesis.

Depending on the way of using the result of previous computations, partial evaluation can be classified into two categories, *i.e.* program specialization and data specialization. Program specialization encodes the results of previous computations in a *residual program*, while data specialization encodes these results in the data structures like caches [22].

Program specialization is more aggressive in the sense that it optimizes even the control flow, but it can lead to a code explosion problem due to over-specialization. For example, code explosion can occur when a loop is unrolled and the number of iterations is large. Furthermore, code explosion can degrade the performance of the specialized program due to increased instruction cache misses. On the other hand, data specialization is much less sensitive to code explosion because the previous computation results are stored in a data structure which requires less memory than the textual representation of program specialization. However, this technique should be carefully applied such that the previously cached computations are expensive enough to amortize the cache access overhead. The cache can also be implemented in hardware to amortize the cache access overhead [87].

To summarize, instruction-level optimization was a major trend in low-energy software optimization in the past. Recently, high-level code optimization (source-to-source transformation) has become a new trend in embedded software design to achieve higher degree of performance and energy improvements. One of the promising high-level techniques is program specialization based on value profiling. However, program specialization has a few limitations including code explosion and needs a method to overcome the limitations for practical use.

1.6 Thesis Contributions

This thesis focuses on energy reduction techniques for processor-based systems. The contributions can be summarized in two parts.

The first contribution is an OS-level energy reduction technique. I propose two adaptive *Dynamic Power Management* policies to handle the non-stationary service requests of the intractive applications for energy saving. Also, these techniques are implemented on both laptop and desktop computers to control the power states of their hard disk drives.

The second contribution is an application-program-level energy reduction technique. I propose an automated low-energy software optimization framework to improve average energy consumption as well as performance by specializing the computation intensive application programs. The framework performs a source to source-code transformation using the value locality obtained from the profiling. The framework is implemented based on the SUIF [100] and the transformation effect is validated on ARM [6] and ST200 [95] processors.

1.6.1 DPM for Non-Stationary Service Requests

The first technique is a predictive policy called *sliding window technique* and the second one is a stochastic policy called *adaptive learning tree*.

The sliding window technique tackles the non-stationary property of user behavior, which is the most critical limitation of all stochastic DPM policies. This technique adopts parameter-learning schemes for the workload source (*e.g.*, the user, also called service requestor) that capture the non-stationarity of its behavior. The time-varying parameters of the stochastic model of the workload source are monitored within sliding windows. The captured time-varying parameters are well integrated with an existing stationary stochastic policy [11] by interpolating the optimal policies computed under the assumption that user requests are stationary.

The *adaptive learning tree* technique is different from the previous predictive techniques, because it supports multiple-sleep state devices to exploit idleness more efficiently by selecting the most appropriate sleep state depending on the length of the

idle period. For this purpose, the concept of multiple break-even times is used. Based on this concept, the idle period length is quantized and each quantum is mapped to each sleep state. The adaptive learning tree technique records the most recent idleness history in the form of a tree and predicts the corresponding quantum of the next idle period from the tree.

Both *sliding window* and *adaptive learning tree* techniques are implemented in Windows2000 from Microsoft running on latop and desktop computers with hard disk drives to show their feasibilities in real system environments. The *sliding window* technique requires a pre-characterization step to build a set of policies to cope with the non-stationary user behavior, while this step is not necessary in *adaptive learning tree* technique. But the pre-characterization step in *sliding window* technique allows us to trade off precisely between energy consumption and performance, which is the benefit of the *sliding window* technique over the *adaptive learning tree* technique. The experimental results show that the proposed methods outperform other DPM policies in terms of both energy reduction and performance penalty. In the best case, the *sliding window* technique improves the energy consumption of a hard disk drive up to more than a factor of two compared to the hard disk drive without power management.

1.6.2 Low Energy Software Optimization Framework

The technique presented in this thesis is based on program specialization without any hardware assistance for embedded software design and implemented as a tool-chain. The proposed technique differs from previous approaches [27] as follows.

First, I propose a computational effort estimation technique which combines value profiling with execution frequency profiling. Using the estimation technique, it is possible to identify the common cases (computationally intensive procedure calls with their effective known parameter values for the specialization) in an automated fashion.

Second, this approach provides a systematic loop controlling strategy to avoid the code explosion problem (which was manually controlled by the user in previous work).

Third, this approach supports the inter-procedural effect analysis of the program specialization which was mentioned only in a few papers [28]. This analysis is especially important when multiple procedure calls are specialized.

I adopt offline partial evaluation rather than online partial evaluation, since offline partial evaluation is more advantageous in the context of real-time applications, which are common embedded systems [71]. In detail, the advantages of offline partial evaluation can be summarized as two reasons.

First, online partial evaluation does not guarantee the worst case performance due to the dynamic optimization and run-time code generation, while it is possible to avoid such inefficiency by offline partial evaluation because it is not the run-time technique.

Second, the intrinsic problems of partial evaluation - code explosion and inter-procedural effect analysis for multiple specialized calls cannot be handled properly at run time due to their complexities.

The advantages of the proposed method are implemented in a single framework to automate the overall code transformation procedure with minimal human intervention. Also, the effectiveness of the framework was validated in two different processor environments, *i.e.*, strong-ARM and ST200 processors. Several DSP programs transformed using the framework show great improvements of both performance and energy consumption. Our tool improves both performance and energy consumption of the source code up to more than a factor of two and in average about 35% over the original program.

1.6.3 Limitations and Future Work

The *DPM* techniques in this thesis consider single component instead of multiple components to be power-managed. The concurrent power management of multiple components is still challenging area to be researched because it is more practical and the overall energy reduction will increase. The proposed *DPM* methods can be extended to solve this challening problem, but the extension may face in some problems like the overhead of the power manager.

The low-energy software optimization technique also has the possibility of being further improved. For example, the current optimization framework only considers procedure-level optimization. But the lower level (*i.e.* loop-level) optimization may be more effective for both performance and energy consumption. Also, for energy critical systems, it is possible to achieve further energy saving by combining the technique with *Dynamic Voltage Scaling* (DVS). The combination of these two techniques enables us to translate the performance improvment into further energy reduction.

1.7 Thesis Organization

Chapter 2 explains the first adaptive DPM technique, *sliding window technique*. I also demonstrate its effectiveness by applying it to hard disk drives and comparing it to the previous stationary DPM policies. Chapter 3 presents the second adaptive DPM technique, *adaptive learning tree technique* and shows its effectiveness by applying it to the hard disk drives with multiple-sleep states. Chapter 4 compares the proposed adaptive DPM techniques to other DPM policies by implementing them in Windows2000 running on laptop and desktop computers. The efficiency of each policy is measured by energy consumption and performance penalty. Chapter 5 describes the low-energy software optimization framework implemented based on SUIF. The effectiveness of the framework is shown by applying it to two well-known processors, strong-ARM and ST200. Finally, Chapter 6 concludes this thesis and discusses the directions for future work.

Chapter 2

Sliding Window Technique for DPM

This chapter and the following two chapters describe adaptive DPM techniques. Chapter 2 and Chapter 3 will describe a stochatic DPM policy and a predictive DPM policy, respectively. Chapter 4 will compare these two techniques to other DPM policies by measuring the energy consumption and performance penalty of the hard disk drives installed in laptop and desktop computers.

The *sliding window technique* to be discussed in this chapter extends the stationary stochastic DPM policy to handle the non-stationary property of the service requests. I will explain the adaptive DPM in steps. First, I will describe the basics of the stochastic optimal control formulation in the case of a known and stationary environment. Then, I will introduce adaptive techniques for the unknown stationary case. In the case of DPM, estimation of the unknown parameters is decoupled from the control of the power-managed system. Hence, it is possible to exploit theoretical results on adaptive stochastic control to prove asymptotic optimality of DPM control policies for initially unknown Markovian workloads. Finally, I will extend the proposed approach to non-stationary, general workloads. In this case, optimality cannot be proven, and DPM algorithms are heuristic. The effectiveness of heuristic DPMalgorithms will be demonstrated through extensive simulation analysis

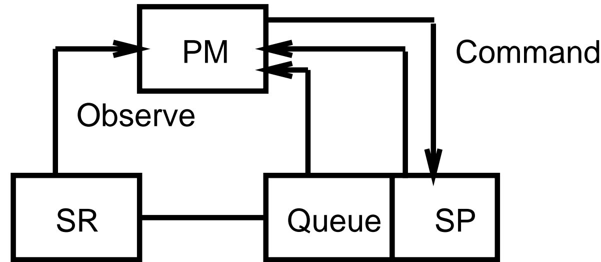


Figure 2.1: Overall system model for DPM

2.1 *DPM* in Known Stationary Environment

In this section, I briefly review the system model introduced in [11]. The model for non-stationary requests described in Section 2.3 can be seen as an extension of the stationary approach of [11], with the consideration of time-varying request probability.

The overall system model for DPM is shown in Figure 2.1. An electronic system is modeled as a unit providing a service, called *service provider* (SP) while receiving requests from another entity, called *service requestor* (SR). A *queue* (SQ) buffers incoming unserviced requests. The service provider can be in one of several states (*e.g. active*, *sleep*, *idle*, *etc.*). Each state is characterized by the ability (or the inability) of providing a service and by a power consumption level. Transitions among states may have a performance penalty (*e.g.*, latency in reactivating a unit) and a power penalty (*e.g.*, power loss in spinning up a hard disk).

The *power manager* (PM) is a control unit that controls the transitions among states. I assume that the power consumption of the PM is negligible with respect to the overall power dissipation [1]. At equally spaced instants in time, the power manager evaluates the overall state of the system (provider, queue and requestor) and decides to issue a command to stimulate a state transition. For the sake of conciseness, I borrow the following notations to represent the states of the units and PM commands from [11]:

- SP: $s_p = \{0, 1, \cdots, S_p - 1\}$

[1] This assumption has been validated in practice for several classes of systems [12, 64], and it will analyzed in detail later in this chapter.

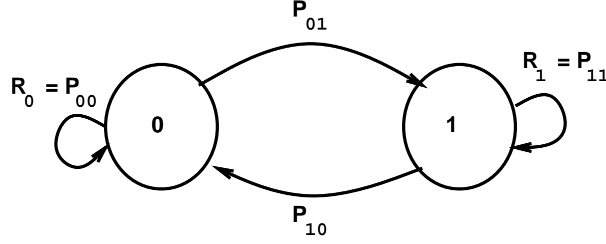


Figure 2.2: An example of a Markov chain for stationary SR

- SR: $s_r = \{0, 1, \cdots, S_r - 1\}$
- SQ: $s_q = \{0, 1, \cdots, S_q - 1\}$
- A: $a = \{1, 2, \cdots, N_a\}$

where A is a command set issued by PM to control the power state of SP. I model the system components as discrete-time Markov chains [11]. In particular, I use a controlled Markov chain model for the system provider, so that its transition probabilities depend on the command issued by the power manager.

In [11], the SR as well as SP are modeled as stationary processes. A generic requestor can have S_r states. I will discuss the case of $S_r = 2$, as shown in Figure 2.2. SR stays in state 0 when no request is issued for the given time slice, otherwise SR is in state 1. The corresponding transition matrix is denoted by P_{SR}. I call the diagonal elements of P_{SR} *user request probabilities* and I denote them by $R_i, i = 0, 1$. The probabilities $R_i = Prob(s_r(t+1) = i | s_r(t) = i), i = 0, 1$ (and the entire transition matrix P_{SR}) are time-invariant for a stationary workload.

With this assumption, the entire system model does not include any time-variant parameters, thus the complete system can be described by a controlled stationary Markov chain. The *control policy* for the given system model can be optimized under performance or power constraints by solving a linear programming problem. Details are provided in [11].

The optimal policy for a Markov system model is also Markovian, *i.e.*, the decision at any point in time depends only on the current system state instead of the entire past history. Therefore, a policy, the final result from policy optimization, can be thought

of as a matrix, that associates a probability of issuing each command ($a \in A$) with each system state. The matrix is called a *decision table* and its dimension is $S \times N_a$, where $S = S_r \times S_p \times S_q$.

A *control policy* is a sequence of decisions. At each time slice, the PM observes the current system state and issues a command based on the probability of each command for the given system state in the decision table. The decision made at each time slice i is denoted as δ_i and the policy π is the sequence of the decisions.

Even though this approach provides a way to obtain an exact solution and control the trade-off between performance and power, the following two assumptions limit its practical application:

- The user request probabilities for the given workload are known through offline analysis.
- The user request probabilities for the given workload are constant over time.

The workload of many practical systems is not stationary in time. Furthermore, statistic workload characteristics may not be available for offline policy optimization. Therefore, we need to extend the approach by relaxing these two assumptions. In Section 2.2, the approach is first extended to the unknown stationary environment by relaxing the first assumption, and in Section 2.3, it is further extended to the unknown non-stationary environment.

2.2 DPM in Unknown Stationary Environment

In Section 2.1, it is assumed that SR can be characterized through offline analysis of stationary workloads. Offline analysis at design time can be impossible in practice, especially for general-purpose systems (such as PCs or workstations), where workload strongly depends on the applications that the end user will run on the general-purpose platform. For these reasons, even the straight-forward timeout PM policies implemented on current portable computers are user-customizable. This customization process puts on the user the responsibility of following the trial-and-error process

that leads to the choice of an optimal timeout value. While this choice may be acceptable for tuning a single timeout, it is certainly not possible to assume that the user would be able to manage the complex characterization process for a Markov model of system workload. Thus, we need techniques for automatically "learning" a Markov workload model and for computing the corresponding optimal PM policy. Clearly, both estimation and policy optimization overhead should be small for online application in real-life systems.

Our approach will follow classic techniques of adaptive control theory [48, 43], based on the principle of estimation and control. The unknown parameters of the stochastic system (*i.e.*, the parameters that characterize the workload) are estimated with estimators that are guaranteed to converge (with probability one) to the true parameter values. The policy applied at each time step is chosen assuming that the current parameter estimates are the true values. It can be shown that, under some restrictive assumptions (which are verified in our case), this approach (henceforth called *EC* for "estimation and control") leads to a *self-tuning* policy, *i.e.*, a control law that produces the same long-term average cost as the stationary, optimal policy obtained with complete a priori knowledge of parameter values [48].

For the estimation technique, I will adopt Maximum Likelihood Estimation (MLE), which satisfies the requirements for statistical convergence towards true parameter values. MLE is described in Subsection 2.2.1. For the computation of the optimal control law for a given ML parameter estimate, we could in principle apply the exact optimization techniques introduced in [11]. In practice, this solution may not be applicable, because the computational burden of re-computing an optimal policy every time slice could be sizable, thereby violating the assumption that the power manager takes fast decisions with negligible power.

Therefore, I propose a novel table look-up method with linear interpolation (described in Subsection 2.2.2) to compute the control law to be applied at each time step. Using a look-up table is equivalent to enforcing a discretization on the continuous range of optimal control policies. This may, in principle, prevent the achievement of an optimal solution. Fortunately, theoretical results [43] show that a succession of optimal discretized policies tends to the optimal policy as discretization is refined.

$$\mathbf{P_{SR}} = \begin{pmatrix} P_{11} & P_{12} & \cdots & P_{1m} \\ P_{21} & P_{22} & \cdots & P_{2m} \\ \vdots & \vdots & & \vdots \\ P_{m1} & P_{m2} & \cdots & P_{mm} \end{pmatrix}$$

(a) State transition matrix of SR

state	1	2	$\cdots$	m	n_i
1	n_{11}	n_{12}	$\cdots$	n_{1m}	n_1
2	n_{21}	n_{22}	$\cdots$	n_{2m}	n_2
:	:	:		:	:
:	:	:		:	:
m	n_{m1}	n_{m2}	$\cdots$	n_{mm}	n_m
					n

(b) Observed transition table of SR

Figure 2.3: State transition matrix and observed transition table of SR

Furthermore, linear interpolation helps in reducing discretization errors, as demonstrated by our experiments.

2.2.1 Estimation of Stationary SR

Maximum Likelihood Estimation (MLE) produces estimators that are consistent and, under certain regularity conditions, can be shown to be most efficient in an asymptotic sense (*i.e.* as the sample size n approaches infinity) [106]. The principle of this method is to select as an estimate of θ the value for which the observed sample is most likely to occur.

Suppose there is a service requestor SR, which has the state space $S_r = (1, 2, \cdots, m)$ and the SR is observed until n transitions have taken place. Then, the state transition matrix of SR can be represented as Figure 2.2.1 (a) and the observed transitions can be collected in tabular form as shown in Figure 2.2.1 (b), where n_{ij} is the number of transitions observed from state i to state j and $n_i = \sum_{j=1}^{m} n_{ij}$.

Then, MLE of the given SR is $\hat{P}_{ij} = \frac{n_{ij}}{n_i}$ $i, j = 1, 2, \cdots, m$. I do not show the details of the derivation for MLE, but the complete derivation can be found in [14]. The estimator may be biased for small n, thus in our approach, every transition from time slice 0 is recorded in the transition table like Figure 2.2.1 for the estimation. As n increases, $\hat{P}_{ij}$ will converge to a certain time-invariant matrix like the transition probability matrix of SR in stationary known environment. In other words, for reasonably large n, $\hat{P}_{ij}(n) \approx P_{ij}(\infty)$.

We can define the convergence time of estimation by introducing the tolerable error, ϵ. Then, the convergence time is the smallest n, such that $|\hat{P}_{ij}(n) - P_{ij}(\infty)| \leq \epsilon$ for $\forall i, j$.

The $n - step$ transition probability can be computed from $1 - step$ transition probability and initial probability vector. Namely, it is a function of $1 - step$ transition probability and n [106].

For example, $P_{00}(n)$ of two-state SR can be represented as follows [106].

$$\begin{aligned} P_{00}(n) &= \frac{P_{10}(1) + P_{01}(1) * (1 - P_{01}(1) - P_{10}(1))^n}{P_{01}(1) + P_{10}(1)} \\ &= P_{00}(\infty) + \frac{P_{01}(1) * (1 - P_{01}(1) - P_{10}(1))^n}{P_{01}(1) + P_{10}(1)} \end{aligned} \tag{2.1}$$

Thus, the convergence time is n which makes the second term smaller than ϵ. Notice that convergence time depends on the time-step, n as well as the property of the given stationary process.

It is important to stress the fact that ML estimation of the probability matrix of the SR is completely independent from the PM policy adopted for the SP. In other words, estimation of the unknown parameter does not interfere with control. This *identifiability* condition is sufficient to guarantee that the basic EC adaptive control is self-tuning [48].

2.2.2 Decision Policy

As mentioned in Section 2.1, the optimal policy π is the sequence of decisions chosen from the optimized decision table according to the system state in every single time

slice. This approach is possible because the transition probability matrix of SR is determined before optimizing the decision table. But in the unknown stationary environment, it is not possible to build the decision table in advance because the transition probability matrix is unknown. For this reason, it is necessary to provide a decision table for the estimated transition probability matrix dynamically. On the other hand, the EC adaptive policy requires a new policy optimization for every new ML estimate for the SR. This is hard to apply in practice because the computation required to optimize the policy is demanding.

In this section, I describe a table look-up method augmented with a linear interpolation technique that relaxes computational requirements without significantly degrading solution quality. For the sake of simplicity, it is assumed that SR has two states, but this method can handle SRs with more than two states.

Look-up Table Construction

The transition probability matrix of a stationary SR can be characterized by user request probability, $R_i \in [0,1]$, $i = 0,1$. If each dimension, R_i is sampled with a finite number of samples, each sampling point in dimension i is denoted as $R_{ij}, j = 0, i, \cdots, NS_i - 1$, where, NS_i is the number of sampling points for dimension i. Based on these sampling points, a look-up table called *policy table* is constructed as shown in Figure 2.4.

Each cell of a policy table corresponds to a SR of which user request probability is (R_{0j}, R_{1k}) ($j = 0, 1, \cdots, NS_0 - 1$ and $k = 0, 1, \cdots, NS_1 - 1$). And each cell of a *policy table* is also a two-dimensional table, which I call a *decision table* (See Section 2.1 and [11]). A *decision table* is a matrix with as many rows as the total system states and as many columns as the command issued by the PM to SP. Each cell of a *policy table* can be indexed as a pair (j,k) and its corresponding request probability pair is (R_{0j}, R_{1k}). For each pair (j,k), a policy optimization is performed to get the corresponding decision table and the obtained decision table is stored to a cell of the policy table with the corresponding index. The overall table is constructed once for all, and its size is $NS_0 \times NS_1 \times$ the size of the table used in [11].

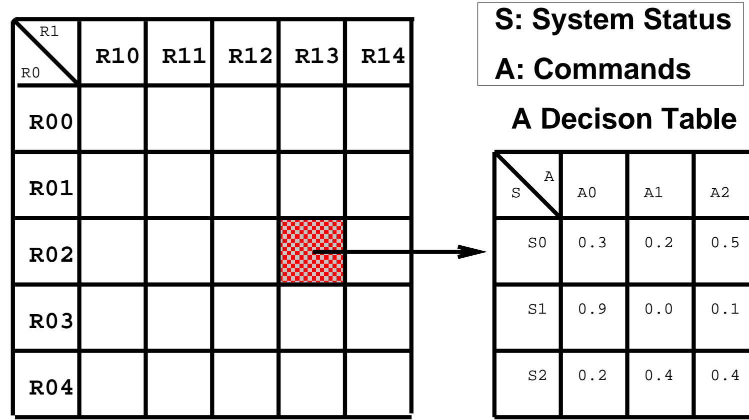


Figure 2.4: An example of a 2D policy table ($NS_0 = NS_1 = 5$)

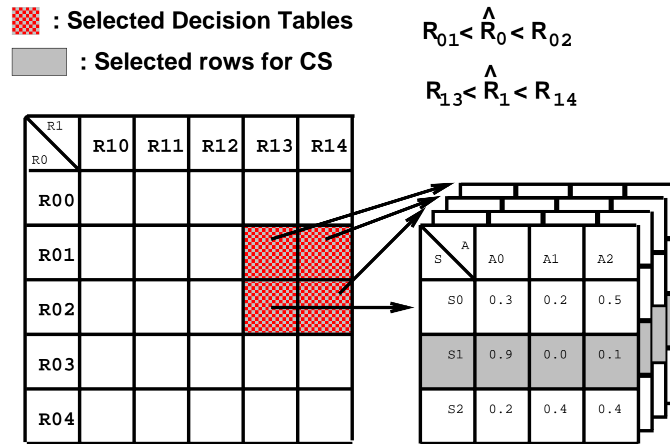


Figure 2.5: Decision Table and Rows Selection from Policy Table

Decision Using Interpolation

For a given time slice, $(\hat{R}_0, \hat{R}_1)$ can be obtained using the estimation technique mentioned in Section 2.1, and two consecutive indices can be chosen for each dimension such that $R_{0j} \leq \hat{R}_0 \leq R_{0(j+1)}$ and $R_{1k} \leq \hat{R}_1 \leq R_{1(k+1)}$. Thus, four decision tables corresponding to the chosen indices can be used to calculate the decision for the given $(\hat{R}_0, \hat{R}_1)$ and current observed system state. From each decision table chosen, a row corresponding to the current system state denoted by CS is selected as shown in Figure 2.5. From these four rows, the final decision row can be obtained

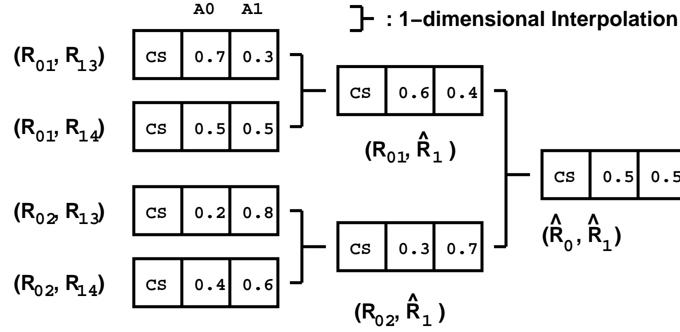


Figure 2.6: Visualized 2-dimensional Interpolation Example

by two-dimensional interpolation technique - applying the one-dimensional interpolation represented in Equation 2.2 iteratively. In a one-dimensional function $f(x)$, the function value $f(x)$ for any point x which is located in between any two points - x_1 and x_2 can be linearly interpolated as follows.

$$f(x) = \frac{(f(x_2) - f(x_1))x + x_2 f(x_1) - x_1 f(x_2)}{x_2 - x_1} \tag{2.2}$$

The iterative procedure is visualized in Figure 2.6 and the pseudo-code of the interpolation/extrapolation procedure including decision table and row selection is shown in Figure 2.7.

Extrapolation is used if $\hat{R}_i > R_{i(NS_i-1)}$ or $\hat{R}_i < R_{i0}$. In all other cases, the interpolated value is computed as three successive one-dimensional linear interpolations on the selected table entries. The proposed technique in this section is described for a SR with two states, but it can be extended to handle SR with more states by increasing the number of dimensions. For n state SR, the required number of dimensions is $n(n-1)$, thus the computational effort is increased proportionally to n^2. In this application, it is enough to model SR with two states with reasonable computation effort.

```
2DInterpolation (CS, cell, $\hat{R}_0$, $\hat{R}_1$, $NS_0$, $NS_1$)
for (i = 0; i < 2; i++) {
  if ($\hat{R}_i \leq R_{i0}$) { /* extrapolation */
    $id_i^0$ = 0;
    $id_i^1$ = 1;
  } else if ($\hat{R}_i \geq \hat{R}_{i(NS_i-1)}$) { /* extrapolation */
    $id_i^0$ = $NS_i$ - 2;
    $id_i^1$ = $NS_i$ - 1;
  } else { /* interpolation */
    $id_i^0$ = j $s.t.$ $R_{ij} \leq \hat{R}_i \leq R_{i(j+1)}$;
    $id_i^1$ = j+1;
  }
}
for (i = 0; i < 2; i++) {
  for (j = 0; j < 2; j++) {
    Select a decision $d_{2i+j}$ from cell($id_0^i$, $id_1^j$)
    for State CS;
  }
}
foreach (command) {
  $d_5$ = OneDimInterp($d_0$, $d_1$);
  $d_6$ = OneDimInterp($d_2$, $d_4$);
  $d_7$ = OneDimInterp($d_5$, $d_6$);
}
return($d_7$);
```

Figure 2.7: 2-dimensional Interpolation

2.3 DPM in Non-Stationary Environment

2.3.1 Non-Stationary Service Requestor

In many practical applications, the assumption of stationary SR does not hold. Workload is subject to changes over time, as intuitively suggested by the observation of typical computer systems. In this section, I describe DPM policies tailored to non-stationary SR models. These adaptive DPM policies are more generally applicable to real system environments than those in Section 2.1 and 2.2, even though they are heuristic, because we cannot claim global optimality in a non-stationary environment.

Our first step is to model a non-stationary SR as shown in Figure 2.8. A non-stationary workload denoted by U^l is modeled by a series of stationary workloads which have different user request probabilities. Each stationary workload is denoted by $u_s, s = 0, 1, \cdots, N_u - 1$, where N_u is the total number of stationary workloads

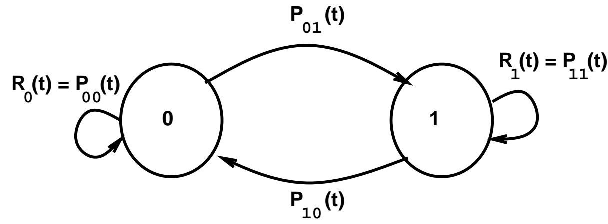


Figure 2.8: An example of a Markov Chain for non-stationary SR

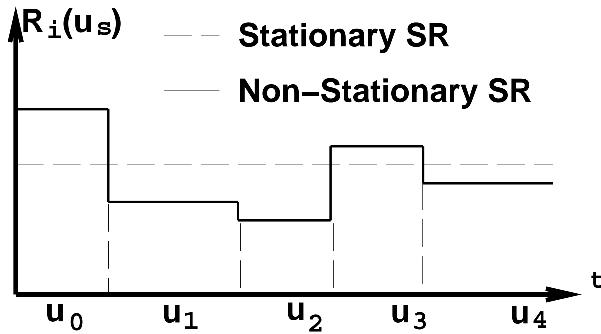


Figure 2.9: $R_i(u_s)$ of non-stationary SR .vs. stationary SR

forming the non-stationary workload, U^l. Thus, a non-stationary workload can be represented as $U^{N_u} = (u_0, u_1, \cdots, u_{N_u-1})$. In this model, the R_i becomes a function of the given sequence and can be distinguished from the R_i of stationary SR as shown in Figure 2.9.

Notice that the non-stationary SR model is very general: by increasing N_u, we can model any given workload with arbitrary accuracy. In fact, for any given sequence of zeros and ones of length λ, we can set $N_u = \lambda$, and define N_u different two-state Markov chains with deterministic transitions that reproduce exactly the given sequence.

Clearly, the knowledge of such a model at time zero is equivalent to assuming the existence of a perfect oracle that can predict the future with no uncertainty. Realistically, we can only expect to be able to predict the future based on past experience, and take into account non-stationarity by limiting the effect that the remote past will have on our current prediction. In other words, I will track changes in transition probability of the non-stationary Markov model by observing the workload

on a limited-size time window in the past.

In the non-stationary environment of Figure 2.9, the optimal policy is to take decision based on the decision table optimized for the R_i for each u_s. I call such an ideal policy the **best-adaptive policy** which requires the perfect knowledge of the change of u_s and cannot be implemented in a real situation. Therefore, the objective in this section is to propose techniques that achieve results comparable to *best-adaptive policy*. The look-up table based interpolation technique introduced in Section 2.2 is still employed for dynamically choosing the most appropriate policy for the estimated SR, but the estimation technique in Section 2.2 should be replaced due to the non-stationarity of the workloads. I propose two window-based approaches to handle the non-stationarity of the workloads.

For the sake of clarity, I enrich our notation: P_{SR} becomes a function of the sequence and denoted by $P_{SR}(u_s)$. From now on, I will denote the actual values as function of u_s because they are constant over time for a given u_s and the estimated values are represented as a function of time. For example, $R_i(u_s)$ is the actual user request probability of a sequence u_s and $\hat{R}_i(t)$ is the estimated user request probability at time t.

2.3.2 Single Window Approach

A sliding window stores the recent user-request history to predict future user requests. This approach is a derivation of MLE (Section 2.2) because it estimates the request ratio depending on recent user history (the information stored in a sliding window) instead of the whole history.

A sliding window denoted as W, consists of l_w slots and each slot, $W(i), i = 0, 1, \cdots l_w - 1$, stores one previous user request, *i.e.* $s_r \in 0, 1, \cdots, S_r - 1$. The basic window operation is to shift one slot constantly for every time slice.

An example of a window operation for a two-state user requests is shown in Figure 2.10. At each time point, $W(i+1) \leftarrow W(i), i = 0, 1, \cdots, l_w - 1$ and $W(0)$ stores a new user request from SR.

At a given time point t, $\hat{P}_{ij}$ in P_{SR} can be simply estimated by the ratio between

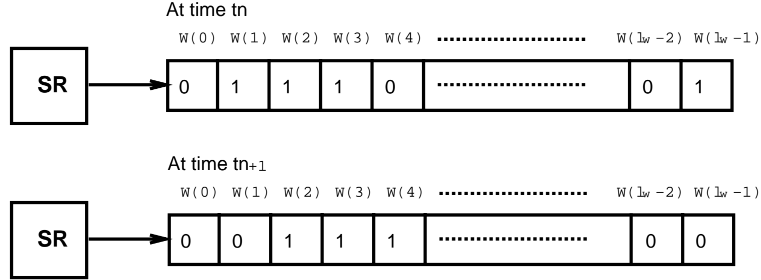


Figure 2.10: Single window operation for two-state user requests

the total number of transitions from state i to j and the total number of occurrences of state i observed within the window.

It may be impossible to define $\hat{P}_{ij}$ when the sliding window does not have any information of state i at a certain time point. In this case, I define $\hat{P}_{ij}$ as 0 when $i = j$ and $1/(S_r - 1)$ when $i \neq j$, respectively.

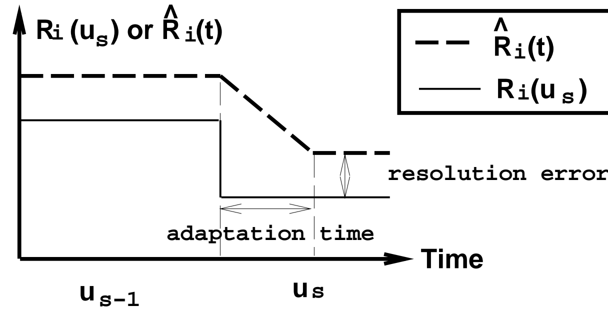
Let us denote the total number of state i observed by the sliding window by A_i, then $A_i = \sum_{k=1}^{l_w-1}(W(k) = i)$ and $\hat{P}_{ij}$ at a given time t can be formally expressed as Equation 2.3.

$$\hat{P}_{ij}(t) = \begin{cases} \frac{1}{A_i}\sum_{k=1}^{l_w-1}[(W(k) = i) \wedge (W(k-1) = j)] & if \;\; A_i \neq 0 \\ 0 & if \; A_i = 0, \; and \; i = j \\ 1/(S_r - 1) & otherwise \end{cases} \tag{2.3}$$

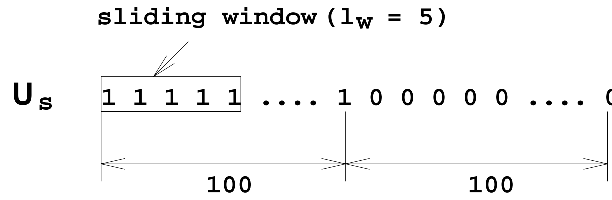
where, "=" is the equivalence operation with a Boolean output, (*i.e.* it yields "1" when the two arguments are same, otherwise returns "0"), and where "∧" is the "*conjunction*" operation.

There exist three possible estimation error sources - resolution error, biased estimation error, and adaptation time.

1. **Resolution error** is due to the maximum precision of $\hat{R}_i(t)$, which is limited to $1/l_w$, For example, if $l_w = 10$, $\hat{R}_i(t)$ cannot express two digit effective numbers such 0.95. The longer l_w is, the smaller the effect of resolution error is.



(a) Resolution error and adaptation time



(b) Biased estimation error

Figure 2.11: Estimation error source

2. **Biased estimation error** happens when l_w is shorter than the burst lengths of sequences. Suppose a SR generates 100 $1's$ after 100 $0's$ and $l_w = 10$. When the sliding window is in the middle of 0 (1) sequence, the window does not have any information of state 1 (0) which causes the estimator to guess the $\hat{R}_1(t)$ ($\hat{R}_0(t)$) arbitrarily (the second or the third case of Equation 2.3). The longer l_w is, the smaller the effect of biased estimation error is.

3. **Adaptation time** is considered when the sliding window is observing u_{s-1} and u_s - the window is experiencing the switching of two stationary processes. The estimation of the new stationary process (u_s) is disturbed by the old stationary process (u_{s-1}). Thus, it is the time required to fill the window, W fully with the transitions of the new sequence u_s. This error source can be reduced by reducing l_w.

These error sources are graphically represented in Figure 2.11. It is obvious that the resolution error is limited by $\frac{1}{l_w}$ and the adaptation time is always l_w independent

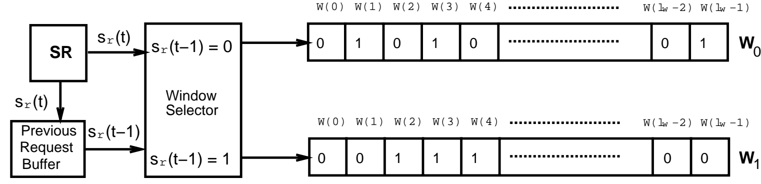


Figure 2.12: Multi-window operation for two-state user requests

to u_s. Also, to avoid the biased estimation error, l_w should be larger than the sum of average sequence length of 0 state and 1 state in case of two-state SR. For example, l_w should be larger than 200 in case of Figure 2.11 (b).

The average burst length of each state i for a given u_s ($I_i(u_s)$) can be expressed as follows.

$$I_i(u_s) = \sum_{k=0}^{\infty} k P_{ii}^k(u_s)(1 - P_{ii}(u_s)) + 1 = \frac{1}{1 - P_{ii}(u_s)} \tag{2.4}$$

This equation represents the average number of self-transitions of each state i whenever state i is first visited. Thus, for two-state SR, the required l_w to avoid biased estimation is simply $I_0(u_s) + I_1(u_s)$ for a given u_s.

Finally, if $l_w \to \infty$, both resolution error and biased estimation error become negligible, but adaptation time becomes infinite. Thus single window approach becomes the MLE of the unknown stationary environment.

2.3.3 Multi-Window Approach

In the single window approach, it is not guaranteed that the previous history observed by the window at a given time point always provides complete state information. Due to this limitation, the second and third case of Equation 2.3 can be frequently used, especially when l_w is small. To avoid this situation, l_w should be increased, but increasing l_w is not desirable because *adaptation time* is also increased. The multi-window approach is devised to overcome this situation by keeping the previous history of each state separately.

The basic structure for the multi-window approach is shown in Figure 2.12. There are as many windows as S_r of SR and their sizes are the same (l_w). For convenience, each window is denoted by W_i which is dedicated to the state $s_r = i$. Therefore, W_i stores only the previous transitions from state i. At a time point t, the Previous Request Buffer (PRB) stores $s_r(t-1)$ and controls the window selector to select a window W_i, where $s_r(t-1) = i$. At each time point t, $W_i(j+1) \leftarrow W_i(j)$, $i = s_r(t-1)$, and $j = 0, 1, \cdots, l_w - 1$. Note that only the selected window W_i, $i = s_r(t-1)$ performs the shift operation, while the other windows stay constant. Thus, each window W_i stores l_w previous user requests and plays a role in predicting the transition probabilities from state i to any other states. Each row of $P_{SR}(u_s)$ is mapped to the window corresponding to the state which is source of the transition and $P_{ij}(u_s)$ can be easily calculated as follows.

$$\hat{P}_{ij}(t) = \frac{\sum_{k=0}^{l_w-1}(W_i(k) = j)}{l_w} \quad \textit{for all } i,\ j \tag{2.5}$$

The estimation error sources of the multi-window approach are resolution error and adaptation time, but there is no biased estimation error because each state has its dedicated window to store past history.

While the resolution error is simply $1/l_w$ (like for the single window approach), the adaptation time is not a constant unlike the single window approach. The adaptation time is determined by the window which is fully filled with the new requests (u_s) in the latest.

Consider a system component power-managed by multiwindow approach. Suppose a stationary SR which can generate either u_i or u_j depending on the initial state of SR, where $u_i = 00000110000011\cdots$ and $u_j = 11000001100000\cdots$. Also, suppose that l_w of each window (W_0 and W_1) is 10. Then, to completely fill W_0 with u_i, we need two repetitions of sequence 0000011, whereas we need five repetitions of sequence 0000011 for W_1. Thus, the adaptation time is $5 \times 7 = 35$ time slices determined by W_1. On the other hand, for u_j, we only need $5 \times 7 - 5 = 30$ time slices because $0's$ in the last repetition is of no use (W_0 is already filled with u_j). Therefore, the average adaptation time for the given SR is 32.5.

For two-state SR, it can be generally represented as follows.

$$\begin{aligned} t_{adapt} = & \lceil \frac{l_w}{m} \rceil (I_0(u_s) + I_1(u_s)) \\ & - \frac{1}{2}(\lceil \frac{l_w}{m} \rceil - \frac{l_w}{m})(I_0(u_s) + I_1(u_s)) \end{aligned} \tag{2.6}$$

where, $m = \min(I_0(u_s), I_1(u_s))$, thus $\lceil \frac{l_w}{m} \rceil$ represents the number of repetitions required to fill the window for the given sequence of which length is $I_0(u_s) + I_1(u_s)$. The last term represents the unnecessary part of the sequence in the last repetition. Finally, the last term is divided by 2 to get the average value with the consideration of different initial states.

2.4 Experimental Results

The effectiveness of the proposed algorithms is validated by the simulation in the context of the system model of Figure 2.1. The key advantage of simulation is flexibility: the executable models of all system components are C routines that can be easily modified to simulate different operating conditions and tune the power management policy.

On the other hand, real-world experiments are necessary to test the actual applicability and effectiveness of the proposed techniques: no approximation is introduced by modeling assumptions, all implementation issues have to be addressed and solved, power savings can be evaluated by means of measurements and performance degradation can be directly experienced by the user instead of being represented by arbitrary cost metrics. The real-world experiments for this method will be discussed with other DPM policies in Chapter 4.

2.4.1 Experimental Setting

The experiments were performed on a Sony VAIO PCG-F150 laptop computer, and on a VA Research VArStation desktop computer. The service providers for our experiments were commercial power-manageable HDDs by Fujitsu (VAIO PCG-F150)

HDD	P_s	P_a	T_{sd}	P_{sd}	T_{wu}	P_{wu}
	Watt	Watt	sec	Watt	sec	Watt
IBM	0.75	3.48	0.51	2.12	6.97	7.53
Fujitsu	0.13	0.95	0.67	0.54	1.61	2.72

Table 2.1: Power states of commercial HDDs from Fujitsu and IBM

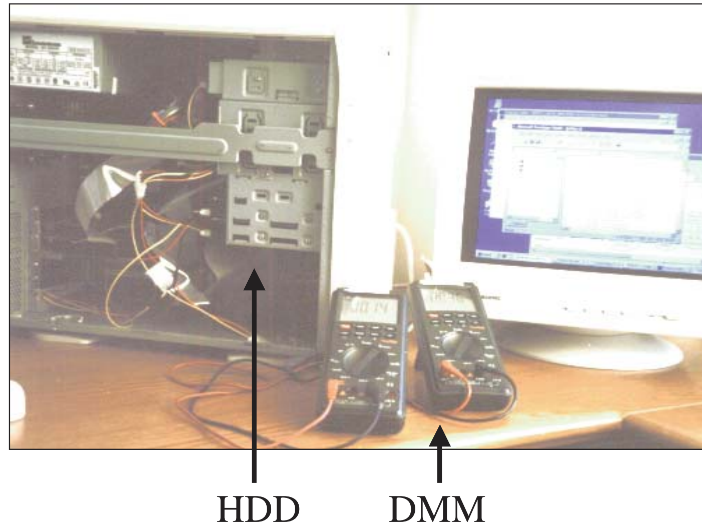


Figure 2.13: Hardware setup for HDD power measurement

and IBM (VArStation). Table 2.1 reports their average power consumption measured in the active and sleep states (P_a and P_s) and during shut-down and wakeup transitions (P_{sd} and P_{wu}). Transition times T_{sd} and T_{wu} are also reported in Table 2.1. The numbers reported in Table 2.1 are obtained from real measurement using the hardware setup shown in Figure 2.13 and its logical diagram is shown in Figure 2.14.

The 12V and 5V power lines go through two digital multi-meters as shown in Figure 2.14 and both meters are connected to data collection computer through the RS232 port. Readers interested in the details of measurement may refer to [63, 65].

I modeled the Fujitsu's HDD of Table 2.1 as a SP with four power states, representing *active*, *sleep*, *wakeup* and *shutdown* operating modes. When active, the SP serves one request per time slice, while it has no throughput when in sleep and

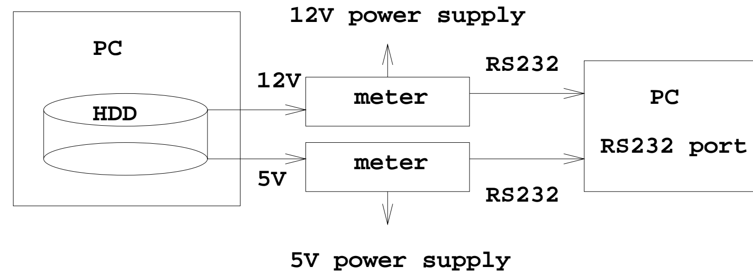


Figure 2.14: The logical diagram of Figure 2.13

transient states. A queue SQ is available to store up to 3 incoming requests when the SP is not ready to serve them. The SR is a two-state Markov chain that issues a request per time slice when in state 1 and no requests when in state 0. The overall system is a Markov chain with 32 states.

According to the actual behavior of the HDD, wakeup transitions are triggered by incoming requests, while shut-down transitions are triggered by a GO_TO_SLEEP command issued by the PM. In practice, the PM controls the SP by issuing two alternative commands, GO_TO_SLEEP and GO_TO_ACTIVE. When the SP is in active state with no incoming and waiting requests, the GO_TO_SLEEP causes a shut-down transition, while GO_TO_ACTIVE leaves the SP in the active state. On the other hand, when the SP is in sleep state with no incoming and waiting requests, GO_TO_SLEEP leaves the SP in the sleep state, while GO_TO_ACTIVE wakes up the SP. In all other conditions (there are incoming or waiting requests or the SP is in either shutdown or wakeup state) the PM has no control on the SP. Though the complete PM policy can be viewed as a 32×2 matrix, there are only two significant rows, corresponding to a state ($SP =$active, $SR = 0$, $SQ = 0$) and the other state ($SP =$sleep, $SR = 0$, $SQ = 0$). For all other states, power manager does not issue any command, which reduces the computation overhead due to power management.

Moreover, the two entries in the row represent complementary probabilities, so that the second one can be obtained as the 1's complement of the first one, representing the conditional probability of issuing a GO_TO_SLEEP command. The value of such probability is the only degree of freedom available for policy optimization. Thus, the memory space required to store the decision table is only a few bytes. Therefore,

total memory required for the policy table is under $1K$ bytes, even when NS_i for each dimension i (number of sampling points for each dimension) is 10. This low memory requirement is especially advantageous when multiple devices must be controlled by our power management policy.

PM policies were designed to minimize power consumption subject to performance constraints expressed in terms of upper bounds on two performance metrics: the average waiting time (hereafter called *waiting time* and denoted by W_p) and the average probability of loosing an incoming request because of a queue-full condition (hereafter called *request loss* and denoted by L_p).[2]

For our experiments I used $L_p = 0.05$, representing the probability of loosing up to the 5% of incoming requests, and $W_p = 1$, representing an average delay of one time slice experienced by each service request. The look-up table (LUT) of PM policies was constructed by keeping the constraints unchanged while varying workload parameters R_0 and R_1 with a 0.05 step. For each (R_0, R_1) pair policy optimization was performed (as described in [11]) and the resulting policy stored in the corresponding entry of the LUT. The entire process took less than 10 minutes on a SUN Sparc2, with a 200MHz clock rate and 520MB of memory.

The estimation and control approach proposed in this chapter is characterized by two sources of error: estimation and interpolation. In the following subsections I report the results of simulations performed to isolate and analyze the effects of estimation and interpolation errors. The overall quality of the estimation and control strategy applied to non-stationary workloads is reported and discussed in the next sections. All simulations were performed using an in-house cycle-accurate stochastic simulator, with 10^6 time steps by default.

2.4.2 Estimation Error

Since the asymptotic convergence of the maximum likelihood estimators is theoretically demonstrated, I need only to evaluate the dynamic properties of the estimators,

[2]The request loss is a model for the incoming requests when the queue is full. In practice, no request is lost in the implementation because a queue full state will trigger alternative mechanism for buffering. Still, the queue full state is undesirable.

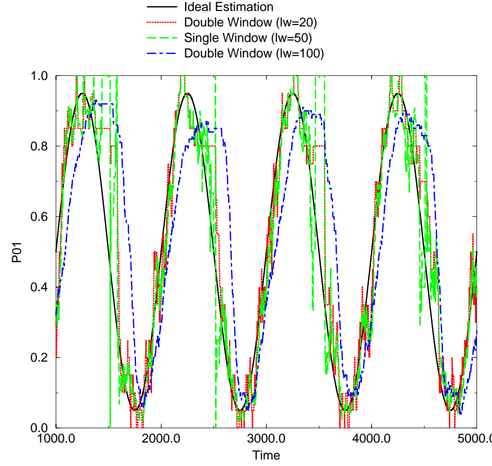


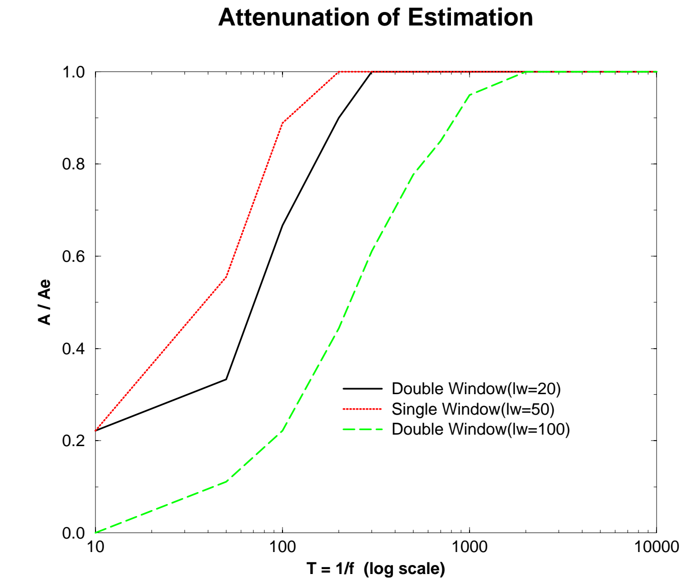
Figure 2.15: Ideal and Estimated curve at $f = 0.001$

i.e., their capability of tracing the time-varying parameters of a non-stationary workload. For this purpose I used a family of non-stationary two-state SRs with self loop probabilities defined as sinusoidal functions of time:

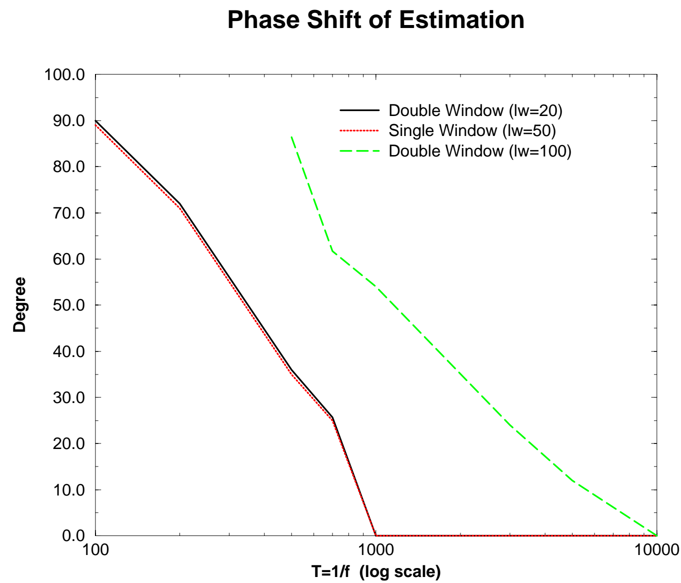
$$R_0(t) = 0.5 + A \times sin(2\pi \frac{t}{T}) \tag{2.7}$$
$$R_1(t) = 0.5 + A \times cos(2\pi \frac{t}{T})$$

where T is the period and A is the amplitude of the variation. Since Equations 2.8 depend only on T and A, I use the notation $SR(T, A)$ to represent sinusoidal SRs.

I simulated $SR(T, A)$ for different values of T and I applied single and double-window estimators to trace the variation of R_0 and R_1. Figure 2.15 shows the actual behavior of $R_0(t)$ for $T = 1000$ and $A = 0.45$, together with the estimates provided by a double window of length 20 (DW20), a single window of length 50 (SW50) and a double window of length 100 (DW100). We can compare the estimated waveforms with the original one in terms of: *attenuation*, that is the ratio between the amplitude of the estimated waveform A_e and that of the original one A, *delay*, evaluated as the time gap between a local maximum of the original waveform and the corresponding maximum of the estimated one, and *noise*, that adds higher-frequency fluctuations



(a) Attenuation



(b) Phase shift

Figure 2.16: Attenuation and phase shift of window-based estimates

to the sinusoidal waveforms. In Figure 2.15, it is apparent that DW100 is less noisy than DW20 and SW50, at the cost of sizable estimation delay and attenuation. It is also worth noting that DW20 is closer to the actual waveform and less noisy than SW50.

The above observations are supported by the results of the analysis in the frequency domain, reported in Figure 2.16. I repeated the simulation experiment of Figure 2.15 for different values of T, and I computed attenuation and delay for each estimator. Estimation delays were divided by $T/2\pi$ and expressed in degrees to obtain comparable *phase shifts*. Attenuation and phase shift were then plotted as functions of T, as shown in Figures 2.16(a) and (b), respectively.

As expected, all window-based estimators act as low-pass filters: as the period T of workload variations decreases, both the attenuation and the phase shift become critical, while they are negligible for values of T larger than a cut-off value that depends on the estimator. In general, the larger the window the higher the cut-off period. This can be better explained by thinking of a non-stationary workload obtained as the concatenation of two stationary ones, u_0 and u_1. When the workload statistics switch from u_0 to u_1, *adaptation time* proportional to the window length l_w (see Section 2.3) is required to completely update the contents of the windows in order to estimate the parameters of u_1 independently of u_0. This effect can be appreciated in Figure 2.16 by comparing the curves associated with DW20 and DW100, while the comparison between DW20 and SW50 suggests a different trend. This counterintuitive result is due to the inherent capability of double-window estimators of selectively keeping trace of significant past events whose distance in time may exceed the window length.

2.4.3 Interpolation Error

To evaluate the effect of discretization and policy interpolation I applied the estimation-and-control strategy (based on a double-window estimator with $l_w = 20$) to a set of 20 stationary workloads with randomly generated values of R_0 and R_1. For each workload I also performed ad-hoc policy optimization using the same constraints used for LUT characterization. Then I run simulations for 50,000 time steps and I compared

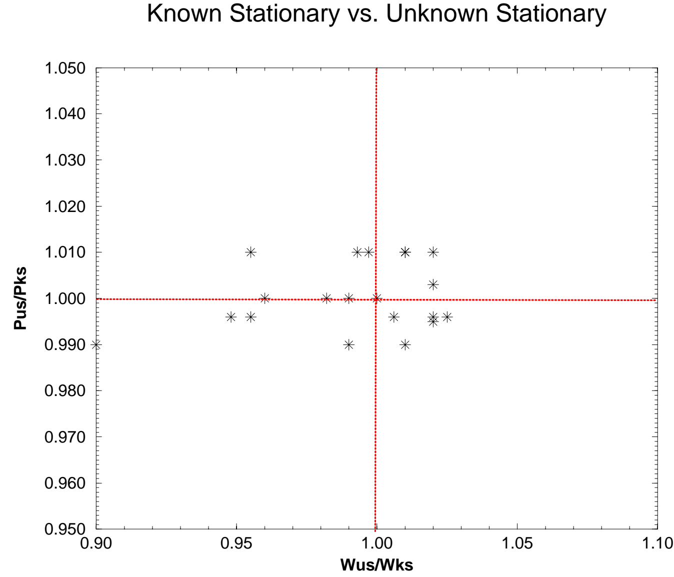


Figure 2.17: Sub-optimality of interpolated policies.

the power-performance trade-off's provided by the adaptive PM with those provided by the optimum ad-hoc policy.

Results are reported in Figure 2.17 as a scatter plot. Each experiment is represented by a point whose coordinates are the ratio between the power consumptions obtained by interpolated and ad-hoc policies (P_{us}/P_{ks}), and the ratio between the corresponding waiting times (W_{us}/W_{ks}).[3] Points close to (1,1) represent situations where the approximation introduced by estimation and interpolation produced negligible effects on the actual quality of the control policy. In other words, the adaptive policy produced almost the same benefit of the optimum one (KS). The average penalty of adaptation was below 5% both in terms of power and performance.

As a final remark, notice that results of Figure 2.17 are affected both by the interpolation error and by the estimation error. However, the estimation error is negligible because the simulation time was much longer than the cut-off time of DW20 (50,000 time steps against a few hundreds).

[3]Subscripts us and ks stay for *unknown stationary* and *known stationary*, respectively.

2.4.4 Overall Quality of Estimation and Control

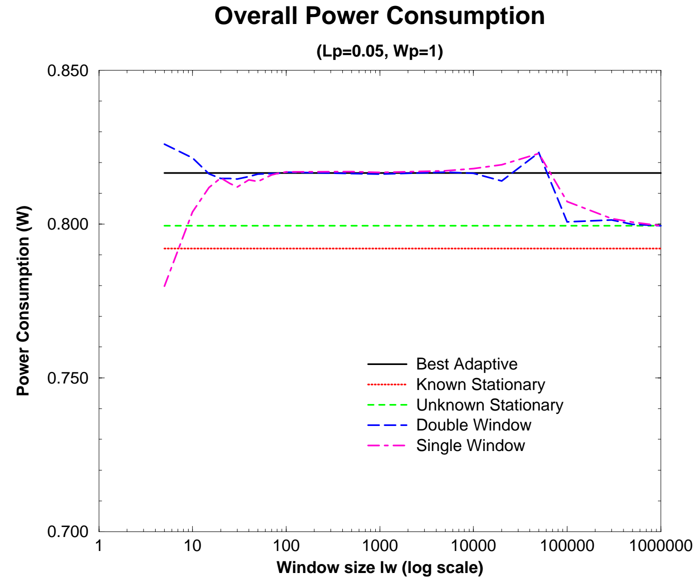
To evaluate the overall quality of the proposed approach we simulated a highly non-stationary workload, built as the concatenation of SR traces of variable lengths (ranging from 40,000 to 60,000 time steps) generated by the 20 stationary workloads of which were used in interpolation error estimation. For the sake of conciseness, in the following I use U^{20} to denote the non-stationary workload trace and u_s $(s = 0, \cdots, 19)$ to denote each of the stationary traces that compose it.

I simulated the effect on U^{20} of adaptive control based on single-window (SW) and double-window (DW) estimators with different window sizes. I also implemented and simulated known stationary (KS), unknown stationary (US), and best-adaptive (BA) mentioned in Section 2.3.

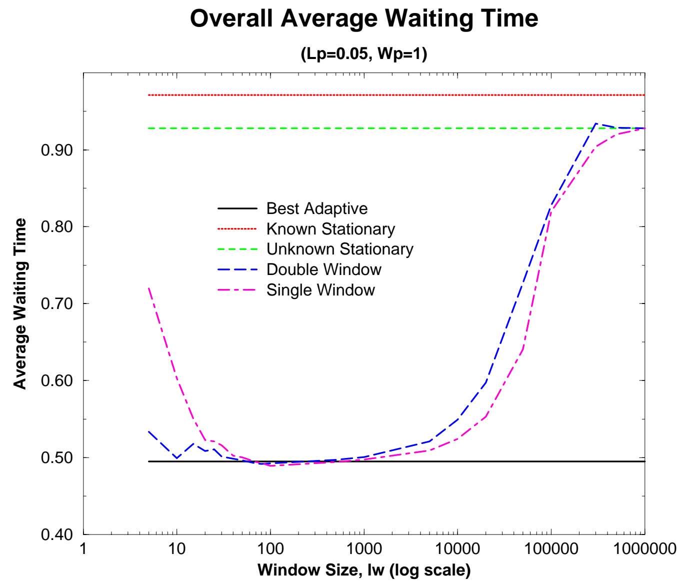
I compared the above five policies to the *best oracle* (BO) policy. BO is the most ideal policy in the sense that it perfectly knows the arrival of future requests deterministically. It deterministically decides to shut down the SP at the beginning of idle periods longer than the break-even time t_{be} (*i.e.*, long enough to compensate the shut-down and wakeup cost) [12]. Also, it wakes up the SP T_{wu} before the next incoming request is issued by SR, thus BO never pays performance penalty for power saving. This policy cannot be implemented in practice, but its effect can be quantified through offline analysis of any workload. Since it is the "best" possible policy, it is useful for comparisons.

The power consumption and waiting time provided by the power management strategies are reported in Figures 2.18(a) and (b) as functions of the sliding window size l_w. For a wide range of values of l_w (from 50 to about 5,000) both SW and DW approaches provide almost the same power-performance trade-off of BA policy. Outside this range, estimation errors cause sizeable violations of performance constraints mainly due to the estimation noise for $l_w < 50$, and to the estimation delay for $l_w > 5000$.

It is also worth to mention that the given performance constraint represents the maximum performance penalty allowed to achieve minimum power consumption. Thus, if increasing performance penalty (but still less than the given constraint) does not help to save more power, the waiting time is kept smaller than the given



(a) Power Consumption



(b) Average Waiting Time

Figure 2.18: Power and Average Waiting Time Comparison for Various Window Size ($L_p = 0.05$, $W_p = 1$)

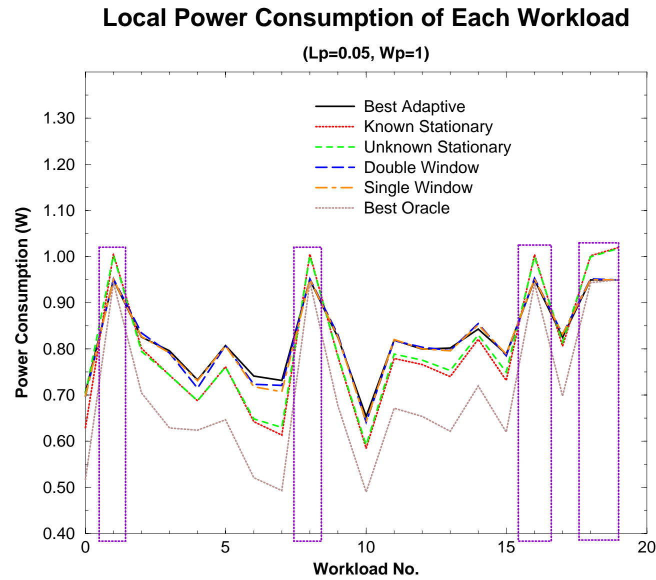
constraint by the control policy. One extreme case can be shown when SR generates requests without idle periods longer than break-even time T_{be}. In this situation, shutdown does not decrease power consumption, but it increases performance penalty. These points will be further explained by means of Figure 2.19.

As for KS and US, though they provide more power savings than BA, they completely violate performance constraints (again, represented by the average waiting time of BA). Their constraint violations are caused by the wrong hypothesis of stationary Markov SR they are based on. It is also worth noting that the estimation errors made by SW and DW approaches when the windows they use are too small or too large are never as critical as those caused by the stationarity assumption.

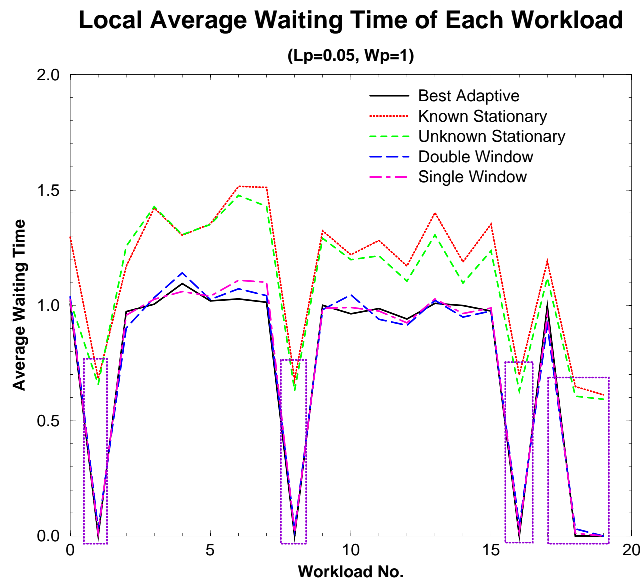
Figures 2.19(a) and (b) show the power and performance values achieved by the PM strategies for each stationary sub-trace u_s in U^{20}. Index s is reported on the x axis. Boxes are used to point out the cases in which the constraint on W_p was inactive either because it was dominated by that on L_p or because there were no idle periods longer than the break-even time. In both cases the actual waiting time was well below the given constraint, causing the overall average reported in Figure 2.18(b) to be around 0.5 instead of 1.

Interestingly, the performance of SW and DW (for $l_w = 50$) is always comparable to that of BA both in terms of power and in terms of waiting time, meaning that both estimation and interpolation errors may be made almost negligible by carefully selecting l_w. On the other hand, the ideal BO strategy often provides a much better (but unreachable) trade-off. As for US and KS, their apparent advantage in terms of power is paid in terms of performance violations that become evident on Figure 2.19. The average waiting time they impose often exceeds by 50% the given constraint.

It is also worth noting that when there are no idle periods longer than the break-even time, all adaptive policies take the correct decision of keeping the resource always on, locally reaching the same quality of BO (see, for instance, $u_s = 8$). In contrast, US and KS policies still issue GO_TO_SLEEP commands that cause both a power waste and a performance penalty.



(a) Local power consumption



(b) Local average waiting time

Figure 2.19: Local power consumption and average waiting time provided by the PM policies for each sub-trace u_s of U^{20} $(L_p = 0.05, W_p = 1)$.

2.5 Chapter Summary

In this chapter, I described how to derive adaptive power management policies for non-stationary workloads based on stochastic approach. The proposed adaptive approach is based on sliding windows and two-dimensional linear interpolation to find an optimal policy from a optimal policy table which is pre-computed. Thus the on-line computational requirements are mild (0.8% of the overall system power consumption). The proposed approach deals effectively with highly non-stationary workloads. Moreover, our adaptive method offers the possibility of trading off power for performance in a controlled fashion. Simulation results show that my method outperforms non-adaptive policies.

As in the case of most current and previous research, I addressed the problem of power managing a single device (e.g., hard disk) abstracted as a single service provider. I believe that this method can be extended to control multiple devices, as long as their number is small. Nevertheless, the problem of performing concurrent power management of multiple devices, under non-stationary workloads, remains a challenging problem for future research.

Chapter 3

Adaptive Learning Tree for DPM

This chapter describes a DPM policy called *adaptive learning tree*. This technique is a predictive approach, while the *sliding window* technique proposed in Chapter 2 is a stochastic approach. The major advantage of *adaptive learning tree* technique is that it can handle multiple-sleep state device which have not been considered in previous predictive DPM policies. Another benefit of this technique is that it provides the self-adaptive capability to adjust the power state control policy to cope with the time-varying (or non-stationary) property of service requests, which is supported by only a few previous predictive DPM policies.

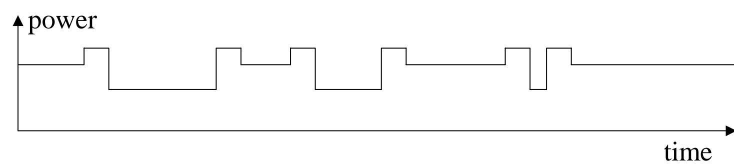
The major concepts of this technique can be summarized in three parts. First, the idle period length is quantized and each quantum is mapped to a power state in which the energy consumption can be maximally reduced. Second, a sequence of the previous idle periods is transformed into a sequence of discrete events using the first concept and this information is recorded in the form of a tree structure. Also, a predictor is assigned to each recorded sequence. Third, whenever a new idle period is observed, one of the sequence recorded in the tree structure is selected and the decision for shutdown is made based on the corresponding predictor. Also, the predictor is updated depending on the result of the shutdown prediction.

3.1 Idle Period Grouping

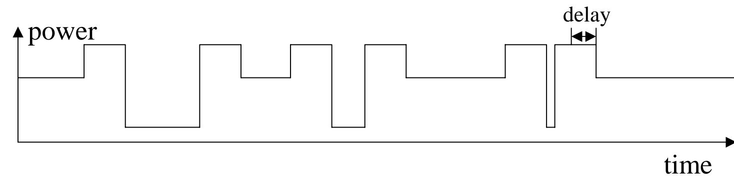
In this section, I introduce the idle period clustering scheme which is the base of *adaptive learning tree* technique for multiple sleep state systems. A system can be abstracted as a two-state finite-state machine as shown in Figure 1.2.

An *idle* period is defined as the period from the time when the system enters the idle state to the time when the system exits the idle state. Similarly, *busy* periods are the time intervals spent in busy state. Thus, the overall system behavior can be modeled as a time series of busy and idle periods. When an idle period is long enough to amortize the shutdown cost, the system can be shut down for power saving. For a system with a single sleep state, such as the one analyzed in [44] we can define a *threshold*, which is the minimum idle time required to reach the break-even point between shutdown cost and power savings. For a multiple sleep state system, we need as many *thresholds* as sleep states because each sleep state has a different shutdown cost. Figure 3.1 illustrates the need for multiple *thresholds* and the efficiency of multiple sleep states compared to the single sleep state when the system workload is known. Usually, shutting down to a deeper sleep state requires more transition time and power consumption (i.e., higher cost). Energy consumption during the idle period can be calculated by estimating the area under the line corresponding to the selected sleep state. In Figure 3.1, the deeper sleep state is more efficient during the first idle period, but the shallower sleep state is more efficient during the second idle period. During the third idle period, by selecting the deeper sleep state, severe delay overhead and less power saving are observed. Finally, during the last idle period, no sleep state is helpful because the idle period is too short. From this example, it is obvious that multiple sleep states and multiple *thresholds* are required for more efficient DPM.

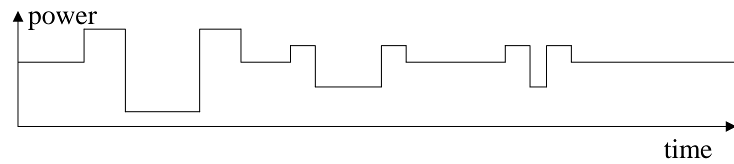
Let n be the number of sleep states; then the total number of power states in the system while it is idle is $n+1$ (*i.e.*, all sleep states plus the *fully on* state). Let $P = \{p_0, p_1, \cdots, p_n\}$ be the set of power states. The n *threshold* values (one for each sleep state) are determined based on the assumption that a deeper sleep state offers lower power consumption at the price of higher transition cost. For a given idle



(a) With a shallower sleep state



(b) With a deeper sleep state



(c) With both sleep states

Figure 3.1: An example of system shutdown with multiple sleep states

period, t_{idle}, energy consumption, E_i by selecting a power state p_i, $i = 0, 1, \cdots, n$, can be computed as follows.

$$E_i = td_i * pd_i + tu_i * pu_i + (t_{idle} - td_i - tu_i) * p_i \tag{3.1}$$

Where, td_i is the transition time from idle state to power state p_i and tu_i is the transition time from power state p_i to idle state. Also, pd_i and pu_i are the power consumption levels corresponding to each transition and p_i is the power consumption while the system is in power state i. In our notation, power state p_i is a shallower sleep state than power state p_{i+1} ($p_i > p_{i+1}$) and power state 0 (p_0) is the idle state in which the system is not shut down. Hence, E_i should be greater than or equal to E_{i+1} and the equality holds when t_{idle} is the *threshold* between power state i and power state $i+1$. We can compute *threshold* values for every $i, i = 0, 1, \cdots, n$ by equating E_i with E_{i-1} and solving for t_{idle}. Let I_i be the *threshold* value between power state p_i and p_{i+1}. Then

$$\begin{aligned} I_i = & \frac{(pd_{i+1} - p_{i+1}) * td_{i+1} + (pu_{i+1} - p_{i+1}) * tu_{i+1}}{p_i - p_{i+1}} \\ & - \frac{(pd_i - p_i) * td_i + (pu_i - p_i) * tu_i}{p_i - p_{i+1}} \end{aligned} \tag{3.2}$$

The time axis can be partitioned in $n+1$ disjoint intervals, bounded by the *thresholds*. We can then associate with a given idle period t_{idle} the index of the power state $IG(t_{idle})$ giving the best savings for that idle period:

$$IG(t_{idle}) = \begin{cases} 0 & if\, t_{idle} < I_0 \\ i+1 & if\, I_i < t_{idle} < I_{i+1} \;\; for \;\; 0 \leq i < n \\ n & if\, I_n < t_{idle} \end{cases} \tag{3.3}$$

Thus, a sequence of idle periods can be transformed into a sequence of integers $0 \leq IG(t_{idle}) \leq n$, which represent the best power state that could be chosen for each idle period. Let s denote the sequence. If s has finite length l, it is denoted as s^l. Also, s_i denotes the ith value of the sequence and s_0 is the most recent event among all s_i's. The optimal power state for an idle period represented by s_i is p_{s_i}.

3.2 Adaptive Learning Tree

Predicting the values of a discrete event sequence is a fundamental problem in learning theory [20]. The idle period clustering technique mentioned in Section 3.1 transforms the sequence of idle periods into the sequence of discrete events. In other words, the problem to be solved is "which value will $IG(t_{idle})$ have in the next idle period for the current sequence s^l?" By predicting the next $IG(t_{idle})$, the system can choose the most appropriate sleep state. In previous studies, learning tree algorithms have been reported to find rules from experience [15, 32, 84, 83]. These algorithms are static in nature, and can be seen as techniques to organize knowledge and drive inference. To be effective, the algorithm must be highly dynamic, and be able to adapt rapidly to changes in the workload.

The learning tree that I propose can be applied to binary as well as multi-valued sequences. Idle periods are observed by the PM and they are transformed into integers $IG(t_{idle})$. This information can be seen as a sequence s^l. The PM predicts the next IG(t_{idle}) for the given s^l based on the current status of the learning tree. The learning tree is updated as soon as the prediction result is available. The sequence s^l is updated by shift operation whenever a new idle period is observed by PM such that $s_i \rightarrow s_{i+1}$ and the new value is stored as s_0. The basic assumption behind the proposed algorithm is that we can predict the future idle periods with high accuracy by observing idle periods in the recent past. The proposed approach has some analogy with advanced branch prediction schemes widely used in computer architectures to reduce the penalty of mispredicted branches [73].

3.2.1 Basic Structure

An example of an adaptive learning tree is shown in Figure 3.2. The proposed adaptive learning tree consists of decision nodes (circles), history branches (solid lines), prediction branches (dashed lines), and leaf nodes (rectangles). The tree is levelized: the top decision node corresponds to s_0, nodes in the second level correspond to s_1 and so on. All leaf nodes are predictions for the next idle period regardless of their ancestor levels. Each leaf stores the Prediction Confidence Level (PCL). The higher

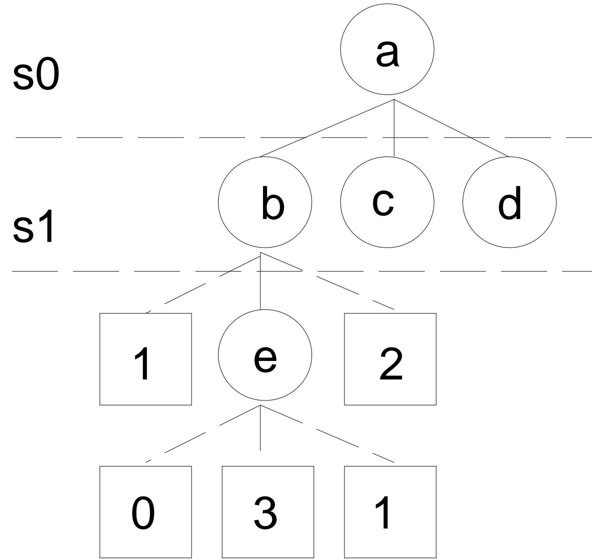


Figure 3.2: An adaptive learning tree (with two sleep states)

the PCL is, the higher the confidence is for a prediction. Each decision node can have both history branches and prediction branches, but the total number of branches is always n, and a prediction branch can only be used when the ancestor is a decision node and the descendant is a leaf node. Each branch of a decision node is associated with the index of a power state $IG(t_{idle}) = \{0, 1, \cdots, n\}$. From left to right, they are denoted as $b_i, i = 0, 1, \cdots, n$ regardless of their types.

3.2.2 Decision

A decision for a given sequence s^l is taken based on a path matching procedure. A *path* for a given sequence s^l is defined as a series of decision nodes such that from the top node, we recursively select a history branch b_{s_i} and move to the lower level decision node connected to b_{s_i}. The recursion is terminated when the b_{s_i} is a prediction branch or the level of the decision node corresponds to s_{l-1}. *Path length (pl)* is defined as the number of decision nodes included in the path. While matching the path, the leaf nodes connected to the decision node included in the path are checked and the leaf node which has the highest PCL is selected. When there are multiple leaf nodes which have the same highest PCL, the leftmost leaf node is selected. After path

matching, the index of the selected leaf node becomes the prediction for the next event. For example, in Figure 3.2, the path, "$a \rightarrow b \rightarrow e$" is matched when the $s^2 =$ "01" and its path length is 2. After path matching, the center leaf node of node e is selected, thus the tree predicts $IG(t_{idle}) = 1$ for the next idle period and issues a command to change the system to power state 1. Also, when the $s^2 =$ "00" or $s^2 =$ "02", the path, "$a \rightarrow b$" is matched. Note that node e is not included in the path for these sequences any more. Thus, the rightmost leaf node of node b is selected in this case. As shown in this example, the number of old events used in decision, pl is varied according to the given sequence. Also, two different sequences ("00" and "02") are classified in the same category and can share the resources of the tree to reduce memory usage.

3.2.3 Learning

In conjunction with prediction, a learning process is needed to maintain the accuracy of the prediction. Whenever an event s_i occurs, the tree is updated to reflect the quality of prediction made when the previous event s_{i+1} occurred. When the prediction is correct, the learning tree should be updated to increase the possibility of choosing the same leaf node for the given sequence. In the opposite case, the reverse action should be performed. This task is achieved by updating the PCL of the leaf nodes. PCL update is controlled by a finite-state machine as shown in Figure 3.3 (the update rule is analogous to that employed in branch prediction buffers for conditional branch prediction). When the prediction is correct, the PCL state is changed to the higher state, in the reverse situation, the PCL state is changed to the lower state. And when it reaches either end state, it keeps the current state. Thus, the PCL is an adaptive feature of the learning tree for non-stationary event sequences. Learning process is more complicated when misprediction occurs, because decreasing PCL of the selected node is insufficient. In other words, the adaptive learning tree has insufficient information to distinguish the given sequence from other sequences and the PCL of the leaf node which should have been selected (desired leaf node) is too low. Thus, two additional procedures are performed. Let us denote the desired value as

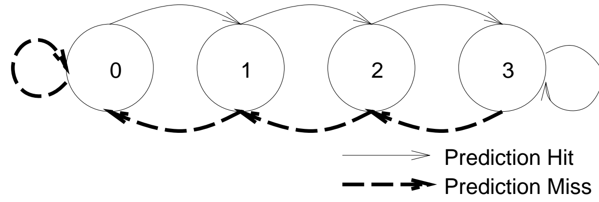


Figure 3.3: PCL operation

$dv = IG(t_{idle})$. First, to increase distinguishability, increase the path length of the current path by replacing the leaf node on the prediction branch $b_{s_{pl+1}}$ connected to the last decision node in the path with a new decision node. Second, to increase the PCL of the desired leaf node, find all leaf nodes which are connected through the prediction branch, b_{dv} on the path and increase their PCL. An example of the updating procedure is shown in Figure 3.4. Suppose there are three different sequences

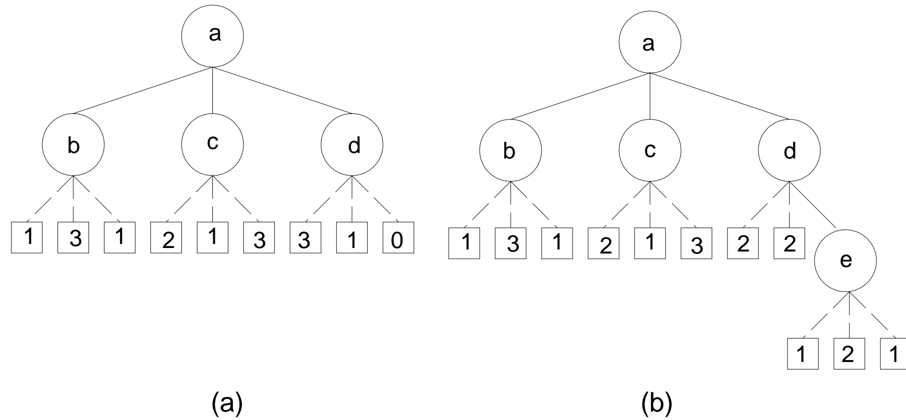


Figure 3.4: An example of learning for a prediction miss

such that A = “20”, B = “21” and C = “22” and the next event after sequence A and B is 0, but the next event after sequence C is 1. The path “$a \rightarrow d$” will be matched for all those sequences in Figure 3.4 (a) and the learning tree will predict 0 for every sequence. This prediction is correct when the given sequence is A or B; it is wrong when the given sequence is C. Thus, it is necessary to distinguish sequence C from A and B. When this prediction miss occurs, the PCL of the leftmost leaf node of node d is decreased. Then, the rightmost leaf node of node d is replaced with

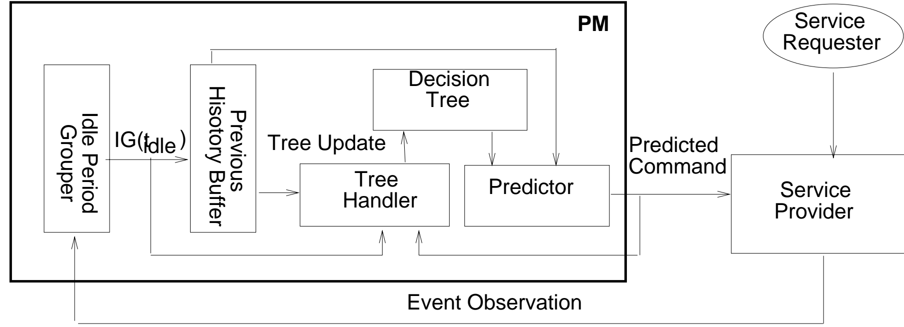


Figure 3.5: Power Manager Configuration

a new decision node because $s_1 = 2$. The leaf nodes of the new decision node have the initial PCL value (in this case, it is 1). Then, the second additional procedure is applied and the final PCL of leaf nodes are as shown in Figure 3.4. Due to these additional procedures, the adaptive learning tree grows in an unbalanced manner and this characteristic is efficient for keeping it small and naturally determines the correlation depth between the future event and old history depending on the sequence characteristics.

3.3 Power Manager

As mentioned in Section 1.4, Power Manager (PM) is the heart of DPM. Thus, the adaptive learning tree is implemented within the PM as shown in Figure 3.5. In Figure 3.5, service requester (SR) is the external environment which triggers the system and the service provider (SP) is the system itself which serves the requests from service requester. The idle period grouper (IPG) observes SP and extract idle periods. Then the idle interval for the observed idle period is calculated and it is passed to the previous history buffer (PHB) and the tree handler. The buffer, PHB stores the observed idle sequence s^l. Whenever a new event arrives, it performs shift operation such that $s_i \rightarrow s_{i+1}$ and the new event is stored as s_0. Finally, the tree handler performs learning and the predictor performs the decision process as mentioned in Section 3.2.

Wakeup and miss correction

The service provider SP can be waken up in two different ways. One is when PM detects a new service request. The other is when the SP stays in power state p_i longer than $I_i - tu_i$. The first case occurs when the predicted idle interval is greater or equal to the actual idle interval. And the second case occurs when the predicted idle interval is less than the actual idle interval. The second case is a prediction miss due to a conservative prediction. After the SP is waken up, PM monitors the system until I_n (maximum *threshold*). During this period, if a new service request comes, the SP can serve this request without wakeup penalty, thus the inefficiency in power saving is compensated by eliminating wakeup performance penalty. Otherwise, the PM shuts down the SP to the deepest power state to save more power. This feature enables the exploitation of very long mispredicted idle periods.

Prediction filter

In many applications, the distribution of idle period intervals shows an L-shaped curve as mentioned in [44, 94], which represents that the ratio of very short idle periods is dominant in total idle periods distribution. Thus, the prediction quality for short idle periods can play an important role in deciding overall prediction accuracy. For this reason, we use a fixed timeout policy preceding the actual prediction. In other words, the command predicted by PM is not issued immediately, but the command issue is delayed for a small amount of time (*threshold* of the fixed timeout policy) to filter out very short idle periods. If a request arrives during this waiting period, the predicted command is canceled, thus only the idle periods longer than the *threshold* can be used for shutdown. The *threshold* value used for the fixed timeout policy is the minimum *threshold*, I_0. Usually, I_0 is small, thus the sacrifice to filter out short idle periods is not a big penalty for power saving, but it prevents excessively aggressive shutdown.

IBM HDD in [11]			
State	ΔT	$Power$	$Threshold$
active	NA	$2.5W$	NA
idle (p_0)	NA	$1W$	NA
idleLP (p_1)	$40ms$	$0.8W$	$680ms$
standby (p_2)	$2.2sec$	$0.3W$	$19088ms$
sleep (p_3)	$6sec$	$0.1W$	$95600ms$
Toshiba HDD in [105]			
active	NA	$2.5W$	NA
idle (p_0)	NA	$0.9W$	NA
standby (p_1)	$1sec$	$0.3W$	$10667ms$
sleep (p_2)	$3sec$	$0.1W$	$70000ms$

Table 3.1: HDD specifications

3.4 Experimental Results

I applied the proposed scheme to two different Hard Disk Drives [11, 105] with the real trace data [110]. I chose two different types of disk traces from [110] - one is the trace for swap purpose only disk and the other is the trace for swap and user data disk. Thus, the distributions of idle period length are different. Two different HDD specifications are shown in Table 3.1 with the *threshold* values computed by the equation 3.2. I implemented a simulator to estimate the performance of the proposed algorithm in terms of power consumption, delay overhead, and energy efficiency. The simulator also supports fixed timeout policies, the *best oracle* policy [24], and other predictive policies [44] for validation purpose. Please refer to Chapter 2 for the details of *best oracle* policy.

The size of PHB in *adaptive learning tree* is 20 bits, thus the maximum path length of the adaptive learning tree was constrained to be less than or equal to 20. Since fixed timeout policy and the prediction policy in [44] does not support multiple sleep states, only the deepest sleep state is used for those policies.

The compared policies are: 1) best oracle ($O1$), 2) *adaptive learning tree* without filter ($M1$), 3) *adaptive learning tree* with filter ($M2$), 4) prediction policy in [44] with miss correction ($H1$), 5) $H1$ with pre-wakeup ($H2$), 6) timeout policy with timeout

value = I_0 ($T1$), 7) timeout policy with timeout value = $1sec$ ($T2$), and 8) timeout policy with *threshold* value that is used in $H1$ ($T3$). $O1$ is the reference in comparison because any other shutdown technique cannot outperform $O1$ and $H2$ has the pre-wakeup feature in addition to the features of $H1$. Several quality measures as shown below were obtained from the simulation.

- **Hit ratio(HR):** is defined as the ratio between the number of correct predictions to the number of total predictions. Thus, it is not used for the fixed timeout policies. Also, the hit ratio of the proposed approach can not be directly compared to that of [44] because they have a different number of sleep states (unit: %).
- **Avg. power(AP):** is the average power consumption during SP is in idle state (unit: W).
- **Delay Overhead(DO):** is the ratio between the increased idle time after applying the policy and original idle time(unit: %).
- **Avg. delay / idle period(AD):** is the ratio between total increased idle time and total number of idle periods. It is a good quality measure for instant availability (unit: sec).
- **Energy(EN):** is the total energy consumed during idle periods normalized to the energy consumption by best oracle policy (unit: J).
- **Efficiency(EF):** is the ratio between the normalized energy in $O1$ and that of each policy. It well represents the efficiency of the policy compared to the ideal policy and is good for considering the power saving and performance penalty together.

The simulation results are shown in Table 3.2. First, the effect of filter is shown from $M1$ and $M2$. In $M2$, by filtering out very short idle periods, the hit ratio is increased by about 10% and this is also reflected in efficiency. Second, $M2$ outperforms $H1$ in terms of hit ratio, even though $M2$ is in a harder situation to increase the hit ratio than $H1$ due to the following reasons. First, $M2$ has more choices than

IBM HDD in [11]: Trace data 0 (swap and user data purpose)								
	O1	M1	M2	H1	H2	T1	T2	T3
HR	100	85.6	96.2	97.5	97.5	-	-	-
AP	0.172	0.217	0.194	0.298	0.998	0.247	0.244	0.234
DO	0.0	1.0	1.0	1.8	1.3	6.1	5.9	1.8
AD	0.0	0.227	0.236	0.428	0.294	1.440	1.393	0.413
EN	1.000	1.273	1.136	1.561	5.202	1.527	1.504	1.384
EF	1.000	0.786	0.880	0.641	0.192	0.655	0.667	0.723
IBM HDD in [11]: Trace data 1 (swap only purpose)								
	O1	M1	M2	H1	H2	T1	T2	T3
HR	100	84.5	94.8	60.7	60.7	-	-	-
AP	0.125	0.148	0.132	0.227	1.014	0.132	0.128	0.149
DO	0.0	0.6	0.5	2.2	1.7	1.3	1.1	1.0
AD	0.0	1.525	1.420	5.891	4.678	3.530	3.093	2.741
EN	1.000	1.190	1.067	1.855	8.242	1.069	1.038	1.203
EF	1.000	0.840	0.937	0.539	0.121	0.935	0.963	0.831
AE	1.000	0.813	0.909	0.590	0.157	0.800	0.815	0.777

Toshiba HDD in [105]: Trace data 0 (swap and user data purpose)								
	O1	M1	M2	H1	H2	T1	T2	T3
HR	100	91.0	99.3	97.6	97.6	-	-	-
AP	0.158	0.200	0.186	0.249	0.898	0.205	0.206	0.234
DO	0.0	0.6	0.5	1.0	0.7	2.1	3.0	1.5
AD	0.0	0.135	0.109	0.234	0.159	1.486	1.705	0.360
EN	1.000	1.273	1.183	1.458	5.242	1.325	1.343	1.503
EF	1.000	0.786	0.845	0.686	0.191	0.755	0.744	0.686
Toshiba HDD in [105]: Trace data 1 (swap only purpose)								
	O1	M1	M2	H1	H2	T1	T2	T3
HR	100	87.6	98.0	59.9	59.9	-	-	-
AP	0.118	0.133	0.126	0.192	0.915	0.125	0.121	0.135
DO	0.0	0.3	0.3	1.1	0.9	0.5	0.6	0.5
AD	0.0	0.803	0.685	3.011	2.387	1.403	1.553	1.381
EN	1.000	1.131	1.071	1.645	7.819	1.065	1.032	1.150
EF	1.000	0.884	0.934	0.607	0.128	0.939	0.970	0.870
AE	1.000	0.835	0.890	0.647	0.160	0.847	0.857	0.778

*AE: Average of EF for trace data 0 and EF for trace data 1

Table 3.2: Comparisons of the various policies

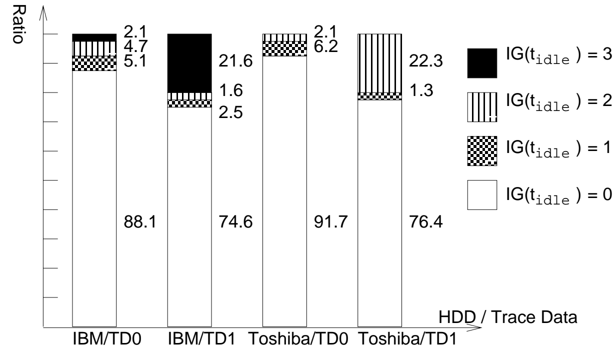


Figure 3.6: Distribution of idle intervals

$H1$ (due to more sleep states). Second, the correction after a miss is considered as a hit in $H1$. Nevertheless, the efficiency of $H1$ is much lower than that of $M2$. It is also observed that the hit ratio of $H1$ is drastically decreased when trace data 1 is simulated. This is an indication that the non-stationary property of trace data 1 is much stronger than that of trace data 0. In contrast, the hit ratio of $M2$ is decreased by about 1%. Thus, the proposed approach adapts well to the variation of SR. $H2$ has very poor efficiency even though the hit ratio is same to that of $H1$. This is because the pre-wakeup scheme wakes up SP even when it meets very long idle periods. Third, in average, $M2$ outperforms any other timeout policy by about $5 - 17\%$. When trace data 1 is applied, policy $T2$ is slightly better than $M2(1\%)$. This fact can be explained by the distribution of idle intervals as shown in Figure 3.6.

For the IBM HDD shown in Table 3.1, the ratio of idle periods in *idle intervals 1 and 2* in trace data 0 is two times more than that in trace data 1. Also, for the Toshiba HDD shown in Table 3.1, the ratio of idle periods in *idle interval 1* is about four times more than that in trace data 1. It means that trace data 1 rarely has intermediate length of idle periods. In other words, the idle periods in trace data 1 are either very short or very long because it is the trace of swap operation only. For this reason, the fixed timeout policy shows good efficiency for trace data 1. But for trace data 0, the efficiency of fixed timeout policy degrades rapidly because the ratio of the intermediate-length idle periods can not be ignored, while the proposed approach shows almost the same efficiency. From these results, we can conclude that

Trace data 0	2nd case		3rd case	
	O1	M2	O1	M2
Hit ratio (%)	100.0	98.1	100.0	99.3
Avg. power (W)	0.183	0.194	0.174	0.213
Delay overhead (%)	0.0	0.9	0.0	0.9
Avg. delay / idle period(s)	0.0	0.206	0.0	0.211
Energy (J)	1.0	1.066	1.0	1.235
Efficiency	0.940	0.882	0.989	0.801
Trace data 1	2nd case		3rd case	
	O1	M2	O1	M2
Hit ratio (%)	100.0	96.5	100.0	97.3
Avg. power (W)	0.125	0.129	0.125	0.138
Delay overhead (%)	0.0	0.5	0.0	0.5
Avg. delay / idle period(s)	0.0	1.450	0.0	1.359
Energy (J)	1.0	1.034	1.0	1.110
Efficiency	1.0	0.967	1.0	0.901
Average Efficiency (IBM)	0.970	0.925	0.995	0.851

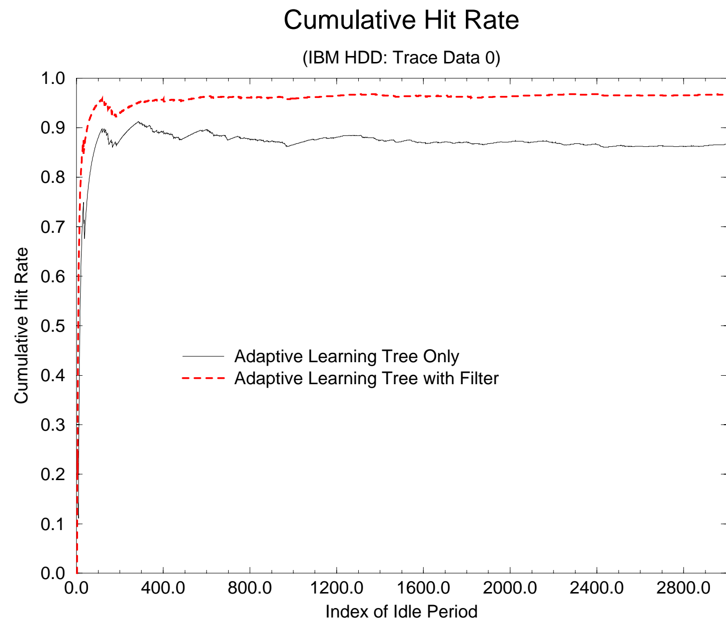
Table 3.3: Comparisons for design guide

the proposed approach has superior reliability.

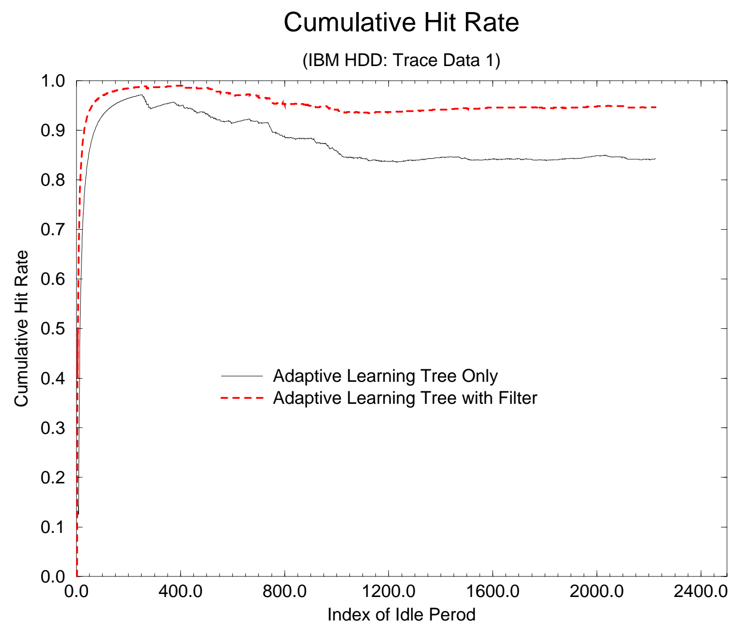
Next, I tested the adaptive speed of the proposed approach and the stability over the time. I used the hit ratio variation over the time for this purpose. As shown in Figure 3.7, the proposed approach achieves a high hit ratio after experiencing less than 1000 idle periods. Moreover, after it reaches a high hit ratio, the variation of hit ratio is very small. The variation in trace data 1 is somewhat larger than trace data 0 because the non-stationary property of trace data 1 is stronger than trace data 0.

Last, I performed another experiment to provide a design guidance in deciding number of sleep states and sleep levels. For this purpose, I simulated three different cases for IBM HDD. The first case is the same case as in the above experiment. In the second case, the standby sleep mode (power state p_2) is eliminated and in third case, the idleLP sleep mode (power state p_1) is eliminated. The simulation results are shown in Table 3.3.

To avoid duplication with Table 3.2, the results of the first case are omitted in Table 3.3. From the comparison of the best oracle policies, it is shown that increasing



(a) Cumulative hit ratio(IBM HDD: trace data 0)



(b) Cumulative hit ratio(IBM HDD: trace data 1)

Figure 3.7: Cumulative hit ratio for IBM HDD

the number of sleep states is ideally more efficient. This is because increasing the number of sleep states enables us to handle various lengths of idle periods. Also, it shows that choosing a deeper intermediate sleep state (standby instead of idleLP) makes it possible to save more power. This situation depends on the distribution of idle intervals. In this experiment, deeper sleep states are preferred, because idle periods in both *idle intervals 1 and 2* have similar ratios in the distribution. Nevertheless, the $M2$ of the second case shows better efficiency than the third case. The reason is that the third case wastes more idle periods when filtering out impulse-like idle periods because its I_0 is much larger than the I_0 of the second case. Even though the hit ratio is increased by a large I_0 (because of perfectly filtering out short idle periods), the increased ratio is only a small amount because the proposed approach already preserves high hit ratio. Thus, to adopt the proposed approach, choosing shallower sleep states or choosing deeper sleep states with small timeout value for filtering is recommended. It is also shown that increasing the number of sleep states is not always the best choice, because it increases the difficulty of the decision process. The results of $M2$ from the first case and the second case supports this argument because their efficiency is almost the same and the hit ratio of the first case is lower than the second case.

3.5 Chapter Summary

In this chapter, I presented a novel adaptive power management policy for non-stationary workloads. This is the first prediction based DPM policy to handle multiple sleep state components. The proposed approach is based on an adaptive learning tree and idle period clustering, and it has been validated through extensive experiments using two different HDD models and two kinds of real disk trace data. The experimental results show that the proposed approach outperforms fixed timeout policy and other prediction methods. Also, it is shown that the prediction accuracy is reliable in the sense that the proposed approach is much less affected by strongly non-stationary workloads. Moreover, the proposed approach reaches reasonable hit ratio before experiencing more than 1000 idle periods.

Chapter 4

Comparison of DPM Policies

This chapter compares the DPM policies proposed in Chapter 2 and Chapter 3 to other DPM policies in real system environments. The comparison was performed for the hard disk drives installed in desktop and laptop computers. Each policy is implemented in Windows2000 running on these computers. The real measurement guarantees a more fair comparison among the DPM policies over the simulation because the simulator of each policy may have different assumptions on hardware modeling. The implementation and experimental data by measurement will be presented in this chapter.

4.1 Compared Policies

The DPM policies compared in this experiment are summarized in Table 4.1.

In Table 4.1, τ represents the timeout value in each fixed timeout policy. As discussed in Chapter 2 and Chapter 3, *best oracle* policy is an ideal policy with offline analysis. It never wastes idle period longer than break-even time and never shuts down the system for the idle period shorter than break-even time. On the other hand, *always-on* policy never shuts down the system regardless of the length of idle period; hence, it cannot reduce the system energy consumption.

All stochastic policies (DW, SW, and US policies) have already been described in Chapter 2 and the *adaptive learning tree* (LT) is described in Chapter 3. Also, the

policy	category	adaptivity	features
BO (*best oracle*)	NA	NA	ideal policy with offline anaysis
DW (double window)	stochastic	yes	double window and interpolation
SW (single window)	stochastic	yes	single window and interpolation
US (stationary)	stochastic	no	optimal for stationary workload
LT (learning tree)	predictive	yes	handling multiple sleep states
EA [44]	predictive	yes	exponential average prediction
CA [46]	timeout	no	$\tau = break - even\ time$
T30	timeout	no	$\tau = 30$
T120	timeout	no	$\tau = 120$
ATO1 [30]	timeout	yes	adjustable τ
ATO2 [62]	timeout	yes	adjustable τ
ATO3 [37]	timeout	yes	adjustable τ
always-on	NA	NA	no shutdown

Table 4.1: Compared policies

details of fixed timeout policies (CA, T30 and T120) can be found in Chapter 1. For this reason, I will discuss only the exponential average policy and adaptive timeout policies in this section.

Exponential Average Policy

In [44], the authors observe that the length of a future idle period can be accurately predicted by the length of the previous idle periods and the prediction of this period. Mathematically, let $t_{predicted}[i]$ and $t_{actual}[i]$ be the predicted and the actual lengths of the ith idle period. The length of the $(i+1)^{th}$ idle period can be approximated by

$$t_{predicted}[i+1] = a \cdot t_{actual}[i] + (1-a) \cdot t_{predicted}[i] \qquad 0 \leq a \leq 1 \qquad (4.1)$$

This is “discounted average” because the effect of the most recent idle period is discounted by factor a while the previous prediction is discounted by $1-a$. Since $t_{predicted}[i]$ is calculated in the same way, we can expand the equation as follows:

$$
\begin{aligned}
t_{predicted}[i+1] &= a \cdot t_{actual}[i] + (1-a) \cdot t_{predicted}[i] \\
&= a \cdot t_{actual}[i] + (1-a) \cdot (a \cdot t_{actual}[i-1] + (1-a) \cdot t_{predicted}[i-1]) \\
&= a \cdot t_{actual}[i] + (1-a)a \cdot t_{actual}[i-1] + (1-a)^2 \cdot t_{predicted}[i-1] \\
&\cdots \\
&= (1-a)^{i+1} t_{predicted}[0] + \sum_{k=0}^{i} a(1-a)^k t_{actual}[i-k]
\end{aligned}
\tag{4.2}
$$

In addtion to Equation 4.2, $t_{predicted}[i+1]$ is restrained such that it cannot exceed $c \cdot t_{idle}[i]$ to avoid inordinate shutdown operations, where c is a constant greater than 1.

If $t_{predicted}$ is larger than the break-even time, the policy shuts down the system. Two parameters, a and c are set to 0.5 and 2 as suggested in [44].

4.1.1 Adaptive Timout Policies

Fixed timout policies cannot adapt to the variation of the workload, thus it may be efficient for some specific workloads, but not for some other workloads. To overcome this limitation, the adaptive timeout policies are introduced. ATO1, ATO2, and ATO3 are adaptive timeout policies which change the timeout value depending on the recent past workload history to reduce the wasted idleness at the beginning of the idle period. ATO1 in [30] considers the length of the previous idle period. If it is short, τ increases; otherwise, τ decreases to follow the variation of idle period length. On the other hand, ATO2 [62] considers the length of a busy period. If a busy period is short, τ decreases; otherwise τ increases. Unlike ATO1 and ATO2, ATO3 [37] updates τ asymmetrically; increasing τ by 1 second or decreasing τ by half a second.

4.2 Policy Implementation

The policies described in Section 4.1 were implemented on both a desktop PC and a laptop PC to control the power state of their hard disk drives. As discussed in

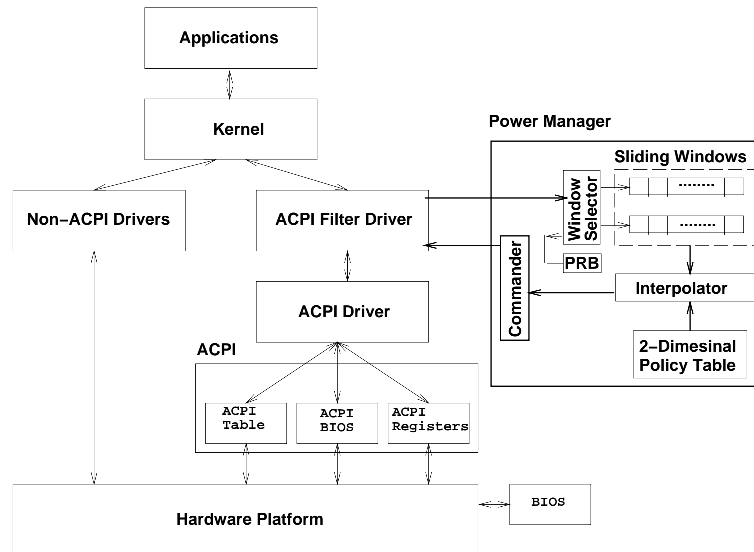


Figure 4.1: ACPI Interface and PC Platform

Chapter 1, DPM policies are implemented in OS level. For this purpose, hardware has to support programming interface and allow the power state ot be changed. There are two widely used standards for power management: *advanced power management*(APM) [4, 2] and *advanced configuration power interface* (ACPI) [1]. The policies were implemented on Windows2000 which supports ACPI.

ACPI specifies protocols between hardware components and operating systems to enable operating system directed power management (OSPM); the OS can adopt system-wide power management policies. Figure 4.1 shows the ACPI interface with the implementation of the double window policy; it consists of the OS which controls the power states, the ACPI interface, and the hardware that responds to ACPI commands and changes power states. An ACPI-compliant device can have up to four power states: `powerDeviceD0 (D0)`, the working state, and `PowerDeviceD1 (D1)` to `PowerDeviceD3 (D3)`, representing three different sleeping states. But only $D0$ and $D3$ states are used in our implementation because the hard disk drives provide only single sleep state. Other policies are also implemented in the same concept.

I used Microsoft Windows2000 in its implementation. In Windows, power management commands are processed as IO commands using *I/O request packets* (IRP).

Special IRP's are required to synchronize ACPI commands so that the transient current does not exceed the maximum capability of the power supply. I implemented power managers using a *filter driver* (FD) template [63]. A filter driver is a device driver attached upon another device driver; it can intercept commands from the OS and responses from the lower driver. The filter driver observes IO activities generated by the OS and applications to update the estimation of stochastic parameters. When the power manager determines to shut down a device, it issues a power IRP to the lower level driver to control the power states.

By implementing power managers in a commercial operating system, we can experiment different algorithms running realistic workloads. Because these filter drivers are visible only to the OS and its lower-layer driver, application programs can run without any modifications. Based on the software-controlled architecture described above, both single and multiple window approaches were implemented and tested.

4.3 Experimental Results

In Chapter 2 and 3, I used simulation to analyze and discuss the inherent properties of the proposed adaptive DPM policies. However, I carried out experiments in a real physical setting to make a fair comparison with other policies in this chapter.

The experiments were performed on a Sony VAIO PCG-F150 laptop computer, and on a VA Research VArStation desktop computer. The service providers for these experiments were commercial power-manageable HDDs by Fujitsu (VAIO PCG-F150) and IBM (VArStation). The hard disk parameters are already presented in Table 2.1.

I implemented single and double-window adaptive control strategies on ACPI-compliant PC's mounting the power-manageable HDDs of Table 2.1: the MHF2043AT HDD (3.8GB) by Fujitsu was installed on a VAIO PCG-F150 laptop computer from Sony (Pentium II, 32MB memory), while the DTTA-350640 HDD (6.44GB) by IBM was installed on a VArStation desktop computer from VA Research (Pentium II, 256MB memory). The base unit of time slice was set to 1 *second* which is large enough to tolerate the computation cost of the power manager.

Alternative DPM algorithms [30, 44, 62, 46, 37, 94] and timeout mechanisms

[64] were implemented on the same platforms for comparison. All PM schemes were applied under the same workload conditions, represented by an 11-hour trace of disk accesses generated by text editors, debuggers and graphical tools running on top of Windows-NT. The quality of each control strategy was evaluated based on 5 metrics:

- P: Average power consumption. (unit: W)
- N_{sd}: Number of shutdowns.
- N_{wd}: Number of wrong shutdowns causing a power overhead.
- T_{ss}: Average sleeping time per shutdown. (unit: sec)
- T_{bs}: Average idle time before shutdown. (unit: sec)

Notice that all metrics shown above are related to only HDD, namely, P is the power consumption of HDD alone. The experimental environment described in Section 4.2 provides the run-time support for online computation of the above metrics.

While average power consumption P provides a direct measure of the objective function of policy optimization, performance metrics are more involved and need some explanations. The number of shutdowns N_{sd} is directly related to the performance penalty that has to be paid (regardless of the PM scheme) to wakeup the SP. The number of wrong shutdown decisions N_{wd} is a measure of inefficiency: a wrong decision causes both a performance and a power penalty. On the contrary, the average sleeping time per shutdown T_{ss} is a measure of efficiency: the longer the sleeping time, the lower the number of shutdowns required to achieve the same power savings. Finally, T_{bs} can be viewed as a measure of inefficiency since it represents wasted idle time. In summary, good PM strategies should be characterized by low values of P, N_{sd}, N_{wd} and T_{bs} and by large values of T_{ss}.

Experimental results are reported in Table 4.2: rows are associated with PM algorithms, columns with power/performance metrics. Algorithms are sorted for increasing power consumption. Notice that the reported power value is only consumed by each target HDD.

It is also worth mentioning that DPM aims at reducing average power consumption of the target device, but it can increase the peak power consumption because

changing power state (especially when the device is waken up) usually requires more power than active state as shown in Table 2.1. The symbols in Table 4.2 to denote each policy is identical to those in Table 4.1.

Performance constraints used for policy optimization were $L_p \leq 0.05$ and $W_p \leq 2$, while window sizes of 50 and 20 were chosen for SW and DW, respectively. As for simulation, BO policy was designed based on the offline analysis of the trace. Rows referring to timeout policies are denoted by the corresponding timeout values ($\tau = 30$ and $\tau = 120$). Finally, the *always-on* row reports the power consumption of the HDD without power management.

Notice that I could not implement a best-adaptive (BA) policy as I did for simulation in Chapter 2, because of the nature of the workload. In fact, best-adaptive policies can only be conceived for piece-wise stationary workloads as those artificially constructed in Section 2.4 for simulation experiments. Real-world workloads are not piece-wise stationary.

The method proposed by Karlin et al. [46] achieved the best power-performance trade-off on the desktop computer, but DW provided comparable results both in terms of performance (N_s) and in terms of power (P), the difference being within 5%. And LT is ranked at fourth and shows large performance penalty compared to the two top policies. A different trade-off (with higher consumption at lower performance penalty) was provided by $\tau = 30$, [62] and [30], while all other approaches showed much worse results.

On the laptop computer, DW provided the lowest power consumption, followed by [46] and LT. I remark, however, that in this case results are not comparable: the performance penalty caused by [46] and LT is almost twice that of DW, with lower power savings.

It is worthwhile to mention that LT (adaptive learning tree) is not as efficient as the simulation results shown in Chapter 3. I believe that this is because the number of sleep states of the target device used for the simulation in Chapter 3 is more than one, while the sleep state of the target device in this measurement is only one. The limited choice (actually, forced choice) of the sleep state in the measurement degrades the efficiency of the adaptive learning tree policy because it intends to support multiple

Policy	P	N_{sd}	N_{wd}	T_{ss}	T_{bs}
BO	0.33	250	0	118	0
DW	0.43	191	28	127	13.4
CA [46]	0.44	323	64	79	5.4
LT	0.46	437	217	56	6.1
ATO1[30]	0.47	273	73	88	12.4
EA [44]	0.50	623	427	37	3.0
TO30	0.51	139	7	157	30.0
SW	0.51	226	83	96.05	20.05
ATO2 [62]	0.52	196	48	109	24.5
US	0.62	173	54	102	35.2
ATO3 [37]	0.64	881	644	19	2.3
TO120	0.67	55	0	255	120.0
always-on	0.95	-	-	-	-

(a) Laptop computer

Policy	P	N_{sd}	N_{wd}	T_{ss}	T_{bs}
BO	1.64	164	0	166	0
CA [46]	1.94	160	15	142	17.6
DW	1.97	168	26	134	18.7
TO30	2.05	147	18	142	30.0
LT	2.07	379	232	62	57
ATO2 [62]	2.09	147	26	138	29.9
ATO1 [30]	2.19	141	37	135	27.6
ATO3 [37]	2.22	595	430	41	4.1
SW	2.25	295	188	68.42	14.25
TO120	2.52	55	3	238	120.0
US	2.60	105	39	130	48.9
EA [44]	2.99	595	503	30	7.6
always-on	3.48	-	-	-	-

(b) Desktop computer

Table 4.2: Algorithm Comparison

l_w	P	N_{sd}	N_{wd}	T_{ss}	T_{bs}
5	1.85	224	61	110.49	6.908
10	1.91	198	45	120.39	10.7
20	1.97	168	26	134	18.7
50	2.23	189	47	104.7	26.9
100	2.35	166	89	100.37	33.68

Table 4.3: Experimental results for window-size sensitivity analysis on desktop PC

sleep state devices.

The performance of SW is worth discussing. Both on desktop and laptop experiments, SW results were much worse than DW, while they provided comparable results on simulation experiments. I believe this is due to the bursty nature of real-world workloads. As discussed in Section 2.3.2, if the SR does not enter a given state for more than l_w cycles, SW provides no information about state transition probabilities from that state. The arbitrary assumptions made in this case by Equation 2.3 about workload parameters may cause sizeable estimation errors. On the contrary, this situation does not impair the performance of DW.

I also ran a set of experiments on the desktop PC to analyze the sensitivity of DW to window size l_w. Results are shown in Table 4.3. Power savings increase monotonically as l_w decreases because of a lower adaptation delay. Performance metrics, on the other hand, show that the minimum penalty is achieved when $l_w = 20$. Since a further reduction of l_w does not reduce power significantly, while impairing estimation accuracy, $l_w = 20$ provides the best trade-off between power and performance.

Finally, I measured the power overhead caused by the policy computation and the power saving of the overall system achieved by our approach. For this purpose, I compared DW and the competitive approach proposed in [46] to always-on policy on the laptop computer. There are two reasons to select the competitive approach: *i)* its power saving for HDD is comparable to our approach, *ii)* its policy computation is very simple, because it is a timeout approach by setting the timeout value to the break-even time, whereas our approach requires more complex computation.

I measured the current drawn by the overall system by connecting the multi-meter

to the AC adaptor. For each policy, I prepared two versions of the power manager. The first version is the same as the power manager used for Table 4.2, but the second version does not issue any command to HDD. In other words, the second version also performs the entire policy computation, but it does not change the power state of the HDD. The first version is useful to measure the impact of each policy on the power saving of the overall system, while the second version can be used to measure the impact of the policy computation overhead on the overall system. Using the second version, I measured the power overhead of the policies for the entire system; it proved to be very small for both approaches (0.8% for DW and 0.6% for the competitive approach).

On the other hand, the measurement of the overall power saving by DW was not obvious due to the power consumption and fluctuation caused by other components such as display and processor. The impact of the power saving for HDD on the entire system can vary significantly depending on the power control policies of other components. To eliminate such variation, I measured the power reduction by DW for the OS controllable fraction [1] of total system power budget. DW achieved 20.2% of power saving, while the competitive approach did 15.7% of power saving (both include the power overhead of the policy computation). Also, the peak power of the system was increased by 6% for both approaches. To summarize, DW outperforms competitive approach in terms of overall power saving, while preserving the same peak power.

4.4 Chapter Summary

In this chapter, I described how to implement DPM policies in real system environment. The policies are implemented in Windows2000 running on both desktop and laptop computers based on ACPI standard and filter drivers.

The efficiency of the implemented policies are compared for the hard disk drives

[1] This subtracts the quiescent power from the total power. The quiescent power is consumed by the laptop computer when no user program is running and HDD is shut down; it was not controlled by DW.

installed in these computers by real measurements. The results show that double window policy outperforms all other compared policies in terms of both power and performance.

Chapter 5

Low Energy Software Optimization

The objective of the work presented in this chapter is to create a framework for the optimization of embedded software application programs. I present algorithms and a tool flow to reduce the computational effort of programs, using value profiling and partial evaluation. Such a reduction translates into both energy savings and average case performance improvement, while preserving a tolerable increase of worst-case performance and code size.

The tool reduces the computational effort by specializing frequently-executed procedures for the most common values of their parameters. The most effective specializations are automatically searched and identified, and the code is transformed through partial evaluation.

Experimental results show that our technique improves both energy consumption and performance of the source code up to more than a factor of two and in average about 35% over the original program. Also, the automatic search engine greatly reduces code optimization time with respect to exhaustive search.

5.1 Basic Idea and Overall Flow

5.1.1 Basic Idea and Problem Description

The technique described in the following sections aims at reducing the computational effort of a given program by specializing it for situations that are commonly encountered during its execution. The ultimate goal of this technique is to improve energy consumption as well as performance by reducing computational effort. The specialized program requires substantially reduced computational effort in the common case, but it still behaves correctly. The "common situations" that trigger program specialization are detected by tracking the values passed to the parameters of procedures. The example in Figure 5.1 illustrates the basic idea.

Consider the first call of procedure `foo` in procedure `main`. Suppose the first parameter `a` is 0 for 90% of its calling frequency. Also, suppose the same condition holds for the last parameter `k`. Using these common values, a partial evaluator can generate the specialized procedure `sp_foo` as shown in Figure 5.1 (b) which reduces the computational effort drastically.

In reality, the values of parameters `a` and `k` are not always 0. Therefore, the procedure call `foo` cannot be completely substituted by the new procedure `sp_foo`. Instead, we can replace it by a conditional statement which selects the appropriate procedure call depending on the result of a *common value detection* (CVD) procedure named `cvd_foo` in Figure 5.1 (b). I call this transformation step *source code alternation*. Also, the variable whose value is often constant(*e.g.* `a`) is called *constant-like argument* (CLA).

When the CVD procedure detects a common case, the specialized code corresponding to the detected common case is executed, which yields fewer instruction executions than the original code. On the other hand, the worst case scenario occurs when the CLA does not take any frequently observed values identified by the profiling. In this case, the worst case performance increase per each call is simply the product of the cost of compare instruction and the number of CLA's tested in the conditional statement, therefore the worst case performance degradation is marginal if the target procedure is computationally expensive.

```
main () {
  int i, a, b, k, m, c[100], d[200], e, result = 0;
  ...............
  result = foo(a, 100, c, k);
  for (i = 0; i < 10; i++) {
    result += foo(i, 100, c, m);
    result += foo(b, e, d, m);
  }
}
int foo(int fa, int fb, int *fc, int fk) {
  int i, sum = 0;
  for (i = 0; i < fb; i++)
    for(j = 0; j < fb/2; j++)
      sum += fa * fc[i] + fk;
  return sum;
}
```

(a) Original program

```
main () {
  int i, a, b, k, m, c[100], d[200], e, result = 0;
  ...............
  if (cvd_foo(a, k)) result = sp_foo(c);
  else result = foo(a, 100, c, k);
  for (i = 0; i < 10; i++) {
    result += foo(i, 100, c, m);
    result += foo(b, e, d, m);
  }
}
int foo(int fa, int fb, int *fc, int fk) {
  int i, sum = 0;
  for (i = 0; i < fb; i++)
    for(j = 0; j < fb/2; j++)
      sum += fa * fc[i] + fk;
  return sum;
}
int sp_foo(int *fc) { return 0; }
int cvd_foo(int a, int k) {
  if (a == 0 && k == 0) return 1;
  return 0;
}
```

(b) Specialized program for the first call of foo (a=0 and k = 0)

```
main () {
  int i, a, b, k, m, c[100], d[200], e, result = 0;
  ...............
  if (cvd_foo(a)) result = sp_foo(c, k);
  else result = foo(a, 100, c, k);
  for (i = 0; i < 10; i++) {
    result += foo(i, 100, c, m);
    result += foo(b, e, d, m);
  }
}
int foo(int fa, int fb, int *fc) {
  int i, sum = 0;
  for (i = 0; i < fb; i++)
    for(j = 0; j < fb/2; j++)
      sum += fa * fc[i] + fk;
  return sum;
}
int sp_foo(int *fc, int fk) { return 50*100*fk; }
int cvd_foo(int a) {
  if (a == 0) return 1;
  return 0;
}
```

(c) specialized program for the first call of foo (a=0)

Figure 5.1: Example of source code transformation using the proposed technique

In general, different possibilities for code optimization exist. This gives rise to a set of search problems that aim to detect the best set of transformations for the example shown in Figure 5.1. If we ignore the common value of `k`, the original code will be specialized as shown in Figure 5.1 (c). The `sp_foo` in Figure 5.1 (c) has one more multiplication than the `sp_foo` in Figure 5.1 (b), but the situation that $a = 0$ will happen more frequently than the situation that both `a` and `k` are 0. For this reason, it is not clear which specialized code is more effective to reduce the overall computational effort. This is the first search problem in our approach.

Next, consider two procedure calls inside the loop of Figure 5.1 with the assumption that parameter `e` (the second parameter of the third procedure call) has single common value, 200. Each of two procedure calls has a CLA as their second argument, respectively. Partial evaluation can be applied for each procedure call to reduce computational effort. However, there is not much to be done by partial evaluator except loop unrolling because all other parameters are not CLAs. The effect of loop unrolling can be either positive or negative depending on the system configuration. For this reason, it is required to find the best combination of loop unrolling for each call. In this example, there are four possible combinations for each call, but the number of combinations is exponential with respect to the number of loops. This is the second search problem of our approach.

After each call is specialized with the best combination of loop unrolling, it is also necessary to check the interplay among the specialized calls, because both specialized calls will increase code size and they may cause cache conflict due to their alternative calling sequence. Thus, we need a method to analyze the global effect of the specialized calls caused by their interplay, which is the third problem of our approach. This paper addresses each of these problems and proposes algorithms for the search of the best code specialization.

To summarize, we have three search problems to specialize a program for common cases.

1. **Common-case selection** is to find the most effective common case among several common cases for each procedure call.

2. **Common-case specialization** is to specialize a procedure call for the given common case by controlling loop unrolling.

3. **Global effective-case selection** is to find the most effective combination of specialized calls.

I will use the term "call site" and "procedure call" interchangeably unless there is an explicit explanation. Also, for the sake of simplicity, we will call cycle-accurate instruction-level simulation (simulator) instruction-level simulation (simulator).

5.1.2 Framework Configuration and Transformation Flow

The automated framework configuration is shown in Figure 5.2, where an instrumentation tool and a profiler provide the basic information necessary to search the solution space. The computational effort estimator solves the common-case selection problem and the specialization engine and global effect analyzer solve the common-case specialization and the global effective-case selection problems, respectively. The entire framework is implemented based on SUIF [100]. CMIX [3] is chosen as a partial evaluator in the specialization engine. ISS (instruction-set level simulator) in both specialization engine and global effect analyzer can be selected depending on the target processor to consider the underlying hardware architecture for the specialization. Each tool component in Figure 5.2 corresponds to each step of the overall transformation flow shown in Figure 5.3. Thus, I will briefly describe each step in this section and the details will be described in the later sections.

- **Instrumentation and profiling.** Two types of profiling are performed - *execution frequency profiling* and *value profiling*. Using the information from *execution frequency profiling*, the computational efforts of procedures and procedure calls are estimated. On the other hand, *value profiling* identifies CLAs and their common values by observing the parameter value changes of procedure calls .

- **Common-case selection.** Based on profiling information, all detected common cases are represented as a hierarchical tree (Section 5.3). To reduce the

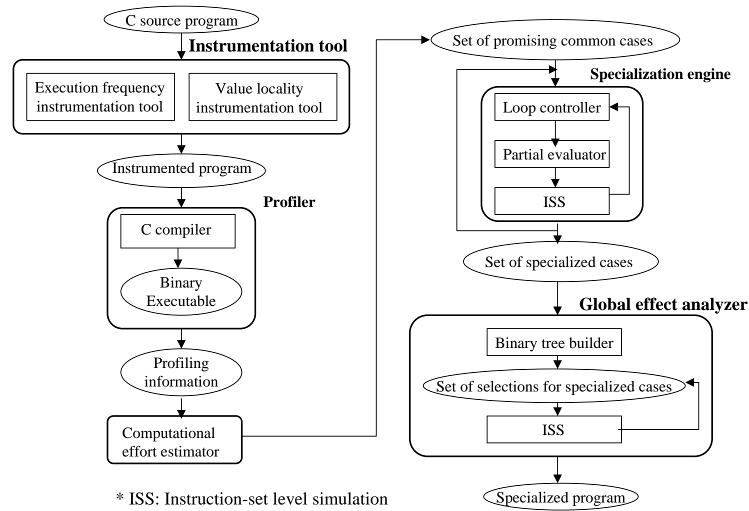


Figure 5.2: The configuration of the proposed framework

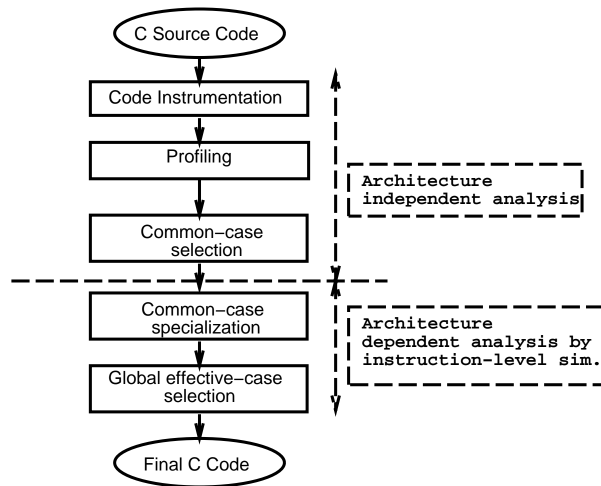


Figure 5.3: Overall source code transformation flow

search space, *normalized computational effort* (NCE) is computed for each object in the hierarchical tree. NCE represents the relative importance of each object in terms of computational effort. By defining a user-defined constraint called *computational threshold* (CT), trivial common cases are pruned.

- **Common-case specialization.** Each case not pruned in the previous step is specialized. In our framework, specialization is performed by CMIX [3] which is a compile-time (off-line) partial evaluator. In addition to the specialized procedure, the *common value detection* (CVD) procedure is generated. Also, source code alternation is performed so that the original procedure call is replaced by a conditional statement as shown in Figure 5.1. For the specialized code of each common case, instruction-level simulation is performed to assess the quality of the specialization and the cases which show improvement by specialization are selected for the next step. The search space of this problem is exponential with respect to the number of loops and the details of heuristic approaches performed by the loop controller for the search space reduction will be described in Section 5.4.

- **Global effective-case selection.** This step analyzes the interplay of the specialized calls chosen at the previous step and decides the specialized calls to be included in the final solution. The search space for this analysis is also exponential with respect to to the number of the specialized calls, thus a search space reduction technique based on the branch and bound algorithm is applied to the binary tree built on the specialized calls.

5.2 Profiling

5.2.1 The Structure of Profiler

Many profiling techniques are based on assembler or binary executable to extract more accurate architecture-dependent information such as memory address tracing and execution time estimation. Since they are designed for specific machine architectures,

they have limited flexibility [77].

In our case, it is sufficient to have only relatively accurate information rather than accurate architecture-dependent profiles, while keeping source-level information. In other words, it is more important to identify which piece of code requires the largest computational effort rather than to know the exact amount of computational efforts required for its execution.

I used the SUIF compiler infrastructure [100] for source code instrumentation. The instrumentation is performed based on the abstract syntax trees (High-SUIF) which well represent the control flow of the given program in high level abstraction. In detail, a program is represented as a graph $G = \{V, E\}$, where node set V is matched to the high level code constructs such as `for-loop`, `if-then-else`, `do-while` and denoted as $v_i \in V$, $i = \{0, 1, \cdots, N_v - 1\}$, where, N_v is the total number of nodes in a program G. Any edge $e_{ij} \in E$ connects two different nodes v_i and v_j and represents their dependency in terms of either their execution order or nested relation. Note that v_i is hierarchical, thus each v_i can have its subtree to represent the nested constructs. For each v_i which is a procedure, I insert as many counters as its descendent nodes to record the visiting frequencies. And for each descendent node, SUIF instructions for incrementing the corresponding counter are inserted for *execution frequency profiling*. *Value profiling* requires additional manipulations such as type checking between formal parameters and actual parameters of procedure calls, recording the observed values and so on.

The proposed profiler has the so-called ATOM-like structure [93] in the sense that the user supplied library is used for instrumentation, namely the source code is instrumented with simple counters and procedure calls. The user supplied library includes the procedures required for both *execution frequency* and *value profiling*. At the final stage, the instrumented source code and the user supplied library are linked to generate the binary executable for profiling.

5.2.2 Computational-Effort Estimation

Computational kernels can be identified by execution frequency profiling and

computational-effort estimation. Execution frequency profiling is a widely used technique to obtain the visiting frequency of each node (v_i in G). This information only represents how frequently each node is visited, but does not show the importance of each node in terms of computational effort.

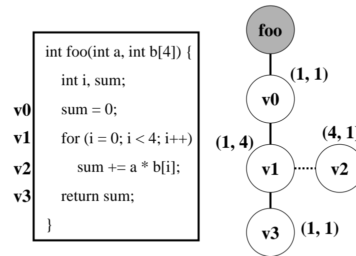
For this reason, I used a simple estimation technique of computational efforts for each basic unit using the number of instructions of each basic unit, where the instruction set used is the built-in instructions defined in SUIF framework. Due to the lack of specification of a target architecture, it is assumed that all the instructions require same computational effort. But I provide a way to distinguish the cost of each instruction when the target architecture is determined using an instruction cost table. Each SUIF instruction is defined with its cost in the instruction cost table, thus the execution time of each node v_i of graph G can be calculated as follows.

$$ce_i = f_i * i_i \sum_{j=0}^{N-1} (o_{ij} * c_j) \tag{5.1}$$

where, ce_i is the estimated computational effort of node v_i, f_i is the execution frequency of node v_i from execution frequency profiling, i_i is the average number of iterations for each visit of node v_i, o_{ij} is the number of instruction j observed in node v_i, c_j is the cost of instruction j, and N is the total number of instructions defined in SUIF. Note that the basic unit of our approach includes for-loop and do-while constructs. For this reason, variable i_i is considered in Equation 5.1. It is also worthwhile to mention that the Equation 5.1 represents the single level computational-effort estimation. As mentioned in Section 5.2.1, the node v_i is hierarchical. Thus, the cumulative computational efforts for each node v_i can be estimated by the sum of current level computational effort and the computational effort of its descendent nodes.

An example of abstract syntax tree is shown in Figure 5.4 (a), where a solid edge represents the dependency of two nodes and a dotted edge represents their nested relation and its corresponding instruction cost table is shown in Figure 5.4 (b). A pair of numbers assigned to each node is (f_i, i_i) which is obtained from the *execution frequency profiling*.

Example: Consider node v_2 in Figure 5.4 (a). f_2 and i_2 are 4 and 1 as shown in



(a) An example of abstract syntax tree

	Instruction type	o	c
v0	load	1	1
v1	compare	1	1
	increment	1	1
v2	add	1	1
	multiply	1	2
	load	2	1
v3	return	1	2

(b) Corresponding instruction cost table

Figure 5.4: An example of abstract syntax tree and instruction cost table

the graph. Also, from Figure 5.4 (b), v_2 has 1 addition, 1 multiplication, and 2 load instructions. Among them, only multiplication has cost twice higher than the other two instruction. By substituting these values into Equation 5.1, $ce_2 = 4 \times 1 \times (1 \times 1 + 1 \times 2 + 2 \times 1) = 20$. Similarly, $ce_0 = 1$, $ce_1 = 8$, and $ce_4 = 2$. Therefore the computational effort of procedure `foo` is 31 by summing these values. ■

5.2.3 Value Profiling

As mentioned in Section 5.1.1, value profiling is performed at the procedure level. In other words, each procedure call is profiled, because any single procedure can be called in many different places with different argument values. We chose value profiling instead of value tracing which records the entire history of value observation, because tracing requires huge disk space and accesses.

One of the difficulties in value profiling occurs when the argument size is dynamic. For example, in a program, one-dimensional integer arrays with any size can be passed to an integer type pointer argument whenever the corresponding procedure is called.

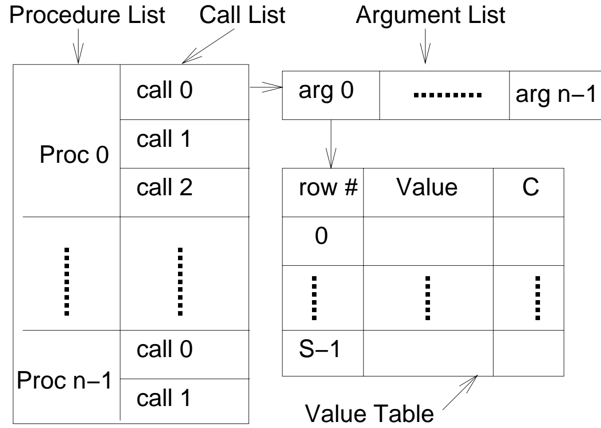


Figure 5.5: Internal data structure of value profiling

Another difficulty occurs when the argument has complex data type because complex data type requires hierarchical traversal for value profiling. For this reason, currently value profiling in our work is restricted to elementary type scalar and array variables. Note that this restriction is not applied to the arguments defined at each procedure, but to the variables passed as arguments for each procedure call. When a procedure call has both types of variables as arguments, only the variables which violate this restriction are excluded from profiling. Pointers to procedures are not considered in our approach due to their dynamic nature.

Figure 5.5 shows the internal data structure of value profiling system. As shown in Figure 5.5, each procedure has a list of procedure calls which are activated inside the procedure. Each procedure call in the list has a list of arguments and each argument in this list satisfies the type constraint mentioned above and has its own fixed size value table to record the values observed and their frequencies. Each row in the value table consists of three fields - index field, value field and count (C) field.

The index field represents not only the index of the row, but also the chronological order of the row in terms of the updated time relative to other rows. Thus, the larger the index is, the more recently the corresponding row is updated. In our representation, each row is denoted as r_i, $i \in \{0, 1, \cdots, S-1\}$, where S denotes the size of value table, *i.e.* the number of the rows in the table. The value field is used to store the observed value, and the c_i field in r_i counts the number of observations of the corresponding value. The table is continuously updated whenever the corresponding

procedure call is executed. At the end of profiling, each argument of the value table is examined to find the values which are frequently observed and only the argument-value pairs which satisfy user defined constraint called *Observed Threshold* (OT) are reported to the user. For this purpose, *Observed Ratio* (OR_i) is calculated for each r_i in the value table as follows.

$$OR_i = c_i/f \tag{5.2}$$

where, f is the visiting frequency of this call site. The larger OR_i is, the more frequently the value is observed. When OR_i is smaller than OT, the value in r_i is disregarded.

The key feature of value profiling is the value table replacement policy [17]. As mentioned above, the size of each value table is fixed to save memory space and table update time. The variable c_i of each value table is initialized to 0. Thus if a new value is observed and at least one of c_i is 0, the new value is recorded in r_i which has the smallest index among these rows. On the other hand, when the table is full (there is no c_i which is 0), the following formula is used to select the row which is to be replaced.

$$rf_i = W * \frac{i}{S} + (1 - W) * OR_i \tag{5.3}$$

where, rf_i, $i \in \{0, 1, \cdots, S-1\}$ is replacement factor which is the metric to decide which row is to be replaced. The smaller rf_i is, the more likely r_i will be selected for replacement. The weighting factor W is used to specify the importance of the chronological order relative to observed count c_i. The selected r_i which has the smallest rf_i is deleted from the table and $r_j \to r_{j-1}$, $j \in \{i+1, \cdots S-1\}$ if $j < S-1$. Finally, the new value is stored into a new row r_{S-1}, and thus the table will contain those S entries for which rf_i is largest.

5.3 Common-Case Selection

As shown in the example of Section 5.1.1, any procedure call with CLAs can be specialized. Some procedure calls can be effectively specialized, while others may not show significant improvement. Also, some CLAs are not useful for specialization. Thus, it is necessary to search the procedure calls which can be effectively specialized by using their common values.

Due to the large search space, I represent all possible common cases as a hierarchical tree based on profiling information and prune out the cases which are expected to show only marginal improvement even after specialization.

5.3.1 Common Case Representation

Figure 5.6 shows the hierarchical tree for the example shown in Figure 5.1 based on the profiling information. Let us consider a simple example how a common case is represented in a hierarchical tree. The program has two procedures: `main` and `foo` (*procedure level*) and procedure `foo` is called three times in procedure main (*call-site level*). The first procedure call has single CLA, `a` which is passed to the formal parameter, `fa` (*CLA level*) and its common value is 0 (*value level*). Finally, the common case is represented as $< 0, -, - >$ by mapping the common value to the corresponding parameter position (*case level*). For the sake of simplicity, I ignore the parameter `fk` which is the fourth parameter of procedure `foo`. I assume that variable `b` (the first parameter of the third call) has two common values - 2 and 3.

In Figure 5.6, the *call-site level* has two-level sub-hierarchies to represent the CLAs and their common values. *CLA level* represents the mapping relation between CLA parameter and its corresponding formal parameter and *value level* is used for common values of CLAs. In *case level*, common values are related to each formal parameter by positional mapping and "-" represents *don't care* - the parameter value in that position is not considered in this case. There are seven possible cases, even though the number of call sites are only three. There is nothing to be examined for procedure `main` because it does not have any CLA.

I introduce some notation for convenience to indicate each level and object in a

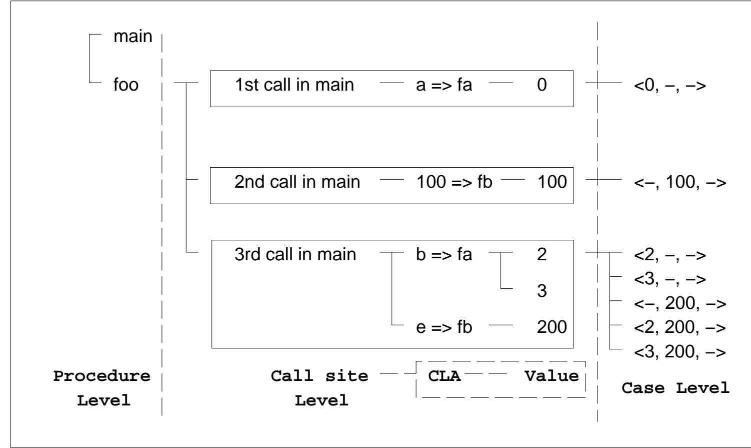


Figure 5.6: Hierarchical tree representation of common cases

level	set	element
procedure	P	p_i
call site	C_i	c_{ij}
CLA	A_{ij}	a_{ijk}
value	V_{ijk}	v_{ijkl}
case	B_{ij}	$b_{ijm} =< cv_0, cv_1, \cdots, cv_k, \cdots, cv_{\lvert A_{ij}\rvert-1} >$

Table 5.1: Notations for a hierarchical tree

hierarchical tree as shown in Table 5.1.

As shown in Table 5.1, *procedure level* is denoted as P which is a set of procedures denoted as p_i. Each procedure p_i has a set of procedure calls, $C_i = \{c_{ij}, i = 0, 1, \cdots, |P| - 1, j = 0, 1, \cdots, |C_i| - 1\}$. And the same rule is applied to *CLA level* and *value level*. Each common case of c_{ij} is denoted as b_{ijm} and each dimension of b_{ijm} (cv_k) corresponds to $a_{ijk}, k = \{0, 1, \cdots, |A_{ij}| - 1\}$, where the bound of m ($|B_{ij}|$) will be shown in Equation 5.5. Also, cv_k of c_{ij} is one of the common values of a_{ijk} or *don't care*, namely $cv_k = \{v_{ijkl}, -\}, k \in \{0, 1, \cdots, |A_{ij}| - 1\}, l \in \{0, 1, \cdots, |V_{ijk}| - 1\}$.

The overall size of the search space to find common cases is calculated by summing the size of search space for each call site. At each call site, we need to examine all possible cases with the consideration of the coherence of the common values (the common value of each CLA may occur at the same time or separately). For example as shown in Figure 5.1, there are four possible cases - *i)* only $a = 0$ (b can have any

p_i			c_{ij}				a_{ijk}			
i	proc	NCE	j	site	freq.	NCE	k	var	value	freq.
0	main	5%	0	-	1	5%	-	-	-	-
1	foo	95%	0	1st	100	8%	0	a	0	100
			1	2nd	10000	29%	1	100	100	10000
			2	3rd	10000	54%	0	b	2	1000
									3	8000
							1	e	200	10000

Table 5.2: Profiling information for the hierarchical tree shown in Figure 5.6

value), *ii)* only $b = 0$ (a can have any value), *iii)* both a and b are 0, *iv)* neither a nor b is 0 (both a and b can have any value). Among these four cases, the last case (case *iv*) is ignored due to the lack of useful information for the specialization and total cases to be examined is three. More generally, the search space of each call site, $|B_{ij}|$ is:

$$|B_{ij}| = \prod_{k=0}^{|A_{ij}|-1} (|V_{ijk}| + 1) - 1 \tag{5.4}$$

where, $|V_{ijk}| + 1$ represents the number of possible values of each CLA (+1 corresp-sonds to any other value except common values) and the last term (-1) represents the case *iv* (none of CLAs has a common value).

The overall size of the search space, S is:

$$S = \sum_{i=0}^{|P|-1} \sum_{j=0}^{|C_i|-1} |B_{ij}| \tag{5.5}$$

5.3.2 Pruning Trivial Cases

Due to the large size of the common case set, it is necessary to reduce the search space without missing promising candidates. I define *common cases* those cases to be included in the search space after search space reduction. The search space reduction is performed based on *normalized computational effort* (NCE). The computational effort of each procedure is obtained from execution frequency profiling and computational-effort estimation technique described in Section 5.2. Based on this, NCE of each

common case can be estimated in a hierarchical order. In other words, NCE of each procedure is estimated first and then NCE of each call site is calculated and so forth.

NCE in a hierarchical tree represents the maximum degree of improvement to be obtained by specializing all cases belonging to the given node. For pruning purpose, a user constraint called *computational threshold* (CT) is defined in terms of NCE. I will assume $CT = 0.1$ for all examples illustrated in this section.

Usually, maximizing the usage of common values is considered to be better because more information is provided to the optimizer. But in our case, maximizing the usage of common values is not always advantageous (*e.g.* the third call in Figure 5.1).

Example: Consider two common cases $< 2, 200, - >$ and $< -, 200, - >$ for the third call of procedure `foo`. The profiling information is shown in Table 5.2 which is a sample profiling information used for all examples in this section. From Table 5.2, $b = 2$ with the probability of 0.1 and $e = 200$ with the probability of 1.0. Then, the probability that case $< 2, 200, - >$ will happen is 0.1, while that of case $< -, 200, - >$ is 1.0. Thus, the specialized code for case $< 2, 200, - >$ is useful only when it reduces the computational effort 10 times more than the specialized code for case $< -, 200, - >$. The cases like case $< 2, 200, - >$ is pruned out before progressing to the next step, *i.e. common-case specialization*, for the sake of the computation efficiency. ■

Pruning is not limited only to *case level*, but also performed at any other level based on NCE. I will describe NCE computation and pruning at each level in the next subsections.

Procedure Level Pruning

Normalized computational effort (NCE) of each procedure is obtained by normalizing its computational effort to the total computational effort. Because NCE of procedure `main` is lower than CT, it is eliminated from the hierarchical tree. Also, the procedure which doesn't have any descendant is eliminated. The pruning at this level has the largest impact on reducing the search space.

Call Site Level Pruning

Call-site level pruning, similar to procedure level pruning, is performed next. The profiler described in Section 5.2 can estimate the computational effort of each procedure as well as each procedure call. Thus, NCE of each procedure call can be computed in the same way as NCE of each procedure is computed. In Table 5.2, the first call of procedure `foo` will be pruned out because its NCE is less than the threshold CT.

I also consider NCE for two sub-hierarchies in *call-site level*. NCE of each CLA is calculated by weighting the NCE of the corresponding procedure call (c_{ij}) by its *observed ratio* (OR_i) and can be represented as Equation 5.6. Also, NCE of each common value (v_{ijkl}) also can be computed similarly.

$$NCE(a_{ijk}) = NCE(c_{ij}) * \sum_{k=0}^{|A_{ij}-1|} OR_k \tag{5.6}$$

Example: Let us consider the third call of procedure `foo`, where a_{120} is variable `b` as shown in Table 5.2. a_{120} has two common values - 2 (v_{1200}) and 3(v_{1201}). Also, from Equation 5.2, $OR(v_{1200}) = 1000/10000 = 0.1$ and $OR(v_{1201}) = 8000/10000 = 0.8$. Thus, $NCE(a_{120}) = 0.54 \times (0.1 + 0.8) = 0.486$ which is larger than CT, thus, a_{120} is not pruned at *CLA level*. At *value level*, $NCE(v_{1200}) = NCE(a_{120}) \times OR(v_{1200}) = 0.486 \times 0.1 = 0.0486$ which is smaller than CT and $v_{1200}$is pruned out, whereas v_{1201} is not eliminated because its NCE is larger CT. ■

Case Level Pruning

Normalized computational effort (NCE) of each case can be calculated using NCE of common values. But NCE at this level cannot be obtained in the same way used in other levels because each case may depend on multiple common values such as case $< 2, 200, - >$. Thus, NCE of each case is obtained by multiplying NCE of common values which are involved in forming the case and represented as Equation 5.7.

$$NCE(b_{ijm}) = \prod_{k=0}^{|B_{ij}|-1} NCE(cv_k) \tag{5.7}$$

Remember that cv_k is $v_{ijkl}, l \in \{0, 1, \cdots, |V_{ijk}| - 1\}$ or "-" and NCE(-) is defined as 1.

Example: Let us consider case $b_{120} =< 2, 200, - >$ which is a child of c_{12} (third call in `main`) and c_{12} is also a child of p_1 (procedure `foo`). From the example in Section 5.3.2, $NCE(v_{1200}) = 0.0486$. Similarly, $NCE(v_{1210}) = 0.54 \times 10000/10000 = 0.54$. From Equation 5.7, $NCE(b_{120}) = 0.0486 \times 0.54 = 0.027$, thus b_{120} is dropped from the search space. But this pruning does not happen in practice because v_{1200} is already pruned out at *value level*. Also, notice that case $< -, 200, - >$ which has less information than case $< 2, 200, - >$ (from the viewpoint of a specializer in the next step) is still in the tree due to its high NCE (0.54). ■

To reduce the search space further, I define *dominated cases* those that can be eliminated from the search space. I say that b_{ijm} is dominated by b_{ijt} if all common values of b_{ijm} appear in b_{ijt} and $NCE(b_{ijt})$ is greater than or equal to $NCE(b_{ijm})$.

$$\begin{aligned} NCE(b_{ijm}) &\leq NCE(b_{ijt}) \\ \forall \; cv_k \; in \; b_{ijm} &\in cv_k \; in \; b_{ijt} \end{aligned} \tag{5.8}$$

where, $a \in b$ is defined as *true* when $a = b$ or $a = -$. For example, b_{121} is dominated by b_{124}. A dominated case needs not to be specialized because it has less information and is less important in terms of NCE than dominant case.

To summarize, pruning is performed at each level, but higher level pruning is more effective because its all descendants are removed. Also, notice that pruning sacrifices the amount of the information useful in the specialization step by increasing the possibility that the common situation occurs more frequently (*e.g.* case $< 2, 200, - >$ is pruned, but case $< -, 200, - >$ is not). This trade-off is controlled by pruning based on the metric - NCE.

5.4 Common-Case Specialization

5.4.1 Overview

After having pruned out trivial common cases (which show marginal improvement, even when they are specialized), we have only common cases (expected to show non-marginal improvement by specialization) left in the hierarchical tree. For each remaining case in the hierarchical tree, we perform the specialization using partial evaluation. The common values of each case are used by partial evaluator for - *(i)* simplifying control flow (pre-computing *if* test or unrolling loops), *(ii)* constant folding and propagation, *(iii)* pre-computing well-known functions calls such as trigonometric functions and so on. These optimizations are not performed independently. Indeed, applying one optimization technique can provide a better chance to other techniques to succeed. For example, loop unrolling can provide better chance to constant propagation/folding by simplifying control dependency and enlarging basic blocks.

Due to such combined effects, it is not easy to estimate the quality of the specialized code analytically. For this reason, this step uses instruction-set level simulator for the purpose of code quality assessment with the consideration of the underlying hardware architecture. It differs from the common-case selection step which performs architecture-independent analysis. Thus, this step takes much longer time than effective case selection step due to specialization and instruction-set level simulation.

Among the techniques mentioned above, loop unrolling should be used most carefully because its side effect (code size increase) can severely degrade both performance and energy consumption. But in traditional applications of partial evaluation, this fact is not deeply studied, based on the assumption that taking more space will reduce computational effort [27]. This assumption may be true for general systems such as workstations, but may not be true for the resource limited systems such as embedded systems. Therefore, we need to address our second search problem by exploring various loop combinations for unrolling. The size of search space for each case specialization is simply 2^n, where n is the number of loops inside procedure p_i.

In case of exhaustive search, the specialization of each case is iteratively performed for the overall search space and each iteration requires instruction-set level

simulation to assess the specialized code quality. In our framework, loop unrolling can be suppressed by declaring the corresponding loop index variable as a residual variable. It means that the residual variable will not be specialized, henceforth the corresponding loop construct will not be affected by specialization either. Because the search space is exponential with respect to the number of loops, two heuristic approaches are proposed in this section. These two approaches may provide lower quality of specialization over the exponential approach, but reduce the search space (both specializations and instruction-set level simulations) drastically.

5.4.2 Semi-exhaustive approach

The first heuristic search algorithm is called semi-exhaustive search. Unlike pure exhaustive search, semi-exhaustive approach performs a complete search for each loop nest rather than for the entire set of loops. Thus, pure exhaustive search guarantees a globally optimal solution, while semi-exhaustive approach can provide a sub-optimal solution. This is the trade-off between the searching time and the code quality. The trade-off will be explained in the experimental part (Section 5.6).

For this purpose, I represent the entire loop structure inside a procedure as a loop tree and an example of loop tree is shown in Figure 5.7. To construct such loop tree, I first levelize the loop structure. The outermost loop is assigned to *level 0* and the next outermost loop is assigned to *level 1* and so on. Next I represent each loop as a node and place each node to its assigned level. Finally, I represent the nested relation between two nodes as an edge connecting these two nodes. Notice that if a loop has multiple loop nests, the connecting edges are identified as a *branch* and I call each branch path *subtree*. For example, the edge between $L0$ and $L4$ and the edge between $L0$ and $L1$ forms a *branch*. Also, there are two subtrees connected to the branch: a subtree formed by $L4$ and $L5$ and a subtree formed by $L1$, $L2$, and $L3$. Each node is represented as $v_i(k)$, where i is the level to which the node belongs and k is the index of the nodes that have the same parent. Thus, if a node is not connected to a branch, k is always 0.

After constructing a loop tree, the best loop combination for unrolling is searched

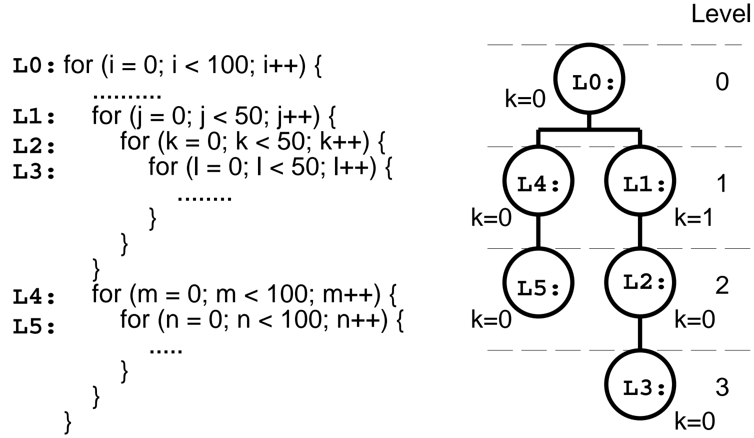


Figure 5.7: An example of loop tree

for each subtree in a bottom up fashion (*i.e.* the *branch* in the lower level is visited first). For a given branch, we visit the subtrees in the order of their computational efforts.

While searching the best solution of each subtree, I exclude the loop combinations which are expected to increase the code size drastically, because such loop combinations increase specialization, compilation, and simulation time drastically. Furthermore, such combinations provide very low quality of specialized code due to the high instruction cache misses. To identify such undesirable cases, I use a code size constraint and a code size estimation technique. The code size constraint is set to the cache size of the target architecture because the code size larger than the cache size will increase the instruction cache miss drastically. Also, the code size is estimated as shown in Equation 5.9.

$$cs_i(k) = \left(\sum_{j=0}^{|K_{i+1}|-1} cs_{i+1}(j) + NI_i(k) \right) * \; I_i(k)/U_i(k) \tag{5.9}$$

where, $cs_i(k)$ the cumulated code size of the descendent nodes of node $v_i(k)$ in addition to the code size of $v_i(k)$ itself. Also, $NI_i(k)$ represents the number of instructions of node $v_i(k)$, $I_i(k)$ represents the average number of iterations per each

visiting of node $v_i(k)$. Finally, $U_i(k)$ returns 1 when node $v_i(k)$ is unrolled, and $I_i(k)$ when $v_i(k)$ is not unrolled. In other words, I estimate the code size to be linearly increased by a factor of $I_i(k)$ when $v_i(k)$ is unrolled. Notice that $I_i(k)$ and $NI_i(k)$ are available from the profiler in Section 5.2.

Example: Consider the loop tree shown in Figure 5.7. Suppose that the subtree on the right *branch* (formed by L1, L2, and L3) has higher computational effort than the subtree on the left *branch* (formed by L4 and L5). In case of pure exhaustive approach, there are 64 (2^6) combinations of loop unrolling, thus the given case should be specialized and simulated 64 times to find the best combination. In case of semi-exhaustive approach, we first visit the right subtree ($L1$) because it has higher computational effort. Because the right subtree is a three-level loop nest(L1, L2, and L3), there are eight combinations of loop unrolling and all combinations are examined to find the best loop combination for the subtree. While examining these eight combinations, the code size of each combination is estimated using Equation 5.9. If the estimated code size is larger than the code size constraint, the combination is excluded from the specialization. After finding the best combination for the right subtree ($L1$), we visit the left subtree ($L4$) which has four possible loop combinations and find the best solution in the same way. After loop unrolling for both subtrees is decided, we move to the top node ($L0$). There are only two combinations for this node because loop unrolling for all its descendent nodes is already decided. Thus, we need to examine total 14 loop combinations using the semi-exhaustive approach. ■

5.4.3 One-shot approach

The second heuristic approach to solve the common case specialization problem is called *one-shot approach*. It is close to *semi-exhaustive approach*, but differs because the choice of the best combination for each subtree depends on just code size estimation instead of exhaustive search within the subtree. The code size estimation is performed in *depth first search* fashion for each subtree. I will illustrate this approach using the following example.

Example: Let us consider the loop tree shown in Figure 5.7. The subtree ($L1$) is visited first due to the same reason in semi-exhaustive approach (higher computational effort). Initially, all nodes are assumed not to be unrolled. However, at this time, all 8 possible combinations are not examined. Instead, unrolled code size is estimated in *depth first order* (from the lowest level ($L3$) to the highest level ($L1$)). First, $L3$ is visited and the unrolled code size is estimated. If the unrolled code size is larger than the code size constraint, the code estimation procedure is terminated and the node is decided not to be unrolled. Also, all nodes in the higher level of this subtree are decided not to be unrolled. Otherwise (estimated code size is smaller than code size constraint), we decide to unroll this node and move up to node $L2$. The same procedure is repeated until it reaches to the top of the subtree. After all nodes in the right graph are traversed, we move to the left graph and the same decision procedure is applied. Finally, we move up to the top node and the same procedure is repeated. ■

To summarize, this approach requires only single specialization and simulation, but it is more limited in improving the quality of partial evaluation.

5.5 Global Effective-Case Selection

The last search problem is to analyze the interplay among the specialized calls to maximize the specialization effect in a global perspective. I already described this problem in Section 5.1 using a simple example in Figure 5.1. I consider now a more complex example.

Example: Consider the situation in Figure 5.8. Suppose that the call of procedure `foo` and both calls of procedure `bar2` inside procedure `bar` are computationally expensive and have common cases. Then, all three call sites are specialized independently in the common-case specialization step. If we analyze their interplay in a local scope (intra-procedural analysis), two calls inside procedure `bar` will interfere with each other marginally. Furthermore, the interplay between procedure call `bar2` and procedure `foo` is not detected because their interplay occurs in inter-procedure level, even though they may affect to each other severely. Thus, the interplay among the

```
main() {
  int *a, b, *c;
  ........
  for (i = 0; i < 100; i++) {
    ........
    if (a[i] > 0)
      foo(a, b);
    ........
    bar(a, c);
  }
}
bar(int *fa, int *fc) {
  bar2(fa, fa[0]);
  ........
  bar2(fc, fc[0]);
}
```

Figure 5.8: A more compex example for global effective-case selection

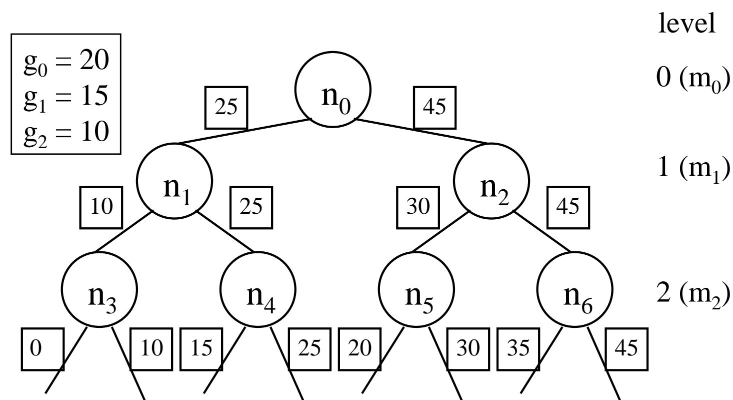


Figure 5.9: An example of binary tree for M

specialized calls should be analyzed in a global scope (inter-procedural analysis). ■
The inter-procedural analysis may reveal that the combination of multiple specialized calls may yield a gain inferior to the sum of the gains of the individual specialized calls, because of mutual interference such as I-cache conflict. Also, it is not obvious to estimate their interference analytically. For this reason, each combination should be assessed by instruction-set level simulation and the best combination is chosen for the final solution.

I represent each specialized call as $m_k \in M$, $k = \{0, 1, \cdots, |M|-1\}$. Each m_k has an attribute called gain, g_k which is the amount of improvement in terms of the given cost metric (either energy consumption or performance) and obtained when each call is specialized at the common-case specialization step. I always sort m_k's in descending order for g_k, *i.e.* $g_k \geq g_{k+1}$. And I denote a combination of the specialized calls as $c_i \in C, i = \{0, 1, \cdots, |C|-1\}$ and $|C| = 2^{|M|}$, thus the search space is exponentially large. Each c_i is a binary vector to represent which specialized calls are included in this combination. For example, $c_0 =< 1, 1, 1 >$ means m_0, m_1, and m_2 are included in the combination c_0. Also, $c_1 =< 1, 1, 0 >$ means only m_0 and m_1 are included in the combination c_1. Each c_i has two gain attributes $ideal_g_i$ and $actual_g_i$ which are *ideal gain* and *actual gain*, respectively.

- **ideal gain** ($ideal_g_i$) is the sum of gains of the individual specialized calls in each combination by assuming that there is no interference with each other. Thus, this is the maximum gain that can be achieved for the given combination.

- **actual gain** ($actual_g_i$) is the sum of gains of specialized calls in each combination with the consideration of their interference. Thus, it is always less than or equal to (when there is no interference) the ideal gain and can be obtained by instruction-set level simulation.

I represent each combination c_i as a path in a binary tree as shown in Figure 5.9. The rightmost path represents $c_0 =< 1, 1, 1 >$ and the second rightmost path represents $c_1 =< 1, 1, 0 >$ and so on. Each level of the tree corresponds to each element of the vector c_i and the right edge and the left edge correspond to "1" and "0", respectively. Thus, the number of levels in the binary tree is always $|M|$. Each edge

```
search_solution (binary_tree) {
  initialze_gain;
  solution = traverse_tree(root_node_of_binary_tree);
  return solution;
}
traverse_tree(binary_tree_node) {
  if (binary_tree_node == NULL) {
    simulate the selected case;
    return sim.  result;
  }
  if (right_node_visited == FALSE) {
    select right_edge;
    gain_right_edge = traverse_tree(right_node);
  }
  if (gain_right_edge >= gain_left_edge) {
    delete left descendent; /* pruning */
    if (solution == NULL)
      save current case to solution;
    return gain_right_edge;
  }
  if(left_node_visited)
    return gain_left_edge;
  else {
    select left_edge;
    gain_left_edge = traverse_tree(left_node);
    if (gain_right_edge >= gain_left_edge) {
      select right_edge;
      save this case to solution
      delete left descendent;
      return gain_right_edge;
    }
    else {
      delete right descendent;
      save current case to solution;
      return gain_left_edge;
    }
  }
}
```

Figure 5.10: Search procedure for the given binary tree

$e_i(l), i = \{0, 1, \cdots, 2^{(l+1)} - 1\}$ and $l = \{0, 1, \cdots, |M| - 1\}$, also has a gain attribute, $g_i(l)$. Where, l is the level to which the edge belongs and i is the index of an edge in level l (from left to right).

Initially, $g_i(|M| - 1)$ (the gain of each edge connected to the leaf nodes) is set to $ideal_g_i$. At the same time, $g_i(l)$ is set to $\max\{g_{2i}(l + 1), g_{2i+1}(l + 1)\}$, namely the edges above than leaf-level inherit the maximum gain of their children. After the gain initialization as shown in Figure 5.9, I perform the search procedure based on *branch and bound* algorithm in Figure 5.10. I will illustrate the how the procedure works using the following example.

Example: The gain for the right edge of n_6 called $g_7(2)$, is initially set to 45 ($ideal_g_0$) because this path corresponds to $c_0 =< 1, 1, 1 >$ which means m_0, m_1, and m_2 are included in the combination c_0. Similarly, $g_6(2)$ (the gain for the left edge of n_6) is set to 35 (corresponds to $c_1 =< 1, 1, 0 >$). Also, $g_3(1) = max\{g_6(2), g_7(2)\} = 45$

and the gains of other edges are also decided in the same way. Next, we apply the procedure in Figure 5.10. First, we visit the rightmost path (c_0). For c_0, we perform instruction-set level simulation to get $actaul_g_0$ and $g_7(2)$ is updated to $actual_g_0$. We compare $g_7(2)$ to $g_6(2)$ which is the maximum gain that can be achieved by combination c_1. If $g_7(2) \geq g_6(2)$, it is obvious that c_0 is better than c_1, thus we eliminate the left edge of n_6 (identical to eliminate c_1). On the other hand, if $g_7(2) < g_6(2)$, c_1 can be better than c_0. Thus, we perform instruction-set level simulation for c_1 and update $g_6(2)$ with $actual_g_1$. Then, we can decide which combination is better and prune out the worse combination. Next, we move to node n_2 in the next level by updating $g_3(1)$ to $max\{g_6(2), g_7(2)\}$ without simulation because we already selected either c_0 or c_1 in level 2. If $g_3(1) \geq g_2(1)$, we can prune out the left descendent of n_2 (c_2 and c_3) due to the same reason. But, if $g_3(1) < g_2(1)$, we visit node n_5 to choose the better combination from c_2 and c_3 by performing the same procedure as we did for c_0 and c_1. After choosing either c_2 or c_3, we compare two edges of node n_2 and select better one. We repeat the same procedure until there remains only one path in the binary tree. ■

To summarize, the algorithm first builds a binary tree to enumerate all possible selections of specialized calls. Second, the expected gain of each path is computed as a cost function for pruning purpose by ignoring the interplay effect. Third, the actual cost of each path is defined as an actual gain considering the interplay effect: this is available from instruction-set level simulation. The purpose of this search problem is to find the path which shows the maximum actual gain among all paths. The pruning occurs when the expected gain of current path is less than the maximum actual gain obtained up to this point which is the bounding function of this search problem. As a final remark, this step can be extended to consider the code size increase constraint by the use of the code size increase estimation mentioned in Section 5.4.

5.6 Experimental Results

5.6.1 Experimental Setting

Even though source code transformations are applicable to a wide set of architectures, I consider now two specific hardware platforms to be able to quantify the results. The Smart Badge, an ARM processor based portable device [89] and ST200 processor developed by STMicroelectronics and Hewlett-Packard [95, 34] were selected as the target architectures. For these target architectures, I applied the proposed technique to seven DSP application C programs - `Compress`, `Expand`, `Edetect`, and `Convolve` from [97], `g721 encode` from [67], and `FFT` from [31], `FIR` [34], and `turbo code` [41], and `SW radio`.

`Compress` compresses a pixel image by a factor of 4:1 while preserving its information content using DCT and `Expand` performs the reverse process using IDCT. `Edetect` detects the edges in a 256 gray-level pixel image using Sobel operators and `Convolve` convolves an image relying on 2D-convolution routine. `g721 encoder` is CCITT ADPCM encoder. `FFT` performs FFT using Duhamel-Hollman method for floating-point type complex numbers (16-point). `turbo code` is iterative (de)coding algorithm of two-dimensional systematic convolutional codes using log-likelihood algebra. Finally, `SW radio` performs a series of operations (CIC lowpass filter, FM demodulation, IIR/FIR deemphasis) for the input in ADC format.

The experiment was conducted for two aspects - search space reduction and quality of the transformed code. The quality of transformed code was analyzed in terms of energy saving, performance improvement, and code size increase. Each application program was profiled to collect computational effort and CLAs with their common values. There exist two important parameters in value profiling as described in Section 5.2.3. First, *Observed Ratio* (OR) is the ratio of the observation frequency of a specific parameter value over the total call site visiting frequency. Second, *Observed Threshold* (OT) is a threshold value to select common values among observed parameter values - only the observed parameter values which shows OR higher than OT is selected as common values. In this experiment, OT was set to 0.5, thus only observed parameter values which have OR higher than 0.5 were selected as common values.

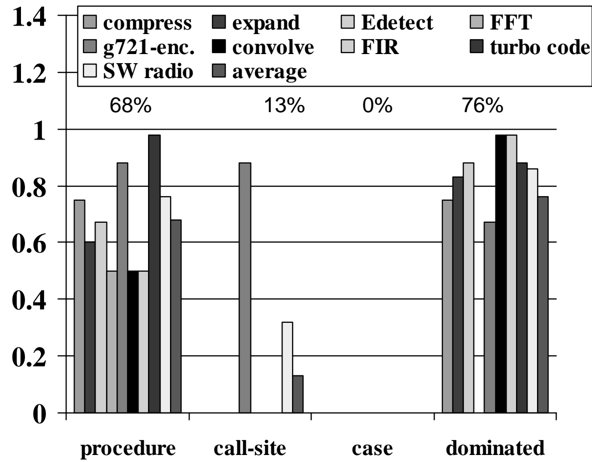


Figure 5.11: Search space reduction using common-case selection

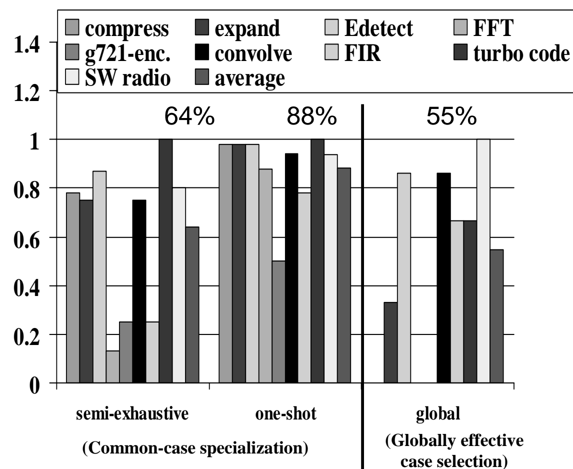


Figure 5.12: Search space reduction ratio in common case and global effective-case selection step

C programs	Code Quality								
	exhaustive			semi-exhaustive			one-shot		
	energy	performance	code size	energy	performance	code size	energy	performance	code size
Compress	0.91	0.91	1.01	0.91	0.91	1.01	0.93	0.93	1.15
Expand	0.84	0.83	1.15	0.84	0.83	1.15	0.90	0.90	1.12
Edetect	0.44	0.37	1.20	0.44	0.37	1.20	0.44	0.37	1.20
FFT	0.86	0.86	1.16	0.86	0.86	1.16	0.86	0.86	1.16
g721 encode	0.88	0.88	1.04	0.88	0.88	1.04	0.88	0.88	1.04
Convolve	0.54	0.48	1.18	0.54	0.48	1.18	0.54	0.48	1.18
FIR	0.53	0.53	1.12	0.53	0.53	1.12	0.53	0.53	1.12
turbo code	-	-	-	0.89	0.90	1.22	0.89	0.90	1.22
SW radio	0.67	0.65	1.09	0.67	0.65	1.09	0.67	0.65	1.09
Average	0.71	0.69	1.12	0.73	0.71	1.13	0.74	0.72	1.14

(a) Specialized code quality in SmartBadge environment

C programs	Code Quality								
	exhaustive			semi-exhaustive			one-shot		
	energy	performance	code size	energy	performance	code size	energy	performance	code size
Compress	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Expand	0.94	0.95	1.08	0.94	0.95	1.08	1.00	1.00	1.00
Edetect	0.27	0.26	1.04	0.27	0.26	1.04	0.27	0.26	1.04
FFT	0.18	0.19	1.14	0.18	0.19	1.14	0.18	0.19	1.14
g721 encode	0.91	0.95	1.01	0.91	0.95	1.01	0.91	0.95	1.01
Convolve	0.65	0.68	1.04	0.65	0.68	1.04	0.65	0.68	1.04
FIR	0.38	0.35	1.06	0.38	0.35	1.06	0.38	0.35	1.06
turbo code	-	-	-	0.82	0.82	1.23	0.82	0.82	1.23
SW radio	0.81	0.79	1.10	0.81	0.79	1.10	0.81	0.79	1.10
Average	0.64	0.64	1.06	0.67	0.67	1.09	0.68	0.67	1.08

(b) Specialized code quality in ST200 processor environment

Table 5.3: Quality of the code transformed with different approaches (normalized to original code)

5.6.2 Search space reduction

I first analyzed the effectiveness of the proposed search space reduction techniques. Figure 5.11 shows the pruning ratio achieved by each step with computation threshold, $CT = 0.1$. Notice that this step is architecture-independent as shown in Figure 5.3, thus Figure 5.11 is common to both SmartBadge and ST200 processor.

The procedure pruning step always plays an important role to reduce the search space, but call-site pruning step shows large variation depending on the property of the application programs. This is because the computational kernels of some programs such as `compress` and `FFT` were called only once while the kernel of `g721 encode` was called several times in different sites with different calling frequencies. Thus, this step is useful for the kernels called frequently in different sites with different frequencies.

The ineffectiveness of the case pruning step was due to high OT which was set to 0.5 for value profiling. Under this OT, the OR of each common value was usually large enough to yield NCE larger than CT used in this experiment (0.1). Dominated case pruning was effective for most of application programs because many of common values were constant ($OR = 1.0$).

Next, the pruning methods used in common-case specialization and global effective-case selection were evaluated. Figure 5.12 shows the pruning ratios of these

two steps for SmartBadge environment. Our technique in ST200 processor environment also showed similar results. As shown in Figure 5.12, both semi-exhaustive and one-shot approach drastically reduced the search space by 64% and 88%, respectively. Also, pruning technique in global effective-case selection step showed 46% of search space reduction and large variation of pruning ratio depending on the property of application programs. There was nothing to be pruned for `Compress`, `FFT` and `g721 encode` programs because only one case was passed from common case specialization step.

5.6.3 Code quality improvement

Both one-shot and semi-exhaustive approaches were compared to exhaustive approach in terms of code quality and specialization time. Common-case selection step was commonly used for each approach to avoid large search space. Also, global effective-case selection step was used in all three specializations because it always guaranteed the optimal solution and its worst case run time was same to exhaustive approach. As expected, the one-shot approach showed the smallest running time and semi-exhaustive approach was ranked at second. In average, both one-shot approach and semi-exhaustive approach are about 8.3 (8.0) times and 2.7 (2.5) times faster than exhaustive approach in SmartBadge (ST200 processor) environment, respectively. Notice that Figure 5.12 only shows the reduction ratio of the search space, which is different from the specialization time in the sense that search space reduction ratio only implies the reduction ratio of the number of specializations, while the specialization time includes partial evaluation, compilation, and instruction-set level simulation.

In SmartBadge environemnt, our tool was executed on SUN UltraSPARC running at 200MHz with 512MB memory. The overall procedure of our tool takes less than 20 minitues with one-shot approach, while it takes from 10 minitues (FIR) to 7 hours (turbo code) depending on the complexity of the loop structure in addition to the overall program complexity and program input data size.

In ST200 environment, our tool was executed on Sony VAIO R538DS equipped

with Pentium III running at 500MHz with 128MB memory. The difference of execution time between one-shot approach and semi-exhaustive approach is still large, but the semi-exhaustive approach benefits from the faster machine (also faster simulator) because it requires much more iterations including simulation, compilation, and specialization. `turbo code` with semi-exhaustive approach still showed the longest executio n time (2 hours and 20 minitues).

It is interesting that exhaustive approach often generated a huge size of code which is one of the main problems in partial evaluation. For the code, compilation or simulation was not terminated within a few hours, which is a bottleneck for automation. For this reason, I adopted time-out approach especially for the exhaustive approach by assuming that the code requiring long simulation time would be very huge and require large energy consumption.

Table 5.3 shows the quality of transformed code in terms of energy, performance, and code size for the three approaches. Notice that the energy consumption was measured with the consideration of shutdown technique. As shown in Table 5.3, semi-exhaustive approach is comparable to exhaustive approach in terms of transformed code quality with much less computation time(63% for SmartBadge and 60% for ST200 processor). One-shot solution is also useful by trading off its code quality and computation time. (About 8.0 times faster and 2% consumes more energy compared to exhaustive approach). I could not perform exhaustive approach for `turbo code` because its computational kernel had too many loops (18) which yielded a huge number of loop combinations ($2^{18} = 261844$). It is also worthwhile to mention that the deviation of improvement is largely depending on the nature of the programs. For the best case, the improvement is more than twice(`Edetect`), but for the worst case, about 10% (0%) is improved (`Compress`) in SmartBadge (ST200 processor) environment.

It is interesting that our tool specialized `Compress` and `Expand` in different ways depending on the target architecture. `Compress` and `Expand` show non-marginal improvement in SmartBadge environment, whereas their improvement ratio in ST200 processor is marginal. Also, the improvement ratio of `FFT` is much larger in ST200 processor environment than in SmartBadge environment, even though the specialized

programs	SmartBadge		ST200 processor	
	energy	perf.	energy	perf.
Compress_float	0.91	0.91	1.0	1.0
Compress_fixed	0.80	0.79	0.91	0.91
Expand_float	0.84	0.83	0.94	0.95
Expand_fixed	0.55	0.53	0.73	0.76

Table 5.4: Improvement ratio of floating-point and fixed-point versions (semi-exhaustive)

programs for both architectures are identical. The common feature of these programs is that the computational kernels of all three programs have floating-point operations which are not directly supported by the hardware in both architectures, but they are handled by floating-point emulation. From the careful analysis of these programs, I found two reasons for this fact. First, the computation cost of floating-point emulation in ST200 processor is much more expensive than in SmartBadge environment (relative to their integer operations). Notice that floating-point emulation is performed by the built-in library functions which is out of the scope in our technique. Second, the loop overhead in SmartBadge is larger than in ST200 processor.

The results in Table 5.4 support this claim. `Compress_float` and `Expand_float` are the floating-point versions used in Table 5.3 and `Compress_fixed` and `Expand_fixed` are their fixed-point versions, respectively. Notice that the improvement by the specialization is mainly due to loop unrolling for both versions of two programs. As shown in Table 5.4, the improvement ratio using our technique is about 2.5 times larger for the fixed-point version compared to the floating-point version in SmartBadge environment. On the other hand, it is about 5 times larger in ST200 processor environment. It means that the relative cost of floating-point emulation in ST200 processor environment is twice larger than that in SmartBadge environment. But, the improvement ratio using our technique in SmartBadge environment is still larger than in ST200 processor environment. It implies that the loop overhead elimination by our technique is more effective (about twice) in SmartBadge environment rather than in ST200 processor environment.

In case of `FFT`, the specialization step eliminates trigonometric functions such as `cos`. The computation cost of `cos` function is four times more expensive in ST200

processor environment than in SmartBadge environment in terms of number of clock cycles (measured by simulators). Thus, the elimination of such functions is more advantageous in ST200 processor than in SmartBadge environment.

In summary, our technique is more effective in fixed-point arithmetic programs, therefore it is desirable to apply our technique after transforming the floating-point arithmetic programs into the fixed-point arithmetic programs as proposed in [91]. Also, the computation cost of the built-in functions such as trigonometric functions is architecture dependent, thus the impact of the specialization varies largely depending on the underlying hardware architecture.

As a final remark, the run time of the optimization flow depends on the two user-defined constraints CT and OT that drive the pruning. Also, program size and loop depth are critical factors in specialization step, because our approach uses instruction set-level simulation. Nevertheless, it is important to remember that low energy and fast execution of the target code is the overall objective, which can be achieved at the expense of longer optimization time for large programs.

5.6.4 Input data sensitivity analysis

The variation of improvement (whatever the metric is) is a common problem of profiling-based techniques because profiling information can be largely varied depending on the trained input data set. Our technique is also affected by input data set and the improvement ratio shown in Section 5.6.3 may be largely varied if common values used for transformation heavily depend on input data set.

I analyzed the common values identified by our framework and they can be classified into two categories. The common values in the first category are sensitive to input data set, while those in the second category are independent to input data set, *i.e.* they are statically declared (or computed) values in somewhere of the program. I call the common values in the first category *dynamic common values* and those in the second category *static common values*. Notice that *static common values* are rarely (or never) changed input data identified by a programmer, but this information is not used for optimization due to the complexity and/or future modification.

A program transformed using *static common values* shows constant improvement ratio because the transformation is independent to input data set. In our experiment, many of benchmark programs were transformed using *static common values.* `Compress` and `Expand` programs initially computes `cos` table with fixed number of sampling points and these results are identified as common values. In `g721 encode` program, the *static common values* are a quantization table and its size defined in a program. It is interesting that two quantization tables with different size are defined in this program, but only one quantization table was consistently used in each call site. Thus, even though many *static common values* were observed in procedure point of view, each call site was related to a single *static common value.* Programs `Edetect`, `Convolve`, and `FIR` identified filter coefficient tables as *static common values* (with their size) and these coefficient values and size were efficiently used for the transformation. In program `turbo code`, the number of delay elements was identified as a *static common value.*

In case of `FFT`, the number of sampling points was identified as a `dynamic common value`, thus the improvement ratio was largely varied. In the worst case, the transformed program does not show any improvement if the identified common value is not observed during the program execution. However, the variation of *dynamic common value* was limited to several numbers such as 4, 8, 16, and so on. Such limited divergence can be handled in our framework because our framework can manipulate multiple common cases for single call site using multi-way branch statement with multiple *common value detection* procedures at the expense of code size increase.

To summarize, our technique shows constant improvement ratio when it transforms programs with *static common values*, but transformation with *dynamic common value* can largely change the improvement ratio depending on the input data like other profiling-based techniques. Also, restricted variation of *dynamic common value* can be treated by our framework at the expense of code size increase.

5.7 Chapter Summary

I presented algorithms and a tool flow to reduce the computational effort of software programs, by using value profiling and partial evaluation. I showed that the average energy and run time of the optimized programs is drastically reduced. The main contribution of this work is the automation of an optimization flow for software programs. Such a flow operates at the source level, and is compatible with other software optimization techniques, *e.g.*, loop optimizations and procedure in-lining.

Within our approach, a first tool performs program instrumentation and profiling to collect useful information for transformations, such as execution frequency and commonly-observed values at each call site. Using the profiling information, another tool selects common cases based on the estimated computational effort. Each selected case is specialized independently using a partial evaluator. In the selection step, code explosion due to loop unrolling - which may hamper partial evaluation - is avoided by code size estimation technique and pruning. Finally, the interplay among the multiple specialized cases is analyzed based on instruction-set level simulation. The transformed code shows in average 35% (26%) energy saving and 38%(31%) in average performance improvement with 7% (13%) code size increase in ST200 processor (SmartBadge) environment.

Currently, our approach is limited to the common cases at procedure level, but I believe that our technique can be extended to lower-level common cases (*e.g.* loop level) which may provide better quality of code specialization. Also, the specialization technique will be extended to consider more architecture dependent characteristics.

Chapter 6

Conclusion

Energy efficiency has become a major design goal for all types of electronic systems. As design complexity increases, designs are forced to adopt processor-based architectures to meet the time-to-market constraint and increase design flexibility. Typically, processor-based system can be structured as three layers. The hardware resides at the bottom layer, the OS (Operating System) layer is on top of hardware layer, and application program is the top layer. In these systems, hardware is the actual energy consumer, however software running on the processor controls the behavior of hardware and strongly affects on the overall system energy consumption. For this reason, software-oriented energy reduction has become one of the major trends for energy efficient system design.

For interactive application programs, I presented two DPM policies to reduce the wasted energy while the system is in idle state. These two approaches especially focus on the adaptive capability to cope with the workload variation and they are implemented in OS-level.

For computation intensive application programs, I presented a low-energy software optimization technique and this technique is implemented as a source to source code transformation framework and a tool. This framework and tool specialize the applications programs to highly expected situations and provides fully-automated transformation environment.

6.1 Thesis Summary

6.1.1 Dynamic Power Management

Dynamic power management policies reduce energy consumption by selectively placing the system into low power states. The quality of dynamic power management policy strongly depends on its adaptability, *i.e.* how well the policy copes with the variation of the workloads. In this work, I present and implement two adaptive DPM policies (*i.e. sliding window* and *adaptive learning tree* techniques) and the measurement results show large energy savings. Both methods commonly address the problem of non-stationary workloads for DPM policy, but they are different in many aspects.

The *sliding window* technique is based on a stationary stochastic policy which provides a theoretical optimal solution for stationary workloads. It extends the stationary stochastic policy to handle the non-stationary workloads using sliding windows and table look-up based interpolation technique. In detail, the adaptive policy computation of *sliding window* technique consists of two steps. In the first step performed at system design time, this technique pre-computes a set of stationary optimal policies by a linear-programming formulation. This pre-computation allows us to trade off between power and performance precisely. In the second step performed at run time, the workload statistics is estimated by sliding windows and the adaptive policy is computed from a set of pre-computed optimal stationary policies. The *sliding window* technique is developed for two-state machines (one active state and one sleep state), but it can be extended for multiple sleep state devices. This is the first stochastic policy which considers non-stationary workloads with a formal model of stochastic optimization.

On the other hand, the *adaptive learning tree* technique is a fully-heuristic method and cannot trade off between power and performance unlike *sliding window* technique. But, this policy does not require any pre-computation step at system design time. This technique predicts the next idle period length using the data structure called *adaptive learning tree* which is evolved as the system experiences the non-stationary workloads. The data structure records the recent user behavior patterns with prediction confidence level and the next user behavior is predicted from the user behavior

pattern with the highest prediction confidence level. This policy supports multiple sleep state devices which was not considered in previous methods.

Experimental results are only the way to compare these polices and their effectiveness are validated by both simulation and real measurement. The simulation and real measurements were performed for power-manageable hard disk drives installed in both laptop and desktop computers. The *sliding window* technique performed well in real measurement as well as simulation. This technique outperformed all other existing DPM policies compared and was comparable to the ideal policy called *best-oracle*. The *adaptive learning tree* technique also performed well in simulation environment, but its performance was not as good as *sliding window* technique in real measurement. This is because the hard disk drives used in real measurement support only single sleep state; hence, its capability cannot be fully tested.

To summarize, adaptability is the essential property of dynamic power management policies to handle the variation of the workloads. I proposed two DPM policies - *sliding window* and *adaptive learning tree* techniques which represent different solutions with different approaches to handle the non-stationary workloads. Both methods showed their outstanding adaptability which yields large energy reduction.

6.1.2 Low Energy Software Optimization

Low energy software optimization is one of the most promising techniques to reduce the energy consumed by software application programs on processor-based systems. This technique is especially effective on computation-intensive applications because the major energy consumer for these applications is the processor on which these application programs are executed.

I created an automatic source to source code transformation framework and a tool which specialize the software for highly expected situations for the energy reduction of the processors. In this framework, the highly expected situation is the commonly observed values passed to the procedure calls as their parameters. In detail, I presented algorithms and a tool flow to reduce the computational effort of software programs, by using value profiling and partial evaluation. I showed that the average energy and

run time of the optimized programs is drastically reduced. The main contribution of this work is the automation of an optimization flow for software programs. Such a flow operates at the source level, and is compatible with other software optimization techniques, *e.g.*, loop optimizations and procedure in-lining.

Within this framework, a first tool performs program instrumentation and profiling to collect useful information for transformations, such as execution frequency and commonly-observed values at each call site. Using the profiling information, another tool selects common cases based on the estimated computational effort. Each selected case is specialized independently using a partial evaluator. In the selection step, code explosion due to loop unrolling - which may hamper partial evaluation - is avoided by code size estimation technique and pruning. Finally, the interplay among the multiple specialized cases is analyzed based on instruction-set level simulation. The transformed code shows in average 35% (26%) energy saving and 38%(31%) in average performance improvement with 7% (13%) code size increase in ST200 processor (SmartBadge) environment.

6.2 Future Work

Energy efficient system design is an evolving area and still many researchers have devoted their efforts to this area with many different viewpoints. My work in this thesis is an approach to energy-efficient design in software perspective and there are still quite a few limitations to overcome.

The dynamic power management policies I presented can be extended to control multiple devices, as long as their number is small. Nevertheless, the problem of performing concurrent power management of multiple devices, under non-stationary workloads, remains a challenging problem for future research. Also, my DPM policies do not distinguish among the programs or processes executed on the processor. In other words, they solely depend on the previous request arrival ratio to identify the characteristics of the workload currently processed. The communication between operating system and application program can provide more information of the workload characteristics, which can improve the adaptability of DPM policies.

The framework for low-energy software optimization is limited to the common cases in procedure level, but I believe that our technique can be extended to the lower level common cases (*i.e.* loop level) which may provide better quality of code specialization. Also, the specialization technique can be extended to consider more architecture dependent characteristics such as memory hierarchy. Another promising challenge is to integrate *dynamic voltage scaling* (DVS) technique with the framework to slow down the processor when it executes the specialized code, which yields further energy reduction with losing the performance improvement obtained from the specialization.

Software restructuring for energy efficient system design is still an evolving area with many unsolved problems and open issues. This research topic will continue to be challenging in the next few years as designs with embedded processors and memory become increasingly more pervasive.

Bibliography

[1] ACPI, "ACPI: Advanced Configuration & Power Interface", available at http://www.teleport.com/˜acpi (1999)

[2] A. Acquaviva, L. Benini, and B. Riccó, "An Adaptive Algorithm for Low-Power Streaming Multimedia Processoing", *Design Automation and Test in Europe*, pp. 273-279, 2001

[3] L. O. Andersen, Program Analysis and Specialization for the C Programming Language, PhD thesis. DIKU, University of Copenhagen. May, 1994.

[4] Advanced Power Management Overview. http: www.developer.intel.com/IAL/powermgm/apmovr.htm

[5] G. Araujo and S. Malik. "Code Generation for Fixed-Point DSPs". *ACM TODAES*, vol.3, no.2, pp. 136-161, April 1998

[6] Advanced RISC Machines Ltd (ARM), ARM Software Development Toolkit Version 2.11, 1996

[7] K. Astrom, B. Wittenmark, *Adaptive Control* Addison-Wesley, 1989.

[8] D. Bacon, S, Graham, and O. Sharp, "Compiler Transformation for High-Performance Computing", *ACM Computing Surveys*, pp.345-420, vol26, No. 4, Dec. 1994

[9] T. Ball and J. Larus, "Optimally Profiling and Tracing Programs", *Proceedings of the 19th Annual Symposium on Principles of Programming Languages*, pp. 59-70, Jan. 1992

[10] L. Benini and G. De Micheli, *Dynamic Power Management: Design Techniques and CAD Tools*. Kluwer, 1997.

[11] L. Benini, A. Bogliolo, G. Paleologo and G. De Micheli, "Policy Optimization for Dynamic Power Management", *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 6, pp. 813-833, June 1999.

[12] L. Benini, A. Bogliolo, G. De Micheli, "A Survey of Dynamic Power Management Techniques," *IEEE Transactions on VLSI Systems*, February 2000.

[13] L. Benini and G. De Micheli, "System-Level Power Optimization Techniques and Tools", *ACM TODAES - Transactions On Design Automation of Electronic Systems*, vol. 5, issue 2, pp.115-192, Apr. 2000

[14] U.N. Bhat, *Elements of Applied Stochastic Process John Wiley & Sons, 1984*

[15] N. H. Bshouty *et al*, "On Learning Decision Trees with Large Output Domains", *Algorithmica*, Vol 20, pp.77-100, 1998

[16] B. Calder, P. Feller, and A. Eustace, "Value Profiling", *International Symposium on Microarchitecture*, pp.259-269, 1997

[17] B. Calder, P. Feller, and A. Eustace, "Value Profiling and Optimization", *Journal of Instruction-Level Parallelism*, vol. 1, Mar. 1999

[18] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle, *Custom Memory Management Methodology: Exploration of Memory Organization for Embedded Multimedia System Design*, Kluwer, 1998

[19] F. Catthoor, S. Wuytack, E. De Greef, L. Nachtergaele, and H. De Man, "System-Level Transformation for Low Power Data Transfer and Storage", A. Chandrakasan, R. Brodersen eds. Low-Power CMOS Design, IEEE Press, 1998

[20] N. Cesa-Bianchi *et al*, "On Bayes Methods for On-Line Boolean Prediction", *Algorithmica*, Vol 22, pp.112-137, 1998

[21] A. Chandrakasan and R. Brodersen. *Low-Power Digital CMOS Design.* Kluwer, 1995.

[22] S. Chirokoff and C. Consel, " Combining Program and Data Specialization", *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '99)*, pp.45-59, San Antonio, Texas, USA, January 1999

[23] P. Chou and G. Borriello, "Software Scheduling in the Co-Synthesis of Reactive Real-Time Systems", *Design Automation Conference*, pp. 1-4, 1994

[24] E.-Y. Chung, L. Benini, A. Bogliolo, and G. De Micheli, "Dynamic Power Management for Non-Stationary Service Requests", *DATE - Design Automation and Test in Europe Conference and Exhibition*, pp.77-81, 1999

[25] E.-Y. Chung, L. Benini, and G. De Micheli, "Dynamic Power Management Using Adaptive Learning Tree", *Proceedings of the Int.l Conference on Computer Aided Design*, pp.274-279,1999

[26] E.-Y. Chung, L. Benini, and G. De Micheli, "Energy Efficient Source Code Transformation based on Value Profiling", *1st workshop for Compilers and Operating Systems for Low-Power*, pp. D-1-D.7, Philadelphia, PA, 2000

[27] C. Consel and O. Denvy, "Tutorial Notes on Partial Evaluation", ACM Symposium on Principles of Programming Languages, pp.493-501, 1993

[28] K. Cooper, M. Hall, and K. Kennedy, "A Methodology for Procedure Cloning", Computer Languages, Volume 19, Number 2, pp 105-117, April, 1993

[29] K.D. Cooper, P. Schileke, "Non-local Instruction Scheduling with Limited Code Growth", *ACM SIGPLAN Workshop on Languages, Compilation, and Tools for Embedded Systems*, pp. 193-207, June 1998

[30] F. Douglis, P. Krishnan, and B. Bershad, "Adaptive Disk Spin-Down Policies for Mobile Computers", *Computing Systems*, vol. 8, pp. 381-413, 1995

[31] P. Duhamel and H. Hollman, "Split-Radix FFT Algorithm", *Electronics Letters*, vol. 20, no. 1, pp.14-16, Jan. 5, 1984

[32] A. Ehrenfeucht and D. Haussler, "Learning Decision Trees from Random Examples", *Information and Computation*, Vol 82, pp.231-246, 1989

[33] G. Esakkimuthu, N. Vijaykrishnan, M. Kandemir, M. Irwin, "Memory system energy: influence of hardware-software optimizations," *International Symposium on Low Power Electronics and Design*, pp. 244-246, 2000.

[34] P. Faraboschi and F. Homewood, "ST200:A VLIW Architecture for Media-Oriented Applications, at Microprocessor Forum", Oct. 9-13, 2000, San Jose, CA

[35] J. Flinn and M.Satyanarayanan, "Energy-aware adaptation for mobile applications", *Symposium on Operating Systems Principles*, pp. 48-63, 1999

[36] F. Gabbay and A. Mendelson, "Can Program Profiling Support Value Prediction?", *30th International Symposium on Microarchitecture*, pp. 270-280, Dec. 1997

[37] R. Golding, P. Bosh and J. Wilkes. "Idleness is not sloth", *Proceedings of Winter USENIX Technical Conference*, pp.201-212, 1995.

[38] R. Golding, P. Bosh and J. Wilkes. Idleness is not sloth. *HP Laboratories Technical Report HPL-96-140*, 1996.

[39] G. Goossens, J. V. Praet, D. Lanneer, W. Geurts, A. Kifli, C. Liem, and P. Paulin, "Embedded Software in Real-Time Signal Processing Systems: Design Technologies", *Proceedings of IEEE*, vol. 85, no. 3, pp. 436-454, Mar. 1997

[40] P. Greenawalt, "Modeling Power Management for Hard Disks", *Interantional Workshop on Modeling, Analysis, and Simulation for Computer and Telecommunication Systems*, pp.62-65, 1994

[41] J. Hagenauer, E. Offer, and L. Papke, "Iterative Decoding of Binary Block and Convolutional Codes", *IEEE trans. on Information Theory*, vol. 42, no.2, March, 1996

[42] S. Hanono and S. Devadas. "Instruction Selection, Resource Allocation, and Scheduling in the Aviv Retargetable Code Generator", *Design Automation Conference*, pp. 510-515, 1998

[43] O. Hernandez-Lerma, *Adaptive Markov Control Processes*, Springer-Verlag, 1989.

[44] C.-H. Hwang and A. Wu. "A predictive system shutdown method for energy saving of event-driven computation", *Proceedings of the Int.l Conference on Computer Aided Design*, pp.28-32, 1997.

[45] N. Jones, C. Gomard, and P. Sestoft, *Partial Evaluation and Automatic Program Generation*, Prentice Hall, 1993

[46] A.Karlin, M. Manasse, L. McGeoch, and S. Owickim "Competitive Randomized Algorithms for Non-uniform Problems", *Algorithmica*, 11(6), pp. 542-571, June 1994.

[47] R. Kravets, and P. Krishinan, "Application-driven power management for mobile communication", *Wireless Networks*, vol. 6, pp. 263-277, 2000

[48] P. Kumar, P. Varaiya, *Stochastic Systems: Estimation, Identification, and Adaptive Control*, Prentice-Hall, 1986.

[49] G. Lakshminarayana, A. Raghunathan, K. Khouri, K. Jha, and S. Dey, "Common-Case Computation: A High-Level Technique for Power and Performance Optimization", *Design Automation Conference*, pp.56-61, 1999

[50] H. Lekatsas, J. Henkel, and W. Wolf, "Code Compression for Low-Power Embedded System Design", *Design Automation Conference*, pp. 294-299, 2000.

[51] K. Lepak and M. Lipasti, "On the value locality of store instructions", *ISCA*, pp. 182-191, 2000

[52] R. Leupers, *Code Optimization Techniques for Embedded Processors Methods, Algorithms, and Tools*, Kluwer, 2000

[53] R. Leupers and P. Marwedel, *Retargetable Compiler Technology for Embedded Systems Tools and Applications*, Kluwer, 2001

[54] K. Li, R. Kumpf, P. Horton, and T. Anderson, "A Quantitative Analysis of Disk Drive Power Management in Portable Computers", *USENIX Winter Conference*, 1994, pp. 279-292.

[55] Y. Li and J. Henkel, "A Framework for Estimating and Minimizing Energy Dissipation of Embedded HW/SW Systems", *Design Automation Conference*, pp.188-193, 1997

[56] Y.-T. Li, S. Malik *Performance Analysis of Real-Time Embedded Software*, Kluwer, 1999

[57] S. Liao, S. Devadas, K. Keutzer, and S. Tjiang, "Instruction Selection Using Binate Covering for Code Size Optimization", *International Conference on Computer Aided Design*, pp. 393-399, 1995

[58] C. Liem, *Retargetable Compilers for Embedded Core Processors Methods and Experiences in Industrial Applications*, Kluwer, 1997

[59] M. Lipasti, C. Wilkerson, and J. Shen, "Value Locality and Load Value Prediction", *ASPLOS*, pp.138-147, 1996

[60] C.L. Liu, Introduction to Combinatorial Mathematics, McGraw-Hill, 1968

[61] J.R. Lorch, A.J. Smith, "Software Strategies for Portable Computer Energy Management", *IEEE Personal Communications*, vol. 5, issue 3, pp.60-73, Jun. 1998

[62] Y.-H. Lu and G. De Micheli, "Adaptive Hard Disk Power Management on Personal Computers, *Great Lakes Symposium on VLSI*, pp. 50-53, 1999

[63] Y.-H. Lu, T. Simunic, and G. De Micheli, "Software Controlled Power Management", *CODES*, pp.157-161, 1999

[64] Y.-H. Lu, E.-Y. Chung, L. Benini, and G. De Micheli, "Quantitative Comparison of Power Management Algorithms", *DATE - Design Automation and Test in Europe Conference and Exhibition*, pp.20-26, 2000

[65] Y.-H. Lu and G. De Micheli, "Comparing System-Level Power Management Policies", *IEEE Design & Test*, pp.10-19, March, 2001

[66] P. Marwedel and G. Goossens (editors), "Code Generation for Embedded Processors", Kluwer, 1995

[67] http://www.cs.ucla.edu/ leec/mediabench

[68] H. Mehta, R. Owens, M. Irwin, R. Chen, and D. Ghosh, "Techniques for Low Energy Software", *International Symposium on Low Power Electronics and Design*, pp.72-75, 1997

[69] J. Monteiro and S. Devadas. *Computer-Aided Techniques for Low-Power Sequential Logic Circuits.* Kluwer, 1997.

[70] W. Nebel and J. Mermet (Eds.). *Low-Power Design in Deep Submicron Electronics.* Kluwer, 1997.

[71] V. Nirkhe and W. Pugh, "Partial Evaluation of High-level Imperative Program Languages with Applications in Hard Real-Time Systems", *ACM Symposium on Principles of Programming Languages*, pp. 269-280, 1992

[72] P. Panda, F. Catthoor, N. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, P. Kjeldsberg, "Data and Memory Optimization Techniques for Embedded Systems", *ACM TODAES*, vol. 6, no. 2, pp. 149-206, Apr. 2001

[73] D.A. Patterson and J. L. Hennessy, Computer Architecture A Quantative Approach, 2nd edition, Morgan Kaufmann Publishers, 1996

[74] P.G. Paulin, C. Liem, T.C. May, and S. Sutarwala, "CodeSyn: a Retargetable Code Synthesis System", *7th International Symposium on High-level Synthesis*, May, 1994

[75] P. Paulin, C. Liem, M. Cornero, F. Nacabal, and G. Goossens, "Embedded software in real-time signal processing systems: application and architecture trends", *Proceeding of IEEE*, vol. 85, no. 3, pp. 419-435, Mar. 1997

[76] M. Pedram, "Power Management and Optimization in Embedded Systems", *Asia and South Pacific Design Automation Conference*, pp. 239-244, 2001

[77] J. Pierce, M. Smith, and T. Mudge. Instrumentation tools. in Fast Simulation of Computer Architectures (T. Conte and C. Gimarc, eds.), Kluwer, Boston, MA, 1995, pp. 47-86.

[78] M. Puterman, *Markov Decision Processes.* Wiley, 1994.

[79] Q. Qiu and M. Pedram, "Dynamic Power Management Based on Continuous-Time Markov Decision Process", *Proceedings of Design Automation Conference*, pp.555-561, 1999

[80] Q. Qiu, Q. Wu, and M. Pedram, "Stochastic Modeling of a Power-Managed System: Construction and Optimization", *Internaltion Symposium on Low Power Electronic Devices*, pp.194-199, 1999

[81] Q. Qiu, Q. Wu, and M. Pedram, "Dynamic Power Management of Complex Systems Using Generalized Stochastic Petri Nets", *Proceedings of Design Automation Conference*, pp. 352-356, 2000

[82] Q. Qiu, Q. Wu, and M. Pedram, "OS-Directed Power Management for Mobile Electronic Systems", *39th Power Source Conference*, pp. 506-509, 2000

[83] J.R. Quinlan, "Induction of Decision Trees", *Machine Learning*, Vol 1, pp.81-106, March, 1986

[84] J.R. Quinlan and R. Rivest, "Inferring Decision Trees Using the Minimum Description Length Principle", *Information and Computation*, Vol 80, pp.227-248, March, 1989

[85] J. M. Rabaey and M. Pedram (editors). *Low-Power Design Methodologies.* Kluwer, 1996.

[86] S. P. Rajan, A.Sudarsanam, and S. Malik, "Development of an Optimizing Compiler for Fujitsu Fixed-Point Digital Signal Processor", *CODES-International Workshop on Hardware/Software Codesign*, pp. 2-6, 1999

[87] S.E. Richardson, "Caching Function Results: Faster Arithmetic by Avoiding Unnecessary Computation", *Tech. report, Sun Microsystems Laboratories*, 1992

[88] Simon Richter and Max Berger ACPI4Linux: http://phobos.fachschaften.tu-muenchen.de/acpi/

[89] T. Simunic, L. Benini, and G. De Micheli, "Cycle-Accurate Simulation of Energy Consumption in Embedded Systems", *Design Automation Conferece*, pp. 867-972, 1999

[90] T. Simunic, L. Benini, and G. De Micheli, "Event-Driven Power Management of Portable Systems", *International Symposium in System Synthesis*, pp.18-23, 1999

[91] T. Simunic, L. Benini, G. De Micheli, and M. Hans, "Source Code Optimization and Profiling of Energy Consumption in Embedded Systems", *ISSS*, pp. 193-198, 2000.

[92] T. Simunic, L. Benini, P. Glynn, and G. De Micheli, "Dynamic Power Management for Portable Systems", *International Conference on Mobile Computing and Networking*, pp. 11-19, 2000.

[93] A. Srivastava and A. Eustace, "ATOM: A System for Building Customized Programming Analysis Tools", *Proceedings of the SIGPLAN 1994 Conference on PLDI*, pp.196-205, Jun. 1994

[94] M. Srivastava, A. Chandrakasan and R. Brodersen. "Predictive system shutdown and other architectural techniques for energy efficient programmable computation", *IEEE Transactions on VLSI Systems*, 4(1), pp.42-55, March 1996.

[95] http://www.st.com

[96] R. Stengel, *Optimal Control and Estimation.* Dover, 1994.

[97] http://www.eecg.toronto.edu/ stoodla/benchmarks/benchmarks.html

[98] C. L. Su, C. Y. Tsui, and A. M. Despain, "Saving Power in the Control Path of Embedded Processors", *IEEE Design and Test of Computers*, vol. 11, no. 4, pp. 24-30, Winter, 1994

[99] A. Sudarsanam. "Code Optimization Libraries for Retargetable Compilation for Embedded Digital Signal Processors". PhD Thesis, Princeton University Department of EE, May 15, 1998

[100] Stanford Compiler Group, The SUIF Library: A set of core routines for manipulating SUIF data structures, Stanford University, 1994

[101] V. Tiwari, S. Malik, and A. Wolfe, "Compilation Techniques for Low Energy: An Overview", IEEE Symposium on Low Power Electronics, pp. 38-39, 1994

[102] V. Tiwari, S. Malik, A. Wolfe, "Instruction Level Power Analysis and Optimization of Software", *Journal of VLSI Signal Processing Systems*, vol. 13, no.1-2, pp. 223-233, 1996

[103] F. Thoen, M. Cornero, G. Goossens and H. De Man, "Software Synthesis for Real-time Information Processing Systems", *Workshop on Languages, compilers, & tools for real-time systems*, pp. 60-69, 1995

[104] H. Tomiyama, H. T. Ishihara, A. Inoue, and H. Yasuura, "Instruction Scheduling for Power Reduction in Processor Based System Design", *Design Automation and Test in Europe*, pp.855-860, 1998

[105] Toshiba America, "HARD DISK DRIVES", available at http://www.toshiba.com/taecdpd/hddover.htm#6.4 (1999)

[106] K.S. Trivedi, Probability and Statistics with Reliability, Queuing, and Computer Science Applications, Prentice-Hall, 1982

[107] S. Udani and J. Smith. The power broker: Intelligent power management for mobile computing. *Technical report MS-CIS-96-12, Dept. of Computer Information Science, University of Pennsylvania*, may 1996.

[108] N. Vijaykrishnan, M. Kandemir, M. Irwin, H. Kim, and W. Ye, "Energy-driven Integrated Hardware-Software Optimizations using SimplePower", *ISCA*, pp.95-106

[109] P. Whittle, *Some Distribution and Moment Formulae for the Markov Chain J. Roy Stat. Soc. B 17, 1955.*

[110] John Wilkes, "HP Laboratories Disk I/O traces", available at http://www.hpl.hp.com/personal/John_Wilkes/traces (1997)

[111] W. Wolf, *Computers as Components - Principles of Embedded Computing System Design*, Morgan Kaufmann, 2001

[112] M. Wolfe, *High Performance Compilers for Parallel Computing*, Addison-Wesley, 1996

[113] Q. Wu and M. Pedram, "Dynamic Power Management in a Mobile Multimedia System with Guaranteed Quality-of-Service", *Design Automation Conference*, 2001

[114] K. Zhou, K. Glover, J. Doyle, *Robust and Optimal Control.* Prentice Hall, 1995.