MICRO-ARCHITECTURE LEVEL LOW POWER DESIGN FOR MICROPROCESSORS

XIA XIAO XIN

(B.Eng., HuaZhong University of Science and Technology)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

NATIONAL UNIVERSITY OF SINGAPORE

Acknowledgements

I would like to thank first and foremost my supervisor, Prof. Tay Teng Tiow, for supporting me throughout this work and for his constant advice and encouragement over the past four years. After leading me to the proposal of this project, he has provided valuable guidance, suggestions and support throughout the course of research. During times of difficulties, he has also shown much understanding and patience, which makes this four years study a memorable part of my life.

I would also like to thank everyone in the Digital and System Application lab for interest, advice and support. In no particular order they are Mr. Zhu Xiaoping, Mr. Pan Yan, Miss. Sun Yang, Mr. Xu Ce and all other members, for their times in constructive discussions over technical and academic problems. These discussions helped very much to clarify questions that are related to the research interest.

I would like to thank everyone in National University of Singapore with whom I have studied and worked over this time, and express my deepest gratitude to all those who have directly or indirectly provided advice and assistance during the course of my research work.

Last, and by no means least, thanks to my dear parents and my dear sister for their endless love and the constant support in my work whilst I've been studying.

Table of Contents

Acknowledgements I

Table of Contents II

Summary VI

List of Tables VIII

List of Figures IX

Chapter 1 1

Introduction 1

1.1 The Problem 1

1.2 Structure 4

Chapter 2 Power Dissipation Source and Low Power Techniques 7

2.1 Static Power Dissipation 7

2.1.1 Static Power Dissipation Sources 7

2.1.2 Static Power Reduction Techniques 11

2.2 Dynamic Power Dissipation 22

2.2.1 Dynamic Power Dissipation Sources 22

2.2.2 Dynamic Power Dissipation Reduction 25

2.3 Summary 34

Chapter 3 Motivation and Analysis Model 35

3.1 Motivation 35

3.1.1 OS Level DVS Algorithms 36

3.1.2 Source-code Level DVS Algorithms 37
3.1.3 Our Micro-architecture Level DVS Algorithms 40
3.2 Analysis Model 44
3.2.1 Basic Model 44
3.2.2 Basic Analysis 47
3.3 Summary 51
Chapter 4 Infrastructure 52
4.1 Benchmarks 52
4.2 Simulation Environment 53
4.2.1 The Simulator 53
4.2.2 Energy Measurements 56
4.3 Summary 58
Chapter 5 IPC-Driven Online Power Reduction Method 59
5.1 Motivation for IPC Indicator 61
5.1.1 IPC variations 61
5.1.2 IPC Indicator 62
5.2 Methodology 65
5.2.1 Identification 65
5.2.2 Prediction 67
5.2.3 Speed-setting 68
5.3 Results 69
5.3.1 Evaluation metric 69

5.3.2 Principal results 71
5.3.3 Impact of interval length 73
5.3.4 Sensitivity to Slowdown Threshold 75
5.3.5 Overhead 79
5.3.6 Comparison 81
5.4 Summary 82
Chapter 6 IPC-Driven Offline Power Reduction Method 84
6.1 Methodology 86
6.1.1 Phase Identification 86
6.1.2 Code Matching 88
6.1.3 Slowdown Process 91
6.2 Results 93
6.2.1 Evaluation metric 93
6.2.2 Principal results 94
6.2.3 Impact of Phase Interval Length 97
6.2.4 Sensitivity to Slowdown Threshold 99
6.2.5 Overhead 103
6.2.6 Comparison with the IPC-driven online Method 105
6.2.7 Comparison with Other Offline Methods 107
6.3 Summary 108
Chapter 7 Methods to Identify Related Micro-architecture Parameters 110
7.1 Application Power Behavior Identification Method 111
7.1.1 Introduction 111

7.1.2 Methodology 112
7.1.3 Results 117
7.1.4 Conclusion 122
7.2 Data Dependence Length Identification Method 124
7.2.1 Introduction 124
7.2.2 Methodology 125
7.2.3 Results 129
7.2.6 Conclusion 133
7.3 Summary 134
Chapter 8 Conclusions and Future Work 135
8.1 Summary of Work 135
8.2 Summary of Contributions 137
8.3 Future work 138
Bibliography 140

Summary

With rapid advances in CMOS technology, power dissipation has become a great concern in modern microprocessor design, not only for battery-operated potable devices but also for high-end computer systems. Minimizing power dissipation of processors leads to many benefits, such as prolonging the battery lifetime of portable devices and reducing the heat dissipation and cooling cost of computer systems.

In this thesis, we are going to propose efficient designs for reducing power dissipation of the microprocessor. First of all, we investigate background and techniques for reducing microprocessor power dissipation. Then we attempt to address power dissipation issue of the microprocessor at the micro-architecture level, and present a realistic analysis model to discuss and identify possible power reduction opportunities during application execution. Finally, based on our analysis model, we propose two novel schemes at the micro-architecture level to reduce runtime power dissipation of microprocessors. Both methods make use of a micro-architecture parameter-IPC to identify potential power reduction opportunities during application execution.

Firstly, an IPC-driven online power reduction scheme is presented. This design employs the micro-architecture parameter (IPC) as the runtime performance indicator to dynamically scale the voltage and frequency of a processor. The basic idea in this interval-based identification and prediction design is to trace the current interval's performance activity level and predict the coming interval at which certain power-performance trade-off would be profitable.

Then, by using the same micro-architecture parameter, an IPC-driven offline power reduction

scheme is presented. This code analysis and reconfiguration design first identifies code sections that have appropriate IPC values and could make contributions to microprocessor power reduction, and then profiles them to dynamically scale the voltage and frequency of the microprocessor at appropriate points during application execution. For both low-power design schemes, simulation results showed that they significantly reduced the processor runtime energy consumption with minimal application performance degradation. Furthermore, both schemes could achieve better results when comparing with other state-of-the-art related works.

Beside the two micro-architecture level low-power designs, we also propose two methods to identify related micro-architecture parameters: runtime power behavior and data dependence length of applications. The two micro-architecture parameters could be used to evaluate the two low-power designs proposed by us.

List of Tables

Table 4.1: Summary of selected benchmarks..53

Table 4.2: Wattch Baseline Simulation Model..56

Table 5.1: Transition times for different IPC threshold 80

List of Figures

Fig. 1.1: Trends in power dissipation and the cost of cooling [6]..............................3

Fig. 2.1: ITRS projections for device power dissipation [7]..8

Fig. 2.2: Leakage current mechanisms of deep-submicron transistors [8].......................8

Fig. 2.3: Static Power Reduction Techniques...11

Fig. 2.4: Maximum Clock Frequency vs. Supply Voltage [41]....................................24

Fig. 2.5: Dynamic Functional Unit Assignment [60]..31

Fig. 3.1: Possible overlaps in peripheral and CPU computing operations......................46

Fig. 4.1: Architecture of the Wattch Simulator...54

Fig. 5.1: IPC variations during a short execution period of “gcc”................................61

Fig. 5.2: Analysis model...63

Fig. 5.3: Principal results: ES, PD, and EPI...71

Fig. 5.4: The impact of different interval length on “gcc”...74

Fig. 5.5: Energy savings for different IPC thresholds..76

Fig. 5.6: Performance degradations for different IPC thresholds...................................77

Fig. 5.7: Energy*performance improvement for different IPC threshold.......................78

Fig. 5.8: Comparison between Our algorithm and Weiser’s algorithm......................81

Fig. 6.1: Principal results: ES, PD, EPI..94

Fig. 6.2: The impact of different interval length on “gcc”...98

Fig. 6.3: Energy saving results...100

Fig. 6.4: Performance degradation results..101

Fig. 6.5: Energy*performance improvement results...102

Fig. 6.6: Transition times for different IPC threshold...104

Fig. 6.7: Comparison between the online and offline algorithms................................106

Fig. 6.8: Comparison between our and Hsu's offline DVS algorithm.........................107

Fig. 7.1(a): Measured runtime power behavior for "vortex"..118

Fig. 7.1(b): Phase estimated power behavior for "vortex"...118

Fig. 7.2(a): Measured runtime power behavior for "adpcmencode".............................120

Fig. 7.2(b): Phase estimated power behavior for "adpcmencode"................................120

Fig. 7.3: Error rates...121

Fig. 7.4: Steps for DDL identification method...125

Fig. 7.5: DDL Example...127

Fig. 7.6: Pseudocode for DDL Identification Algorithm..129

Fig. 7.7(a): MAX_DDL of the complete application execution...................................130

Fig. 7.7(b): MAX_DDL of the representative phases...130

Fig. 7.8(a): TOTAL_DDL of the complete application execution...............................132

Fig. 7.8(b): TOTAL_DDL of the representative phases...132

Chapter 1

Introduction

Power dissipation is becoming a crucial design constraint for modern microprocessors. This thesis investigates low power design schemes at the micro-architecture level to reduce power dissipation of microprocessors. In this chapter, we shall define the problem to be addressed, and describe the structure of the thesis.

1.1 The Problem

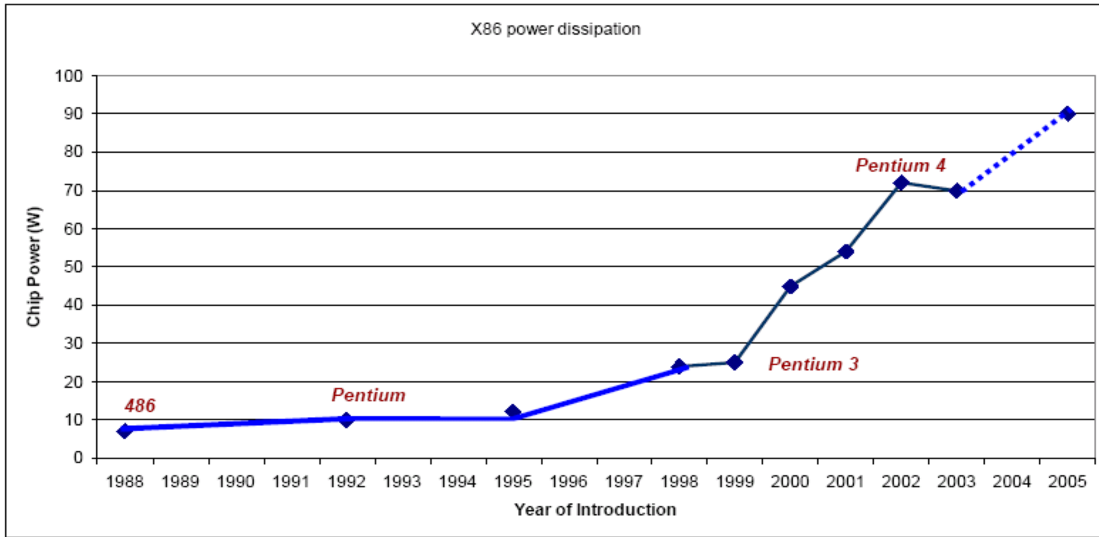
With the rapid growth of the internet and computer technology, portable devices, such as cellular phones, Personal Digital Assistants (PDA) and Global Positioning System (GPS) navigators, have become increasingly popular and widely-used. For these widespread portable electronic products, modern consumers require not only mobile computing ability, but also fast executing speed and various entertainment functions. The ability to fulfill these requirements usually lies on microprocessors embedded in the portable devices. To achieve faster computing speed, modern microprocessors have been pushed to higher clock speed and implemented with greater parallelisms. On the other hand, to accomplish more complicated functions, modern microprocessors have been packed with larger on-chip caches and more complex logic structures.

However, with the dramatic increase in executing speed and on-chip functions in a

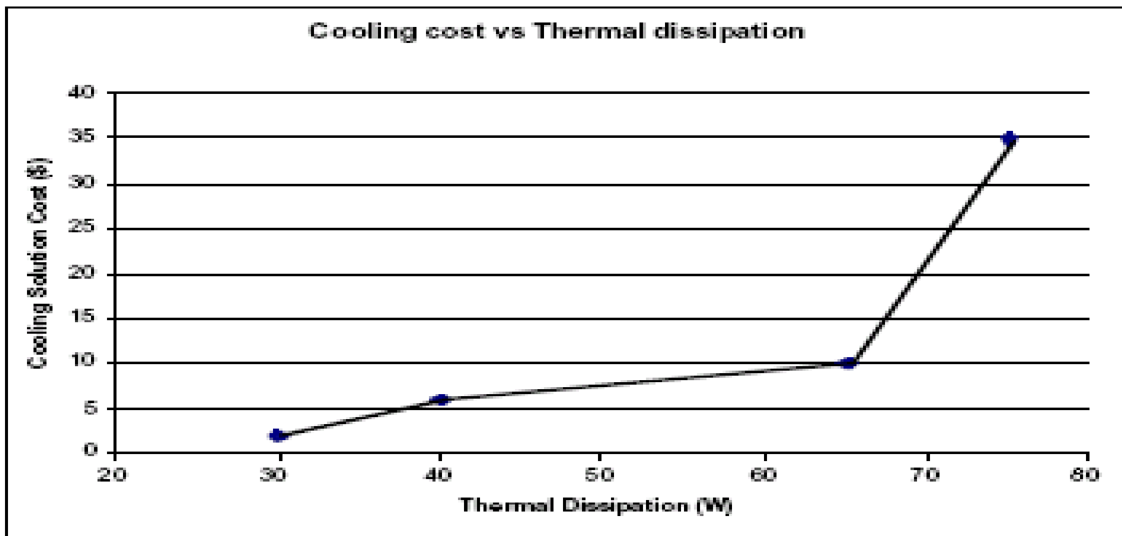
microprocessor, power dissipation also increases significantly. For example, maximum power dissipation of recent microprocessors has reached 130watts [1]. Such high power dissipation of microprocessors causes problems in at least two aspects.

Firstly, high power dissipation of microprocessors limits the "battery-life" of portable products. As is well known, battery-life is an important factor in the adoption of these battery-powered portable devices. In general, the battery life-time depends on both the battery capacity and the power dissipation in a portable device. However, improvements in the capacity of batteries can not keep pace with the increasing power demand of today's portable devices [2, 3, 4]. Thus, minimizing power dissipation of the portable devices is an efficient approach to prolong the battery life. As the microprocessor is a key component in a portable device, minimizing its high power dissipation could contribute much to the total power dissipation reduction of a portable device, and it is also very helpful to increase the battery life of the battery-powered device.

Secondly, high power dissipation of microprocessors leads to high chip temperature during operation. High operating temperature may lead to phenomena such as *electromigration* and *hotelectron* effects in the circuit, thereby reducing reliability of the whole system. As studied in [5], researchers found that every 10℃ increase in operating temperature roughly doubles the failure rate of an Integrated Circuit (IC). To reduce the failure rate caused by high temperature, large and expensive cooling systems have to be incorporated into computer systems to ensure proper operation.



(a): Power trends



(b): Cooling cost

Fig. 1.1: Trends in power dissipation and the cost of cooling [6]

Figure 1.1(a) shows the trends in power dissipation of Intel processors over the past fifteen years. As shown in the graph, more recent processors have much higher maximum power dissipation, increasing by a factor of 2 every four years [6]. Figure 1.1(b) shows the costs involved in removing this power (converted to heat) from the processors. This graph shows how the cost of cooling has increased as the amount of

heat produced has risen. It can be seen that the cooling cost rises non-linearly with the power of the processor [6]. From the two graphs, it is obvious that: reducing the amount of power dissipation in a processor would decrease the overall system cost.

To address the above two issues, a lot of research effort has been focused on developing microprocessors with high performance and minimal power consumption. To achieve this goal, various low-power technologies, from transistor and gate levels to operating system and application levels, have been proposed in the past years, and we will present and discuss them in the next chapter. In this thesis, we focus on reducing power dissipation of microprocessors at the micro-architecture level, and successfully propose two new and efficient low-power strategies, which will be presented in the following chapters.

1.2 Structure

The remainder of this thesis is organized as follows. Chapter 2 describes the basic issues of processor power dissipation and investigates various types of power dissipation sources in microprocessors. In particular, this chapter focuses on reviewing distinguished low-power techniques to reduce power dissipation induced by these sources in microprocessors.

In Chapter 3, firstly, the motivation for our micro-architecture level low-power design schemes is presented. Following that, an analysis model for our schemes is described in detail, and then the trade-off between power and performance of microprocessors in our schemes are studied.

In Chapter 4, the benchmark applications used to evaluate our proposed schemes in this thesis are presented. In addition, the simulation environment and the processor architecture of the simulator are also described in this chapter.

Chapter 5 describes a scheme that employs a micro-architecture parameter (IPC) as the performance indicator for specific processor runtime periods, and implements an interval-based identification and prediction mechanism for processor demand to reduce its power dissipation with minimal performance degradation. The basic idea for this design is to trace the current interval's performance activity level in terms of the IPC value and then use it to predict the processor demand for the coming interval at which certain power-performance trade-off would be profitable. Results show that this design scheme takes advantage of energy reduction as well as provides fine-grained, tight control over performance loss.

In Chapter 6, using the same micro-architecture parameter (IPC), a code analysis and reconfiguration scheme for microprocessor power reduction is presented. This trace-based low power design is implemented to identify code sections in an application that have appropriate IPC values and could make contributions to program runtime power reduction. These traced code sections are then profiled to dynamically scale the voltage and frequency of the microprocessor at appropriate points during execution. Experiment results show that our trace-based code analysis and reconfiguration mechanism significantly reduces the energy consumption of microprocessors without degrading the performance very much.

Chapter 7 presents two efficient methods to identify two useful micro-architecture

parameters, which are runtime power behaviors and data dependence length (DDL) of an application. Firstly, a method to identify application runtime power behaviors is presented. This method employs a phase-based analysis approach to obtain the runtime power dissipation information of an application and then characterize its runtime power behaviors. Then, a data dependence length identification method is presented. This method also uses the phase analysis technique to identify dynamic data dependence information among runtime instructions of a program and then use data dependence length (DDL) to characterize dynamic data dependence of the whole program. Experiment results demonstrate that both methods could identify the target micro-architecture parameter accurately and speedily.

Finally, Chapter 8 concludes this thesis, summarizing the main results and contributions, and describing directions that future work could pursue in this research area.

Chapter 2

Power Dissipation Source and Low Power Techniques

In general, power dissipation of microprocessors can be divided into two categories:

1) Static power dissipation, which arises from leakage currents and is generally independent of logic switching of circuits.

2) Dynamic power dissipation, which arises from the switching activities of logic circuits.

In this chapter, we will investigate both static power dissipation and dynamic power dissipation. In Section 2.1, we shall review leakage-induced static power dissipation. We shall examine the various sources for static power dissipation and the techniques to reduce static power dissipation. In Section 2.2 we shall describe the switching-induced dynamic power dissipation. We investigate sources for dynamic power dissipation and present low-power techniques to minimize them.

2.1 Static Power Dissipation

2.1.1 Static Power Dissipation Sources

In deep sub-micrometer regimes, leakage current increases with reduced threshold voltage, channel length and gate oxide thickness. The high leakage current is becoming a significant contributor to the overall power dissipation of CMOS circuits.

Figure 2.1 shows the projection of the International Technology Roadmap for Semiconductors (ITRS) for the trend of static and dynamic power dissipation with respect to technology progress [7]. It can be seen that the static power dissipation is expected to exceed the dynamic power dissipation unless effective static power reduction techniques are properly applied.

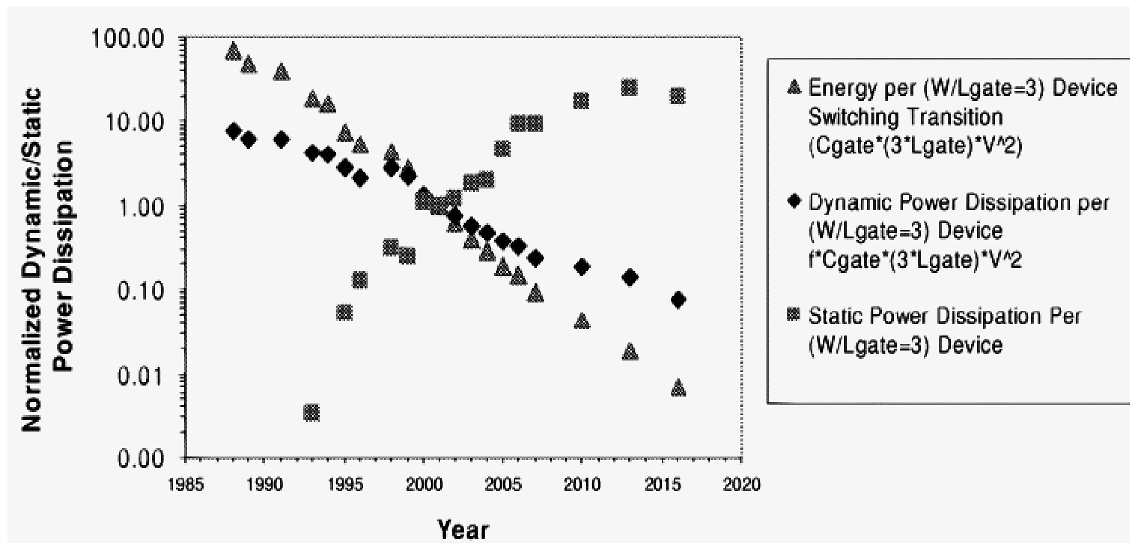


Fig. 2.1: ITRS projections for device power dissipation [7]

As known, for deep-submicron transistors, there are six major leakage mechanisms that contribute to the static power dissipation, as illustrated in Figure 2.2.

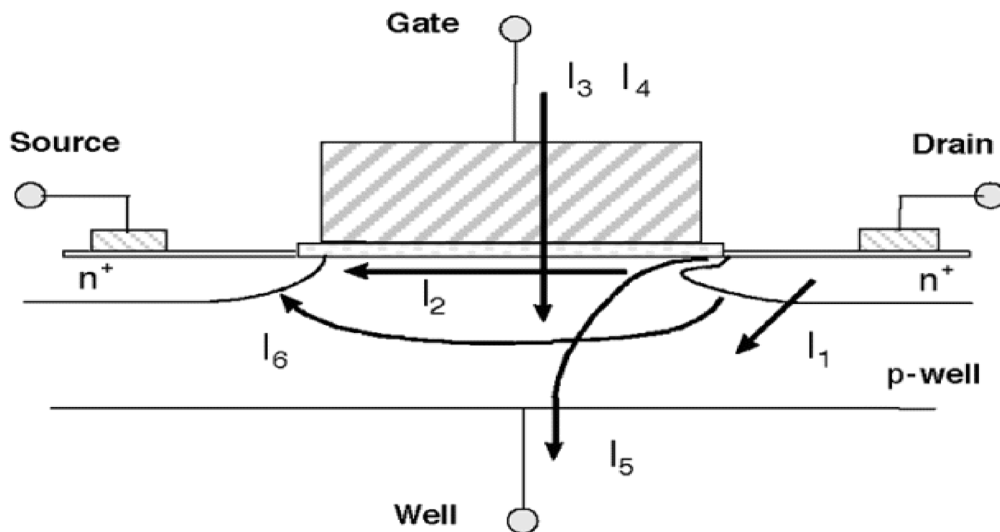


Fig. 2.2: Leakage current mechanisms of deep-submicron transistors [8]

As is shown in Figure 2.2, the six leakage mechanisms [8] are: PN junction reverse-bias current (I_1), sub-threshold leakage (I_2), tunneling into and through gate oxide (I_3), injection of hot carriers from substrate to gate oxide (I_4), gate-induced drain leakage (I_5) and punch-through (I_6). In general, currents I_2, I_5, and I_6 are off-state leakage mechanisms, while I_1, I_3, and I_4 occur in both ON and OFF states.

2.1.1.1 PN-junction reverse-bias current (I_1)

Normally, PN junction leakage current is generated when drain and source to well junctions are reverse-biased. A reverse-bias PN junction leakage (I_1) has two main components: 1) minority carrier diffusion and drift near the edge of the depletion region; 2) electron-hole pair generation in the depletion region of the reverse-biased junction [9]. As is studied in [9], PN-Junction reverse-bias leakage is a complex function of junction area and doping concentration.

2.1.1.2 Sub-threshold leakage (I_2)

The sub-threshold leakage is the leakage between source and drain in an off-state transistor. In modern MOSFETs, weak inversion leakage is the dominant part in the sub-threshold leakage. Other effects like Drain Induced Barrier Lowering (DIBL), Body Effect, Narrow-Width Effect, Channel Length Effect and Temperature Effect may also add to the sub-threshold leakage [8].

2.1.1.3 Tunneling into and through gate oxide (I_3)

The gate oxide tunneling current is incurred from the tunneling of electrons between substrate and gate through the gate oxide. Basically, the tunneling effect occurs when

the high electric field is coupled with low oxide thickness. In general, the mechanism of tunneling between substrate and gate can be primarily divided into two parts: Fowler-Nordheim (FN) tunneling and direct tunneling.

2.1.1.4 Injection of hot carriers from substrate to gate oxide (I_4)

In a short-channel transistor, the hot-carrier injection leakage occurs when electrons or holes gain sufficient energy from the electric field to cross the interface potential barrier and enter into the oxide layer. Usually, this effect is due to high electric field near the Si-SiO_2 interface. Since electrons have a lower effective mass than that of holes and the barrier height for electrons is also less than that for holes, the injection from substrate (Si) to gate oxide (SiO_2) is more likely for electrons than holes.

2.1.1.5 Gate-induced drain leakage (I_5)

Gate-induced drain leakage (GIDL) is due to high field effect in the drain junction of an MOS transistor. As is presented in [96], a path for the GIDL is completed when the substrate is at a lower potential for minority carriers and the induced minority carriers underneath the gate are swept laterally to the substrate. Generally, GIDL is increased by thinner oxide thickness and higher potential V_{dd} between gate and drain.

2.1.1.6 Punch-through (I_6)

In short-channel devices, punch-through occurs when the combination of channel length and reverse bias leads to the merging of the depletion regions. In sub-micrometer MOSFETs, V_{th} adjust implant is usually used to have a higher doping at the surface. This causes a greater expansion of the depletion region below the

surface, and thus the punch-through leakage current is generated below the surface.

2.1.1.7 Static power dissipation model

From the above discussion, it can be seen that the static power dissipation is very complex and thus is not easy to model. However, the static power dissipation can be simplified and represented by the following formula:

$$P_{static} = I_{leak} \times V_{DD} \tag{2.1}$$

Where I_{leak} is the cumulative leakage current due to all the components (I_1 to I_6) described previously.

2.1.2 Static Power Reduction Techniques

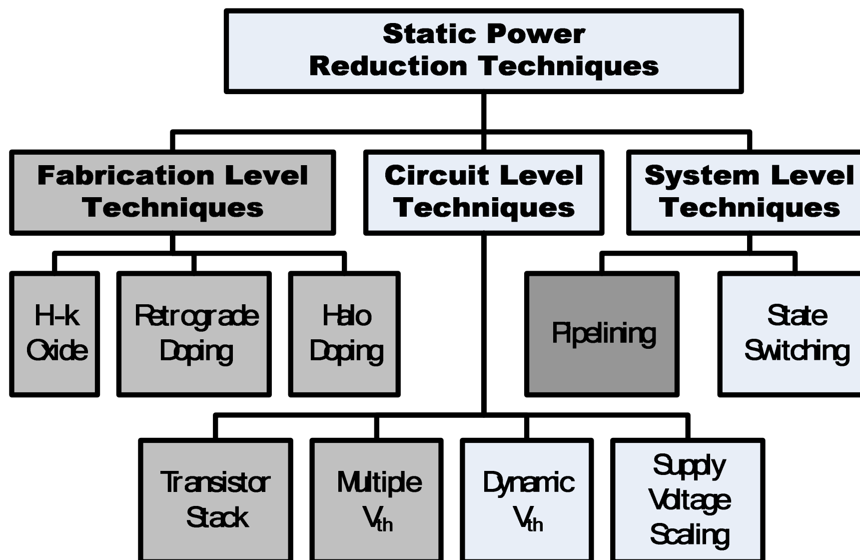


Fig. 2.3: Static Power Reduction Techniques

There is a wide range of low power techniques addressing static power dissipation, from fabrication level engineering to system level design. As a quick summary, we illustrate them in Figure 2.3. Each of these techniques will be presented in the

following sub-sections.

2.1.2.1 Fabrication Level Techniques

To minimize the overall static power dissipation, a straightforward way is to minimize the leakage current in each transistor. This can be done through fabrication techniques of transistors. Currently, fabrication techniques, such as high-k insulating materials, retrograde doping and halo doping, are already in use to provide transistors with the best performance and reduce the leakage at the same time. Here we present some examples for these fabrication techniques, illustrated as below:

- Y. Taur (2000)

  In [10], Y. Taur found that with deep submicron transistors, to maintain performance, scaling happens not only in the lateral dimension (channel length), but also in the vertical dimension, doping concentration and supply voltage. Thus, as gate oxide thickness got thinner, this results in increased leakage through gate node. To solve this problem, the author proposed to use high-k insulating materials, which increases physical thickness of the insulator while keeping reduced equivalent electrical thickness and eventually minimizes the leakage current through gate node.

- S. Thompson et al. (1998)

  As the channel length is scaled down, punch-through current becomes a big issue. At the same time, to maintain device performance, the mobility of the channel surface should be good enough. Thus, a better channel doping profile

should be with a low surface doping concentration followed with a highly doped sub-surface doping region. This is called “Retrograde Doping”.

In the study of [11], S. Thompson et al. illustrated the retrograde doping technique and its useful effect on minimizing the punch-through leakage current. As they found, the low surface doping is to make sure less impurity presented in the surface, and hence the mobility will be higher. Furthermore, the higher sub-surface concentration can counteract the nearing of source and drain regions, which consequently reduces the punch-through leakage current in the channel.

- D. Fotty (1997)

In the study of [12], D. Fotty suggested using the halo doping technology to reduce the sub-threshold leakage. In general, halo doping is introduced to provide a way to control the dependence of threshold voltage on channel length. As the author found, below the edge of the gate, which is also the end of the source or drain region, the introduced halo doping results in a narrower depletion region, and thus reduces the charge-sharing effect and the threshold voltage degradation, and eventually reduces the sub-threshold leakage.

The designs presented in this section have focused on fabrication techniques to minimize the static leakage current in each transistor. In these fabrication techniques, high-k gate dielectrics are expected to lower the static leakage [13]. On the other hand, retrograde and halo doping are also used as a means to decrease the static leakage

current, by scaling the channel length and increasing the transistor drive current [14, 15, 16, 17]. More detailed discussion of these fabrication techniques can be found in [8]. So far, the fabrication techniques are commonly employed in transistors to provide good performance, and also minimize the overall static leakage. With the advance of technology, more and more fabrication techniques are predicted to be used to reduce the leakage-induced power dissipation in future.

2.1.2.2 Circuit Level Techniques

With the fabrication level techniques applied to extremes, additional leakage power reduction can be achieved by carefully designing the circuit structures. In this section, we will present several popular circuit level techniques which are used to reduce the static leakage current.

A) Transistor Stack

One promising way to reduce static leakage is by intentionally turning off a series-connected transistor. In general, sub-threshold leakage current can be reduced when more than one transistor in the stack is turned off. This is known as the stacking effect [18]. Furthermore, according to the study in [19], the leakage of a two-transistor stack is an order of magnitude less than the leakage in a single transistor. Thus, researchers proposed to use transistors stack to reduce the static leakage current and its induced power dissipation. Some applications using transistors stack are presented in the following.

- M. C. Johnson et al. (1999)

As studied in [20], to reduce the leakage current in transistors, researchers proposed an off-state transistor stack approach. By identifying a low-leakage state and inserting leakage-control transistors only where needed, this method carefully selected the input vector so as to allow more off-state transistors in series. According to their experiment results, it was proven to be an effective way to control the sub-threshold leakage.

- M. Powell et al. (2000)

In the work of [21], to reduce leakage power dissipation, M. Powell et al. proposed a circuit-level technique to implement the transistor stack in processors. They employed additional transistors to gate a circuit structure from the power supply, as done with the Gated-V_{DD} circuit technique. Their results indicated that Gated-V_{DD} together with a resizable cache architecture reduced energy-delay very much with minimal impact on performance.

- S. Mukhopadhyay et al. (2003)

As presented in [22], S. Mukhopadhyay et al. first modeled the overall leakage in a stack of transistors, and then explored the opportunities for leakage reduction in the standby mode of operation for scaled technologies. To implement the transistor stack, the researchers proposed a novel technique of input vector selection to reduce total leakage in a circuit. Results showed that their technique achieved 44% savings in total leakage in 50-nm devices compared to the conventional stacking technique.

The designs presented in this section have focused on using transistor stack to minimize the leakage in transistors. Some schemes simply inserted transistor stack to control the leakage power dissipation [23, 24, 25]. As showed in their results, transistor stack is efficient to reduce the leakage current. However, as a result of introducing additional transistors into a chip circuit, this technique increased the transistor number in a chip and made its architecture more complex, thereby leading to additional dynamic power.

B) Multiple V_{th} and Dynamic V_{th}

As the sub-threshold leakage has an exponential dependence upon the threshold voltage, multiple threshold voltages can be provided in a single chip for proper use to reduce the leakage current. In general, higher threshold transistors can suppress the leakage while the lower threshold transistors can provide higher performance. There are various ways to achieve the varied threshold voltage. For example, changing the channel doping, gate oxide thickness, channel length, and body bias [26, 27] can all affect the final threshold voltage of a transistor. Thus, we can change the V_{th} either statically or dynamically. There are some useful strategies proposed by former researchers, as illustrated in the following.

- H. Makino et al. (1998)

In 1998, H. Makino et al. [28] suggested an auto-backgate-controlled MT-CMOS circuit to provide multi-threshold voltages for both p-channel and n-channel transistors. This design is similar to transistor stack. Additional

high-threshold transistors were put in series to low V_{th} circuit and these additional transistors reduce leakage of a circuit in sleep mode. Experiments showed their method achieved good results.

- N. Tripathi et al. (2001)

In [29], researchers proposed an algorithm to realize dual threshold CMOS circuits. In their algorithm, it employed transistors to lower thresholds in critical paths and thus guarantee best performance while applying higher threshold elsewhere. The results showed that their algorithm reduced the leakage current with better results for ISCAS benchmark circuits compared to other reported results.

- T. Inukai et al. (2001)

As is well-known, by changing the body bias of transistors, the threshold voltage can be manipulated at run time. In [30], researchers investigated characteristics of variable threshold voltage CMOS (VT-CMOS) in series connected circuits, and found that the leakage power dissipation of transistors is minimized by utilizing VT-CMOS while the performance degradation is suppressed due to the body effect in series connected circuits.

The designs presented in this section have focused on using multiple V_{th} and dynamic V_{th} to reduce the leakage current in transistors. Some designs employed inserted control transistors or circuits to implement multiple V_{th} and reduce the leakage [31, 32]. Other schemes utilized back-gate bias control to carry out dynamic V_{th} adjustment to minimize the leakage current [33, 34]. Results of these examples

proved that it is an effective way to control the leakage current of transistors by changing the V_{th} statically or dynamically. Although achieving good power reduction, similar to the problem in transistor stack design, it also introduced additional transistors/devices and consequently increased the complexity of chip circuits.

C) Supply Voltage Scaling

Designed to reduce dynamic power dissipation, voltage scaling technique is the most successful and widely used low-power technique. However, as found, it is also an effective method for static leakage reduction. There are some applications by using supply voltage scaling to reduce static power dissipation, described as below.

- A. J. Bhavnagarwala et al. (2000)

  In [35], researchers found that the sub-threshold leakage can be reduced when the supply voltage is scaled down. As is identified by them, the reason is that Drain Induced Barrier Lowering (DIBL) also decreases as the supply voltage decreases. Moreover, their results of experiments proved that supply voltage scaling is helpful to minimize the sub-threshold leakage and static power dissipation.

- S. Tyagi et al. (2000)

  In the study of [36], S. Tyagi et al. presented that supply voltage scaling achieved sub-threshold and gate leakage reduction in the orders of V^3 and V^4 respectively. In their experiments, results showed that it significantly reduced the static power dissipation by scaling supply voltage.

- M. Takahashi et al (1998)

  In [37], researchers proposed to use clustered voltage scaling to reduce the leakage-induced power dissipation for mobile multimedia circuits. In their design, transistors for critical and non-critical paths were separately clustered and powered by higher and lower supply voltages, respectively. By using the clustered voltage scaling, they found that the overall static power dissipation of the design was much smaller since the leakage current in circuits was reduced.

The designs presented in this section have focused on using supply voltage scaling to reduce leakage current in transistors. To achieve low-power benefits, some researchers used static supply scaling to lower supply voltage [38, 39, 40]. On the other hand, researchers employed dynamic supply scaling to minimize the leakage [41]. All these techniques showed that supply voltage scaling is useful to minimize the leakage current and hence reduce the static power dissipation. Thus, although supply voltage scaling is originally designed to reduce dynamic power dissipation, it has an additional and effective purpose for static power dissipation reduction.

2.1.2.3 System Level Techniques

Even higher level low power techniques are proposed by researchers to further reduce static power dissipation. The nature of static power dissipation indicates that it is independent of switching activities and is "static" all the time. Thus, if the total time needed by a specific job can be considerably reduced, the amount of static energy can also be saved. There are some techniques which attempted to reduce static power

dissipation at system level, as is illustrated in the following.

<u>A) Pipelining</u>

Pipelining saves energy in a straightforward way. When using pipelining, it significantly reduces the overall execution time of a certain program. As a result, the time of leakage flowing is also reduced, thereby leading to a reduction in leakage-induced static power dissipation.

- N. S. Kim et al. (2003)

  In the work of [42], N. S. Kim et al. compared the overall power dissipation of pipelined systems with that of series systems, and concluded that "pipelining's combined dynamic and static power leakage will be less than that of the serial case". Thus, their conclusion has proven that pipelining can reduce the leakage-induced static power dissipation.

The design presented in this section has focused on using pipelining to reduce static power dissipation at the system level. As showed in the above example, pipelining is helpful to reduce the static leakage time and consequently achieve energy reduction. Therefore, although pipelining usually is used for improving the performance of processors, it also is an effective method to reduce static energy consumption.

<u>B) Phase Switching</u>

In general, modern day microprocessors are designed for the best performance. However, such best performance is not always needed in most applications. If certain

periods of an application can be identified as "standby" or "dormant", many circuit level techniques can be applied to significantly reduce the leakage power. Then, identifying such phases in applications is a system level effort toward low power design. Some examples by using phase switching to reduce static power dissipation are presented in the following.

- M. Powell et al. (2000)

  In [21], the authors found that there is a large variability in active cell usage both within and across applications. Thus, by using Gated-V_{DD} Caches, they proposed to identify phases with unused SRAM cells and gate their supply voltage and reduce their leakage. Their results indicated that it highly reduces leakage-induced power dissipation with minimal impact on performance.

- E. Rohou et al. (1999)

  E. Rohou et al. in [43] presented an adaptive approach that used feedback information to identify jobs in some phases which consume less power, and then switch phase contexts to manage processor temperature and reduce the leakage-induced static power dissipation. Their technique was implemented in the operating system so that it can both access hardware statistics and control the interleaving of processes. Results showed that their method could significantly reduce the static power dissipation with little cost in performance.

The designs presented in this section have focused on using phase switching to reduce static power dissipation at the system level. As known, the functioning of

certain applications can be divided into various phases in which the processors can be of different level of activity. Therefore, identifying these phases helps in minimizing the static power dissipation. Designs discussed above attempted to switch the processor setting according to phases with different level of activities. Usually, the phase switching design is combined with other schemes, for example DVS, to reduce the static power dissipation.

In summary, many low-power techniques, varied from the fabrication engineering level to the system design level, have been proposed to address static power dissipation. However, there is a trade-off among product cost, system complexity and power saving when applying these static power reduction techniques discussed above. Therefore, careful designing is needed for static power dissipation optimizations. Even though we do not target the leakage reduction in our research work presented in this thesis, it is also important to know that there are so many techniques which could be combined to further reduce the overall power dissipation of a microprocessor.

2.2 Dynamic Power Dissipation

2.2.1 Dynamic Power Dissipation Sources

For many years, efforts toward power reduction are mostly focused on reducing dynamic power dissipation, due to the extensive use of CMOS technology where leakage-induced power dissipation in the static state is many orders of magnitude smaller compared to power dissipated in dynamic switching of states. In general, dynamic power dissipation of microprocessors mainly arises from two circuit sources:

1) transient short-circuit current; 2) repeated charging and discharging of capacitive loads.

2.2.1.1 Transient short-circuit current

The short-circuit current is incurred due to transient conduction in both the pull-up and pull-down circuits in the CMOS circuit. Because such transitions can not realistically be instant, it is possible that the shut-off network is turned on before the previously turned-on network is shut off. However, as is discussed in [42] and [44], this transient short-circuit current is not significant in most circuits, and thus it is often ignored.

2.2.1.2 Repeated charging and discharging of capacitive loads

The major dynamic power dissipation comes from the charging and discharging of the state-keeping nodes. A low-to-high state transition corresponds to the charging up of all the capacitors associated with that node; while a high-to-low transition corresponds to the discharging of the node. With scaled feature sizes in modern transistors, the capacitance per unit area increases, accompanied by the increased switching frequency. Therefore, these trends lead to significant dynamic power dissipation in modern-day processors.

2.2.1.3 Dynamic Power Dissipation model

In the conventional process technology, the dynamic power dissipation involved in the switching is estimated by

$$P_{dynamic} = \alpha \bullet C_L \bullet V_{DD} \bullet \Delta V \bullet f_{CLK} \tag{2.2}$$

Where α is a constant of average activities and less than 1, C_L is the load capacitance involved, V_{DD} is the supply voltage, ΔV is the swing of voltage between two states and f_{CLK} is the switching frequency. For a normal switching in a CMOS circuit, the swing range is the full supply voltage. Supposing an amount of work that takes N clock cycles to finish, the time to finish the work is given by

$$T = \frac{N}{f_{CLK}} \tag{2.3}$$

Furthermore, as is presented in [41], the maximum clock frequency achievable shows a nearly linear dependence upon the supply voltage, which is illustrated in Figure 2.4 below.

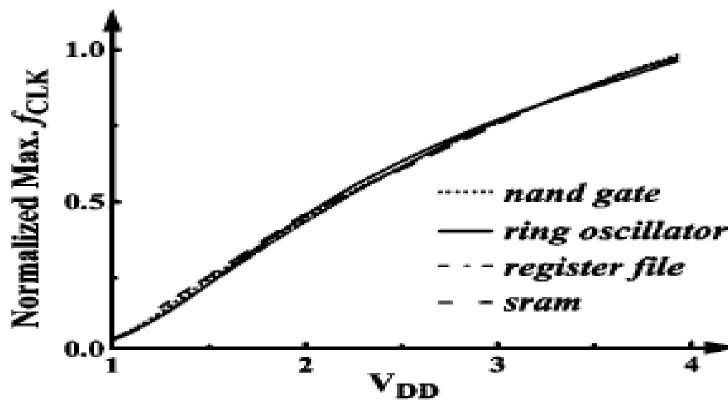


Fig. 2.4: Maximum Clock Frequency vs. Supply Voltage [41]

Thus we can approximately put:

$$f_{CLK} = k \bullet V_{DD} \tag{2.4}$$

As a result, the dynamic power can be estimated by:

$$P_{dynamic} = (\alpha \bullet C_L \bullet k) \bullet V_{DD}^3 \tag{2.5}$$

Obviously, the supply voltage has a very strong effect on the dynamic power dissipation. This leads to the wide-spread employment of voltage scaling techniques

to reduce dynamic power dissipation.

2.2.2 Dynamic Power Dissipation Reduction

In this section we review the low power techniques that target dynamic power dissipation. From the design strategy, these techniques are also grouped into either circuit-level or system level.

2.2.2.1 Circuit-level Techniques

As shown in the previous section, the dynamic power dissipation can be modeled by:

$$P_{dynamic} = \alpha \bullet C_L \bullet V_{DD} \bullet \Delta V \bullet f_{CLK} \tag{2.6}$$

As we can see from the above formula, it is natural to think of reducing the voltage swing (ΔV) and supply voltage (V_{DD}) to minimize the dynamic power dissipation. In general, the voltage swing can be reduced by the use of low-swing signaling, while the supply voltage can be reduced by the use of dynamic voltage scaling. We will detail the two low-power schemes in the following section.

A) Low-swing Signaling

As is discussed in the above, a straight-forward method to achieve dynamic power reduction is to reduce the signal swing. As known, low-swing technology provides high speed and low power at the same time. Instead of driving signals rail-to-rail, special drivers allow reduced signal swing. This may directly result in linearly reduced dynamic power, as expressed by the above equation. At the same time, the time needed to charge or discharge a node is also reduced, enabling faster state

switching. In the following, we will present some research work using this technique to reduce dynamic power dissipation of microprocessors.

- T. Sakurai et al. (1997)

In 1997, T. Sakurai et al. [45] described some circuit level techniques for low-power CMOS designs. In particular, the authors discussed the low swing signaling technique, and presented its applications to a clock system, logic part, and I/O's. They concluded that the low swing signaling technique is useful to reduce dynamic power dissipation.

- H. Zhang et al. (2000)

In the study of [46], H. Zhang et al. reviewed a number of low-swing on-chip interconnect schemes and presents a thorough analysis of their effectiveness and limitations, especially on energy efficiency and signal integrity. After that, they proposed several new interface circuits which employed low swing signaling, and achieved more energy savings and better reliability in experiments than former schemes.

- F. Worm et al. (2002)

In [47], F. Worm et al. introduced and showed the results of a interconnect system using low-swing signaling, which minimized the interconnect voltage swing and frequency subject to workload requirements and S/N conditions. Results showed that their scheme can attain tangible savings in energy, at the same time, achieving more robustness to large variations in actual workload,

noise, and technology quality.

- W. Jeong et al. (2004)

Recently, in 2004, W. Jeong et al. [48] proposed an adaptive supply voltage technique for low swing interconnects. To implement a low swing signaling design, their proposed technique assigned different supply voltages to drive interconnects based on their delay. Simulation results showed that their design could obtain very high power saving.

The designs presented in this section have focused on using the low-swing signaling technique to reduce dynamic power dissipation at the circuit level. As found by researchers, current-mode low-swing signaling techniques provide an attractive alternative to conventional full-swing voltage mode signaling in terms of delay and power dissipation [49, 50]. All these example designs presented here showed that low-swing technology is very useful to minimize the dynamic power dissipation, and provides both high speed and low power. For example, the low-swing signaling technique is already employed in the arithmetic core of Pentium 4 Processors [51].

B) Dynamic Voltage Scaling

Dynamic Voltage Scaling (DVS) is by far the most popular technique in use to reduce dynamic power dissipation. As is deducted in Section 2.2.1, dynamic power has a cubic relationship with the supply voltage in conventional CMOS circuits, while the maximum clock frequency is approximately proportional to supply voltage. Thus, supply voltage reduction, which usually implies a frequency reduction, could produce

a significant power saving. Over the past years, many researchers worked in DVS to reduce dynamic power dissipation of computer systems, which will be presented in the following.

- T. Ishihara et al. (1998)

  In [52], T. Ishihara et al. presented a theoretical study on dynamic voltage scheduling. In their work, they set up a model of dynamically variable voltage processor and analyzed it for power/energy reduction. Eventually, based on their model, they gave basic theorems for power-delay optimization of DVS.

- I. Hong et al. (1999)

  In 1999, I. Hong et al. [53] developed a design methodology for the low power core-based system, based on dynamically variable voltage hardware. Their synthesis technique addressed the selection of the processor core and the determination of the instruction and data cache size and configuration so as to fully exploit dynamically variable voltage hardware, which resulted in significantly lower power dissipation for a set of target applications than existing techniques. As they showed, their approach was effective in a variety of modern industrial-strength multimedia and communication applications.

- K. Flautner et al. (2001)

  In [54], the authors described a software approach to automatically control dynamic voltage scaling in order to optimize energy use, which was implemented in the Linux kernel and required no modification of user programs.

Their method worked equally with irregular and multi-programmed workloads and ensured that the quality of interactive performance is within user specified parameters. Their experiments showed a good result of high energy savings and only a minimal impact on the user experience.

- A. Azevedo et al. (2002)

In 2002, A. Azevedo et al. [55] proposed an intra-task DVS technique under compiler control using program checkpoints. Their defined checkpoints, which carried user-defined time constraints, were generated at compile time and indicated places in the code where the processor speed and voltage should be re-calculated. Checkpoints also carried user-defined time constraints. Their technique handled multiple intra-task performance deadlines and modulated power dissipation according to a run-time power budget. Results showed that their technique resulted in 82% energy savings over the execution of the program without employing DVS.

- K. Choi et al. (2005)

Recently in 2005, K. Choi et al. [56] presented an intra-process dynamic voltage and frequency scaling (DVFS) technique targeted toward non real-time applications running on an embedded system platform. Their DVFS technique relied on dynamically-constructed regression models that allow the CPU to calculate the expected workload and slack time for the next time slot, and thus, adjust its voltage and frequency in order to save energy while meeting soft

timing constraints. This was in turn achieved by estimating and exploiting the ratio of the total off-chip access time to the total on-chip computation time. Results showed that their scheme achieved very high CPU energy saving and low performance degradation for both memory-bound programs and CPU-bound programs.

The designs presented in this section have focused on using DVS to reduce dynamic power dissipation at the circuit level. DVS is the technique for exploiting this tradeoff whereby an appropriate clock rate and voltage is determined in response to dynamic application behavior. A number of DVS algorithms have been proposed to address power/energy optimization issues [57, 58, 59]. As known, DVS has been widely used in modern commercial chips such as Pentium 4 [51]. Furthermore, it is highly compatible with all kinds of circuit structures from memory to logics. It can also be combined with many other dynamic and static power reduction techniques to further minimize power dissipation. Currently, the key challenge is to develop effective DVS scheduling techniques that treat voltage as a variable to be determined, in addition to the conventional task scheduling and allocation. In the next chapter, we will discuss some algorithms for DVS and talk about the motivation for our low power design schemes using DVS.

2.2.2.2 System-level Techniques

At a higher system level, some techniques have also been proposed to reduce dynamic power dissipation. In general, these techniques all make use of system level information to reduce either the voltage swing or the supply voltage.

A) Signal Swing

As is well-known, in CMOS circuits, frequent signal switching consequently produce much voltage swing. Therefore, some researchers suggested reducing the signal swing to minimize the overall voltage swing.

- S. Haga et al. (2003)

Signal switching usually happens at the input port, output port and inside of the functional unit (FU). In [60], S. Haga et al. proposed a hardware method for functional unit assignment, based on the principle that a functional unit's power dissipation is approximated by the switching activity of its inputs. It dynamically assigned instructions to carefully selected Functional Units to minimize signal switching that happens in the FU. In their design, instructions are preferably issued to FU where the previous operands are similar to the current operands. This is illustrated in Figure 2.5.

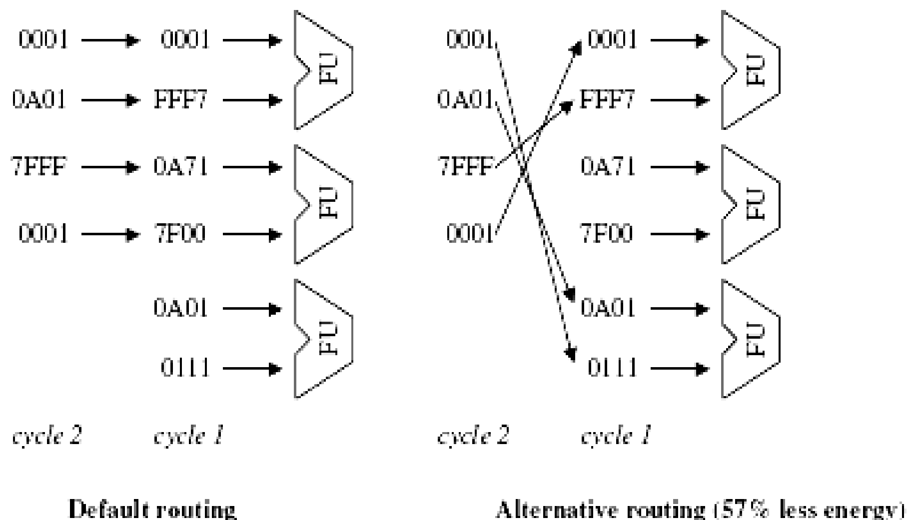


Fig. 2.5: Dynamic Functional Unit Assignment [60]

By using their design, voltage switching happening at the FU is reduced. The

simulation results showed a high reduction of switching activities in various FU and consequently achieved high power reduction.

The design presented in this section has focused on minimizing signal swing to reduce dynamic power dissipation at the system level. Some researchers proposed to use low-voltage swing circuits and techniques for microprocessors to reduce its power dissipation [61, 62]. As showed above, although the dynamic power dissipation is reduced, this is achieved at the price of extra hardware that carries out the comparing of the operands. Therefore, additional algorithms are needed to minimize the hardware cost.

B) State Switching

Scaling the supply voltage can considerably reduce the dynamic voltage at the price of slower execution speed. Thus, the best trade-off between power and performance can be achieved by switching between a spectrum of “active” and “standby” states. Therefore, some designs are suggested to reduce dynamic power dissipation by carefully switching states during system execution, which are presented in the following.

- W. Kim et al. (2002)

In [63], W. Kim et al. proposed a DVS algorithm for periodic hard real-time tasks based on an improved slack estimation algorithm. By deciding the state of different tasks and executing switches between states, their proposed method took advantage of the periodic characteristics of the real-time tasks under

priority-driven scheduling, such as EDF. Experimental results showed that their algorithm reduced the energy consumption by 20~40% over the other DVS algorithms. Their experiment results also showed that their algorithm based on state switching gave comparable energy savings to the DVS algorithm based on the theoretically optimal (but impractical) slack estimation method.

- A. Sinha et al. (2001)

In [64], A. Sinha and A. P. Chandrakasan proposed an adaptive approach to switch states of processors for dynamic voltage scheduling, based on workload prediction by filtering a trace history. In this work, a performance hit metric is defined and a state-switching technique to minimize energy under a given performance requirement is outlined. Their results demonstrated that up to two orders of magnitude energy savings is possible with dynamic voltage scheduling depending on workload statistics.

The designs presented in this section have focused on using state switching to reduce dynamic power dissipation at the system level. These approaches all lead to better power-performance results in microprocessors. In general, this state switching decision can be made by either hardware or software. If it is done by the hardware, additional hardware has to be added and consequently increase the hardware complexity.

2.3 Summary

In this chapter, both static and dynamic power dissipation are discussed. Firstly, we reviewed different sources for static and dynamic power dissipation. Following that, various existing techniques for static and dynamic power reduction have been discussed. In general, many of these static and dynamic power reduction techniques can be combined to minimize the overall power dissipation. For example, our low power design schemes presented in following chapters are going to combine phase switching, state switching and dynamic voltage scaling to reduce the overall runtime power dissipation of microprocessors. The detailed implementation of our schemes will be introduced later in this thesis.

Chapter 3

Motivation and Analysis Model

This chapter describes the motivation and the analysis model for our low-power design schemes presented in this thesis. It is structured as follows. In Section 3.1 we describe the motivation for our approaches. In Section 3.2, we first present the analysis model for our schemes, and then discuss the trade-off between power and performance in our low-power designs. In Section 3.3, we summarize this chapter.

3.1 Motivation

As is discussed in the previous chapter, Dynamic Voltage Scaling (DVS) is by far one of the most popular power reduction techniques in use. Basically, DVS is a technique that varies the supply voltage and frequency of a microprocessor to provide a desired performance with the minimum amount of energy consumption. As an efficient technique to reduce energy consumption, dynamic voltage/frequency scaling (DVFS) has already been used in many computer systems, such as those based on the Transmeta Crusoe[65] and Intel XScale[66] processors.

Various algorithms had been proposed to use DVS to match the changing demands for microprocessor processing speed to achieve reduction of runtime power dissipation. In general, these approaches analyzed system or program dynamic behaviors, adjusted microprocessor settings to better match its performance requirements, and consequently reduced system/application power dissipation. As

known, previous work has been studied exhaustively at the operating system (OS) level and the source code level to employ DVS to achieve power-performance trade-off optimization.

3.1.1 OS Level DVS Algorithms

Algorithms running at the OS level usually use heuristic methods to reduce power dissipation in response to variations in workload during runtime. In general, OS level runtime algorithms use prediction approaches to monitor the current system load and estimate the future demand by using interval-based schedulers with a time window. In 1994, M. Weiser et al. [67] suggested an interval-based predicting algorithm to dynamically scale the frequency and voltage of the processor. Their algorithm divided time into fixed-length intervals and set each interval's speed so that most work is completed by the interval's end. Thereafter, E. Chan et al. [68] in 1995 refined the idea by separating it into two parts: prediction and speed-setting. At the front of an interval, the prediction part estimates how busy the CPU will be during the interval, which is measured via the non-idle fraction of the last interval. The speed setting part uses this information to set the speed of processor.

As known, M. Weiser et al. are the pioneers of the DVS research by first designing the interval-based prediction DVS algorithm at the OS level. Since then, following their idea, many interval-based DVS algorithms [69, 70] have been proposed. Recently in 2008, Seo et al. [105] presented a time-slice based DVS algorithm to adjust processor performance at every context switch in order to match

the performance demand of the next scheduled task. In general, evaluations of these algorithms through simulations showed that they all achieved good results in power savings.

As discussed above, all these OS-based predictive algorithms aim to reduce power dissipation without degrading application performance. However, in a recent study in 2000, D. Grunwald et al. [71] used actual measurements and observed noticeable performance loss for some interval-based algorithms running at the OS level. One possible reason is the misunderstanding about future performance demand of CPU, and it could be caused by inaccurate predictions at the high OS level. As studied in [72], researchers found that the OS itself could have a strong impact on application performance sometimes; hence it will also affect the prediction results in these algorithms, thereby leading to wrong decisions by DVS scheduler and the accompanying performance penalty.

This observation motivates us to estimate the future performance demand of CPU at a lower micro-architecture level to avoid such misunderstandings, because at the micro-architecture level it relies solely on the hardware to identify current performance of CPU and thus obtain a more reliable prediction for future performance demand. Therefore, we propose a micro-architecture level identification and prediction algorithms in the following of this thesis.

3.1.2 Source-code Level DVS Algorithms

Beside DVS approaches employed at the OS level, researchers have addressed low-power

DVS designs at the source-code level, and proposed that applications themselves provide information about their future performance demands, such as deadlines of real-time tasks and other timing constraints.

For applications used in the real-time system, researcher usually utilized hard real-time constraints of tasks as their performance demands to direct DVS scheduling to reduce power dissipation of microprocessors. For example, J. Pouwelse et al. [73] proposed an approach for real-time applications: it indicated the required number of clock cycles (instructions) for the next deadline and allowed the CPU to compute with the lowest speed at which this application could also meet its deadline.

For non real-time applications, researchers suggest using mode-set instructions as performance indicators for the CPU to dynamically schedule its setting. In the work of [74], C. H. Hsu et al. proposed a compiler-based method to reduce the power dissipation in microprocessors. First of all, the authors identified single or multiple regions of a program, which may be memory-bound regions where the CPU can be slowed down without significant performance impact. Then, they employed a nonlinear optimization formulation to insert their defined power-down instructions at regular intervals in the program. Finally, they lowered the voltage and frequency of CPU when encountering these instructions during application execution. Similarly, in [75], M. Huang et al. implemented a profile-based method. It first identified the most frequently executed functions/modules of an application during its training runs, and then used a selection algorithm to choose the best low-power configuration for each module after profiling its runs with mode-set instructions. In both

methods, the researchers directed the voltage scaling of microprocessors by their defined indicators, and eventually reduced the energy consumed by microprocessors. Recently, Zhou et al. [106] in 2005 suggested a combination of optimal speed and limited preemption as mode settings for a dynamic task scheduling algorithms for periodic tasks that minimize the system-level energy; Malani et al. [107] in 2007 proposed a profile-based low power scheduling by using conditional task graph to differentiate different modes.

In [76], D. Mosse et al. presented a method that profiles program regions by mode-set instructions and simultaneously meets their hard time constraint--the worst case execution time (WCET) assumption when determining a voltage scaling schedule. Actually, this method combined the two approaches discussed above, integrating the mode-set indicator into the real-time applications that have hard deadlines, to direct DVS to decrease the total energy consumption.

All the above DVS algorithms running at the source-code level gained power-performance tradeoffs by using various performance indicators to direct DVS scheduling. During execution time of applications, an optimal processor setting, such as appropriate voltage and frequency, can be chosen to minimize its power dissipation and still meet application performance requirement. For all these source-code level DVS algorithms, their evaluations through experiments showed that they most often achieved good results with high power reduction.

However, in the above algorithms running at the source-code level, all the applications must be aware of their processing performance demands in advance. For

applications of real-time systems, there is no problem because they could use hard real-time constraints, which have been clearly defined in the source code, as performance indicators to direct DVS scheduling. But, for applications used in general-purpose systems, such performance demands are usually unknown in advance since these applications have no explicit deadline or other hard timing constraints.

This observation motivates us to employ the micro-architecture parameters, which could be obtained from execution results of non real-time applications, as performance indictors to direct DVS scheduling and thus void limitations of former source-code level algorithms. Therefore, in this thesis, we propose a code analysis and reconfiguration mechanism by using the micro-architecture parameters to direct DVS scheduling.

3.1.3 Our Micro-architecture Level DVS Algorithms

As discussed above, in this thesis, we propose two novel algorithms to schedule DVS at the micro-architecture level to achieve successful low-power designs for microprocessors.

A) Identification and Prediction Algorithm

To estimate a reliable future performance demand of microprocessors, we propose a micro-architecture level identification and prediction algorithm.

This algorithm employs a micro-architecture parameter, Instructions per Cycle (IPC), as the runtime performance indicator to dynamically scale the voltage and frequency of a processor. In this design, we obtain the micro-architecture parameter (IPC)

through hardware support in the processor, and use it to quantify the knowledge of the most recent performance requirements for each program interval and guide the voltage and frequency scaling. Thus, it avoids the impact of the OS on the prediction results in former OS level DVS algorithms. Therefore, the DVS decisions made by our micro-architecture level identification and prediction algorithm are more reliable, resulting in bigger energy reduction and smaller performance losses.

B) Code Analysis and Reconfiguration Algorithm

Using the micro-architecture parameters as performance indicators, we propose a code analysis and reconfiguration mechanism to direct DVS scheduling.

In this mechanism, we first identify code regions of an application that have low performance requirements and could make contributions to runtime power reduction. We utilize the micro-architecture parameter (IPC), which is obtained from execution results of application training runs, as the performance indicator to identify such code regions. Then, we profile these code regions to dynamically scale the voltage and frequency of the microprocessor during application execution.

In our design, the micro-architecture parameter (IPC) could be estimated from execution results of non real-time applications, thereby avoiding the limitations of former source-code level algorithms. Furthermore, based on execution results of application training runs, our mechanism practically identifies power-performance tradeoff opportunities in applications and thus avoid inappropriate switch points for DSV scheduling. Thus, our micro-architecture level code analysis and reconfiguration

mechanism is more reliable, thereby leading to more power reduction and less performance degradation.

However, for any profile-driven algorithms based on application training runs, a common question is how much the quality of the execution results is affected by the training runs.

To answer the above question, it is very important to identify how many factors affect the execution results of a program. Firstly, the results may be affected by the program source code. Different source codes to implement the same function in a program will lead to variations of the execution results. Secondly, the results may be affected by the compiler. Different compilers may have distinguishing strategies to optimize the program source code, resulting in differences in the binary files and the execution results. Thirdly, the results may be affected by the processor architecture implementations. For the same binary file of a program, the execution results may be quite different when executing on processor architecture with different implementations. Lastly, the results may be affected by different input data sets. The execution process of any application may also be data-sensitive, and different input data will lead to many differences in the execution results. Overall, there are many factors that can influence the execution results of an application. Therefore, it is quite difficult for profile-based methods to guarantee the capture all execution behaviors through training runs.

However, for these profile-based methods, some approaches could be applied to avoid or reduce these influences on the execution results, Firstly, to avoid the

affections from the source code and compilers, we can made use of the same binary file of applications in both training runs and later executions. Secondly, to avoid the affection from the processor architecture, we can perform the binary file on a processor with the same architecture implementation. Lastly, we can minimize the affection from input data sets by using good training inputs, which could well match the real instance of application execution and thus well capture the execution behavior of applications. As is well known, it is impossible to eliminate the affection caused by the input data sets since different input files have to be used between the training runs and later executions.

In our code analysis and reconfiguration mechanism, we applied the above approaches to avoid or diminish such affections. For this aim, we chose the well-developed benchmarks (SPEC [77] and MediaBench [78]) in our experiments. In both the training runs and the subsequent actual executions, we performed the same object files, which are supplied by the benchmark suits, on a processor with the same architecture. Thus, we avoid the first three factors. Furthermore, in the benchmark suits, the reference inputs for training runs can well capture the execution behaviors of applications. Thus, we well identified power optimization opportunities in applications from the results of training runs, which is proved by the good results in our experiments.

Overall, our proposed micro-architecture level low-power designs in this thesis could reduce the power dissipation of microprocessors, at the same time, with little performance degradation. Results of our designs show that it is more efficient to

address low-power optimizations at the micro-architecture level.

3.2 Analysis Model

As is discussed in the previous section, the primary idea of our low-power design schemes is an attempt to identify power reduction opportunities in an application where a possible CPU slowdown will not significantly affect the performance. In the following, we will first present a realistic model for our low-power designs. Then, based on the model, we will analyze potential code regions in an application which have such possible opportunities. Lastly, we will discuss the constraints and the resulting penalty for using the model in our low-power strategies.

To derive our analysis model, first of all, we make the following assumptions about the application and the micro-architecture implementation of microprocessors:

- ✓ The logical behavior of an application does not change with the voltage and frequency scaling of microprocessors.
- ✓ Peripheral components, such as I/O and memory, are asynchronous with the microprocessor.
- ✓ The clock is gated when the processor is idle.

3.2.1 Basic Model

Since the goal of our designs is to identify power reduction opportunities in an application, this model is going to define and analyze the problem from the aspect of program code.

Firstly, we shall identify different types of code regions in a program, corresponding

to various operations related to CPU. In general, any program is a union of code regions. In our model, an entire program is divided into two major code regions: CPU computing operation region and peripheral operation region. For the CPU computing operation region, it again can be divided into two sub-sets: parallel computing operation region, which can run concurrently with the peripheral operations; dependent computing operation region, which has to be pended and wait for the results of peripheral operations.

For the above three operations in a code region, we make use of three corresponding parameters (W_{di}, W_{pi}, W_{mi}) to quantify their workload (in cycles). In the following, we will describe the detailed definitions for the three parameters employed in our model:

W_{mi}: the number of execution cycles cost in region R_i where peripheral operations are executed, including memory-bound and I/O-bound operations. For memory-bound operations, both cache hit and cache miss operations are taken into account.

W_{pi}: the number of execution cycles cost in region R_i where CPU computing operations can run in parallel with peripheral operations, which means both operations are active at the same time during execution.

W_{di}: the number of execution cycles cost in region R_i where the CPU computing operation is stalled while waiting for data from peripheral operations, which means that is dependent on peripheral operations.

Then, the total workload for code region R_i is defined as $W_i = W_{di}+W_{pi}+W_{mi}$,

and the total workload for program *P* is defined as $W = \sum_i W_i$.

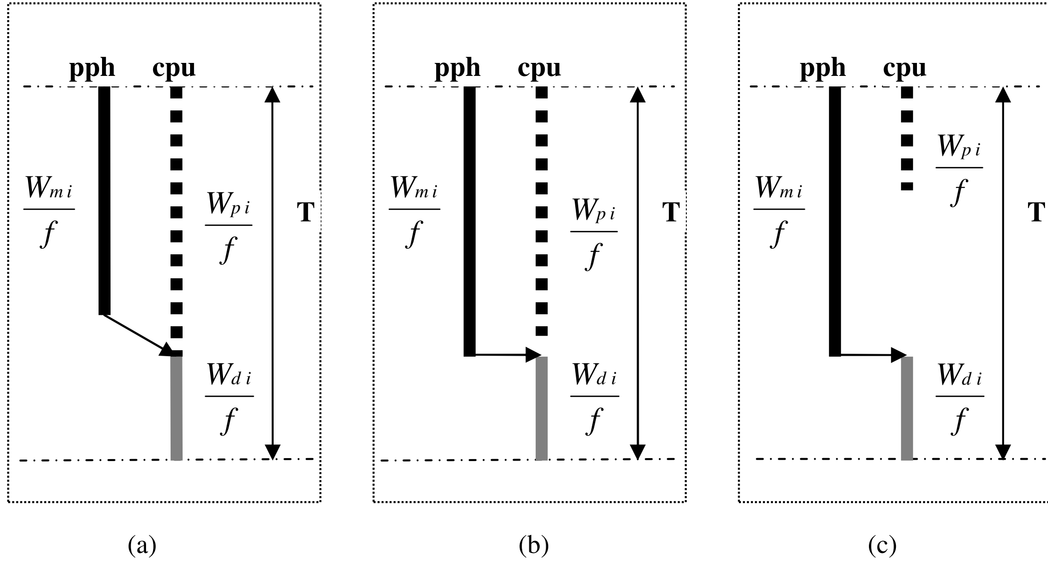


Fig. 3.1: Possible overlaps in peripheral and CPU computing operations.

As shown in Figure 3.1, we can illustrate the execution of any code region in a program within the three cases, corresponding to different relative workload of the above defined three operations. In these figures, "*T*" is the execution time to complete the code region, "*pph"* is the peripheral operations, "*CPU"* is the CPU computing operations, and "*f*" is the frequency of a processor.

In the case shown in Figure 3.1(a), the parallel CPU computing operations dominate the overlap part, and they take relatively longer time than the peripheral operations. Thus, the execution time of the overlap part is determined by the parallel computing operations. For a code region in this case, its total execution time is decided by the time for CPU to finish the computing operations, and could be expressed as $W_{pi}/f + W_{di}/f$. Thus, for this case, we define it as a computation-dominated case.

In the case shown in Figure 3.1(b), the peripheral operations dominate the overlap part, but the time cost by the peripheral operations nearly equals that of the parallel

computing operations. Thus, the execution time of the overlap part is determined by the peripheral operations, and the total execution time of the code region is decided by the peripheral operations and the dependent computing operations. In this case, the total execution time expression is $W_{mi}/f + W_{di}/f$. Since there almost is no CPU idle time in this case, we define it as a peripheral-dominated case without slack.

In the case shown in Figure 3.1(c), obviously, the peripheral operations dominate the overlap part, and the time cost by the peripheral operations is much more than that of the parallel computing operations. Thus, the execution time of the overlap part is absolutely determined by the peripheral operations. In this case, the total execution time is also decided by the peripheral operations and the dependent computing operations, and its expression is $W_{mi}/f + W_{di}/f$. Since there is much CPU idle time in this case, we define it as a peripheral-dominated case with slack.

3.2.2 Basic Analysis

The aim for our low-power design schemes is to identify individual code regions in an application which has possible power reduction opportunities, and then scale these regions to be run at lower frequency and voltage to reduce power dissipation of CPU without affecting their performance very much. Therefore, based on the above model, it is important for us to figure out the following questions: what are the potential code regions having such power reduction opportunities? How to achieve the power reduction and under what constraints? The answers to these questions will help provide insight into what kinds of program code regions are likely to benefit from our

designs and under what scenarios.

A) Potential Code Regions

As we can see from the above analysis, the difference among the three cases shown in Figure 3.1 depends on the balance of the overlap part: the parallel computing operations and the peripheral operations. Thus, for any one of above three cases, the total execution time for a code region can be expressed by one uniform formula:

$$T_i = \max \left(W_{mi} / f, W_{pi} / f \right) + W_{di} / f \qquad (3.1)$$

From the above formula, it is obvious to see that: for a code region, when executed on a processor with a fixed frequency (f), if the peripheral operations (W_{mi}) take much more time than the parallel computing operations (W_{pi}), there will be much slack, i.e. CPU idle time, during the execution of a code region, which means there are possible opportunities for us to achieve power reduction. For example, as shown in Figure 3.1(c), there is a quite long period of CPU idle time because the peripheral operations dominate the overlap part, and spend more time than the parallel CPU computing operations. Thus, in this case, the parallel CPU computing operations could be run at a lower frequency and voltage to reduce power dissipation of the processor.

Therefore, these code regions, in which the peripheral operations dominate the overlap part, are the potential regions which have possible power reduction opportunities and will benefit from the power-performance trade-off optimization. As is well-known, during the execution period of a code region, peripheral operations, including I/O-bound operations and memory-bound operations, usually spend much time to be finished. Thus,

researchers proposed to utilize this observation for low-power designs. For example in [43], cache miss rate was used to direct the processor to scale its supply-voltage and eventually reduce power dissipation of processors. In our model, not only memory-bound operations but also I/O-bound operations are taken into account. Thus, we will identify more power reduction opportunities and achieve more power saving by using this model.

B) Constraints

After identifying these potential code regions which have possible power reduction opportunities, our approach is trying to scale these regions as if they could be run at a certain lower frequency and voltage to reduce power dissipation of CPU.

In general, if a code region R_i is slowed down by a factor of δ, the resulting performance is

$$T_i(\delta) = \delta * W_{di} / f + \max(W_{mi} / f, \delta * W_{pi} / f) \quad (3.2)$$

However, when slowing down CPU to execute these identified code regions to reduce power dissipation, our designs attempt to not significantly affect their overall execution time. Thus, to achieve a successful power-performance trade-off optimization, our low-power designs are subject to the following two time constraints:

1) $\delta * W_{pi} / f \le W_{mi} / f$: When slowing down CPU by a factor of δ, for a code section, the time cost by the parallel computing operations ($\delta * W_{pi} / f$) should be less than that of the peripheral operations (W_{mi} / f). This time constraint is hard to meet since the workloads of both peripheral operations and parallel computing operations in a code

region are uncertain.

2) $T_i(\delta) \le (1+r)T_i$: When slowing down CPU by a factor of δ, the total execution time of a code region ($T_i(\delta)$) should be less than the original execution time plus a user acceptable performance penalty ($(1+r)T_i$), where r represents the user acceptable performance penalty ratio and its range usually is defined from 1% to 10%. For non real-time applications used in general-purpose systems, they often do not have such hard timing constraints, and it is necessary for us to define a soft time constraint ($(1+r)T_i$) for them to guarantee their performance not to be degraded very much. In our experiments, we employ the soft time constraint as a criterion to evaluate the results.

C) Resulting Penalty

As is discussed above, when slowing down CPU to achieve power dissipation reduction during the runtime of identified code regions, our approach is trying to hide the degraded performance behind the peripheral accesses. Nevertheless, we could make use of the three parameters (W_{di}, W_{pi}, W_{mi}) to estimate the performance impact of the CPU slowdown for a code region.

In the three cases shown in Figure 3.1, we are only interested in the peripheral-dominated case with slack since it have possible power reduction opportunities, and the resulting performance for it will become

$$T_i(\delta) = \delta * W_{di} / f + W_{mi} / f \quad (3.3)$$

As a result, a performance penalty of $\delta * W_{di} / f$ will occur if the entire $\delta * W_{pi}$ workload can be hidden behind the peripheral activity workload (W_{mi}). However, if only a partial hiding is possible, an additional performance penalty will be accounted for.

3.3 Summary

In this chapter, we first described the motivation to our low-power design schemes using DVS at the micro-architecture level. Then we presented the analysis model for our designs and use it to direct our low-power designs. Based on our analysis model, in the following chapters, we proposed three low-power design schemes that make use of micro-architecture parameters to identify power reduction opportunities in code regions and dynamically schedule the voltage and frequency of microprocessors during flavor regions execution having such opportunities.

Chapter 4

Infrastructure

This chapter describes the benchmarks and simulation tools used in our experimentations in this thesis. It is structured as follows. In Section 4.1, we shall describe the selected benchmarks. In Section 4.2, we shall first describe the simulator used in our experiments, and then we will list changes made by us in the simulator, finally we shall present the method by which power/energy results were obtained. In Section 4.3, we will summarize this chapter.

4.1 Benchmarks

In this thesis, to evaluate our low-power design schemes, different types of benchmarks are performed in our experiments. First of all, four benchmarks are chosen from the Standard Performance Evaluation Corporation (SPEC) CPU2000 suite [77], which is the standard for research into microprocessor architecture optimizations for general-purpose systems. Then, other four benchmarks are chosen from the MediaBench [78] suite, which is the standard for research into embedded application for embedded systems, such as multimedia and communication systems. Overall, our low-power design strategies in this thesis are verified by benchmark applications from both general-purpose systems and embedded systems.

In our experiments, all these benchmarks were run with the reference input

workloads supplied by the two benchmark suites. Table 4.1 lists our chosen benchmarks and their descriptions.

Benchmark	*Description*
176.gcc	C Programming Language Compiler
181.mcf	Combinatorial Optimization
197.parser	Word Processing
255.vortex	Object-oriented Database
unepic	Image decompression
epic	Image compression
adpcmencode	Differential audio pulse coding
adpcmdecode	Differential audio pulse decoding

Table 4.1: Summary of selected benchmarks

4.2 Simulation Environment

In this section, we shall present the simulator used in our experiments. Firstly, the simulator architecture is described, and then the changes made in the simulator are discussed. Following that, the power/energy model of the simulator is presented, from which the power/energy results for applications were obtained.

4.2.1 The Simulator

In our experiments, all benchmark applications were run on the Wattch [79] tool, which is based on the SimpleScalar [80] simulator. The Wattch tool was chosen because it can model a high-performance superscalar processor and also provide detailed energy/power results. Moreover, it is the standard environment for research in

the area of micro-architecture design for microprocessors. As known, Wattch is widely used by researchers of today in the area of low-power technique designs for microprocessors. In this thesis, to evaluate our designs, we employed the Wattch tool to perform our chosen applications and collect their execution results.

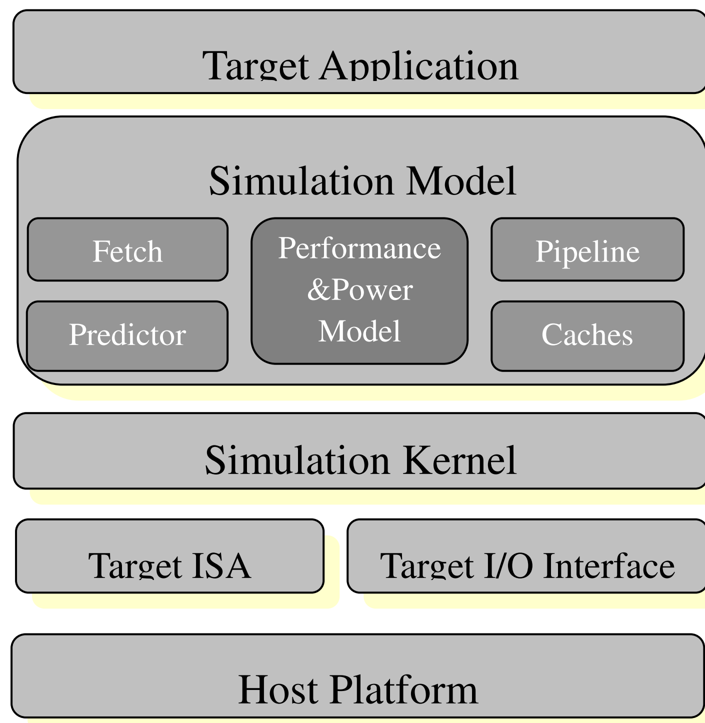


Fig. 4.1: Architecture of the Wattch Simulator

Figure 4.1 presents the detailed architecture of the Wattch simulator. As is shown in the above figure, the Wattch simulator has to be run on a host platform, which could be Linux or Windows platform. Then, Wattch implemented its simulation kernel based on self-defined target ISA and target I/O interface. In Wattch simulator, corresponding to various micro-architectural components inside microprocessors, such as predictor, caches and pipeline, there are specific simulation models to implement the functions of these components. Among these models, the performance & power model is one of the most important simulation models, because it is with the

responsibility for producing the final performance and power results when a target application is executed on Wattch simulator.

However, in order to carry out our low power designs at the micro-architecture level, some changes had to be made in Wattch simulator. Our modifications to Wattch not only provide runtime performance sampling, but also support dynamic voltage and frequency scaling (DVFS). Furthermore, it is important to know that the voltage and frequency transition of a microprocessor has a cost in terms of both time delay and energy consumption. Thus, the impact of DVS cost should be taken account into the total execution time and power/energy data of our simulation results. As an important modification to the Wattch simulator, referring to [81], we modeled the DVS cost with a transition time of 12 μs and a transition energy of 1.2μJ. In this thesis, we are going to make use of the slowdown of microprocessor and achieve power dissipation reduction. Thus, in our simulations, we consider a kind of scaling between a slow frequency and voltage and a normal frequency and voltage. To implement DVFS into Wattch simulator, we assume that the microprocessor has two scaling levels for (v, f): a slow frequency of 150MHz at 1.1v and a normal frequency of 600MHz at 1.6v.

In Wattch simulator, the configuration of the processor was designed to be typical of a high-performance superscalar supporting 8-wide fetch, decode, issue and commit. The details of the configuration can be seen in Table 4.2, which lists the micro-architectural parameters of the baseline model of Wattch simulator.

Parameter	*Configuration*
Memory	150 cycle round trip access
Virtual Memory	8K byte pages, 30 cycle fixed TLB miss latency after earlier-issued instructions complete
Architecture Registers	32 integer, 32 floating point
Instruction Cache	8k 2-way set-associative, 32 byte blocks, 1 cycle latency
Data Cache	16k 4-way set-associative, 32 byte blocks, 2 cycle latency
Unified L2 Cache	1Meg 4-way set-associative, 32 byte blocks, 20 cycle latency
Branch Predictor	hybrid - 8-bit share w/ 8k 2-bit predictors + a 8k bimodal predictor
Out-of-Order Issue	out-of-order issue of up to 8 operations per cycle, 128 entry re-order buffer
Functional Units	8-integer ALU, 4-load/store units, 2-FP adders, 2-integer MULT/DIV, 2-FP MULT/DIV

Table 4.2: Wattch Baseline Simulation Model

4.2.2 Energy Measurements

In this section, we shall discuss the model used to measure energy/power data with the Wattch simulator, which is presented in the previous section.

As is found by many researchers, it is a very difficult task to estimate detailed power dissipation or energy consumption within a superscalar processor. Over the past years, several models and tools are proposed to accomplish this task, but Ghiasi and Grunwald found experiment results of these models varied dramatically [82]. To solve this problem, in the study of [83], N. S. Kim et al. suggested that it can be done by a cycle-accurate simulator to produce meaningful energy/power results. Following this suggestion, D. Brooks et al. designed the Wattch tool, which makes use of the cycle-accurate simulator of SimpleScalar as a base, and successfully implements a framework to estimate detailed

power/energy data at the micro-architecture level.

In this thesis, we make use of the dynamic and static power/energy model from the Wattch tool to obtain the detailed power/energy results for benchmark applications. In Wattch simulator, there are three steps to accomplish its dynamic and static power/energy model, which is described in the following.

1) As a starting point, the Wattch simulator employs the power/energy model of Cacti [84] tool, which is for calculating the power/energy data of memory circuits.

2) Then, Wattch simulator models all the major components of real microprocessors. In general, the main units of microprocessors for the energy/power estimation model used in Wattch fall into four categories:

   - Array structures, including data and instruction caches, cache tag arrays, all register files, register alias table, branch predictors, large portions of the instruction window and load/store queue.
   - Fully associative content-addressable memories, including instruction window/reorder buffer wakeup logic, load/store order checks and TLBs.
   - Combinational logic and wires, including functional units, instruction window selection logic, and dependency check logic and results buses.
   - Clocking, including clock buffers, clock wires, and capacitive loads.

3) Finally, for these components, Wattch simulator parameterizes their power/energy data as accurate as possible, and integrates these power/energy estimations into a high-level simulator.

Basically, the power/energy model of Wattch uses SimpleScalar's hardware configuration parameters as inputs to compute the power/energy data for the various units in the processor. Furthermore, during execution, the Wattch simulator keeps track of which units are accessed per cycle, thus the power/energy model of Wattch could eventually calculate the detailed power/energy results for an application.

Overall, the Wattch simulator demonstrates a fast, usefully-accurate, high-level energy/power measurement model.

4.3 Summary

This chapter has described the infrastructure used in this thesis. In our experiments, benchmarks were chosen from both the SPEC CPU2000 benchmark suite and the MediaBench suite, and then they were simulated using an adapted version of Wattch and SimpleScalar tools, to obtain their performance and power/energy results.

Chapter 5

IPC-Driven Online Power Reduction Method

Nowadays, with fast advances in CMOS technology, power dissipation has already become a great concern in microprocessor design, not only for battery-operated potable devices but also for high-end computer systems due to excessive cooling and power costs. Therefore, various techniques are employed by modern microprocessor designers to address the power reduction issue. As known, dynamic voltage scaling (DVS), which is a technique to vary the supply voltage and frequency of CPU to provide desired performance with the minimum amount of energy consumption, has been identified as one of the most effective ways to reduce energy consumption.

In the past years, many algorithms have been proposed to use DVS to match the changing demands for processing speed and achieve power reduction. Based on when the decisions to switch voltage/frequency are made, they can be broadly classified as compile-time and run-time algorithms.

At compile time, researchers suggested the use of mode-set instructions as indicators to schedule DVS. Hsu et al. [85] defined some power-down instructions for memory-bound regions and provided a non-linear optimization formulation to insert such instructions during program compiling period for optimal DVS scheduling.

Run-time DVS algorithms have received much attention because of the ability to

reduce power dissipation in response to variations in workload. Generally, runtime algorithms are studied at the operating system (OS) level and the micro-architecture level. Algorithms at the OS level usually use heuristic scheduling, including interval-based algorithms like Lorch's proposal [69] and task-based algorithms like Luo's work [70]. However, because of the misunderstanding about future performance demand of CPU caused by inaccurate predictions at the high OS level, researchers suggested scheduling DVS at the lower micro-architecture level to avoid such problems and achieve reliable predictions. For example, in [86], Marculescu proposed the use of cache miss rate to direct DVS by hardware support during execution time.

In this chapter, we introduced Instruction per Cycle (IPC), a micro-architecture level parameter, as the performance indicator for processor runtime period and implemented an interval-based identification and prediction mechanism for reducing processor power dissipation without much performance degradation. The basic idea of our mechanism is to trace the current interval's performance activity level and predict the coming interval at which certain power-performance trade-off would be profitable. Our simulation results of real workloads showed that our approach takes advantage of energy reduction as well as provides fine-grained, tight control over performance loss. The low-power design presented in this chapter is built upon my earlier work in [102].

The rest of this chapter is organized as follows: Section 5.1 gives the motivation of using IPC as the performance indicator in our low power design schemes. Section 5.2 describes the detailed steps of this method. Section 5.3 demonstrates our simulation results and related discussions, and we summarize this chapter in Section 5.4.

5.1 Motivation for IPC Indicator

5.1.1 IPC variations

When applications execute, they regularly show changes in some parameters, such as IPC (Instructions per Cycle). These changes can vary widely during application execution period. Figure 5.1 shows the IPC variations when running "gcc" on the SimpleScalar tool. The IPC value is estimated by measuring how many cycles spent in an execution interval with a fixed number of instructions (100K in this experiment). In Figure 5.1, the IPC value shown on the Y coordinate varies largely even in the very short execution periods.

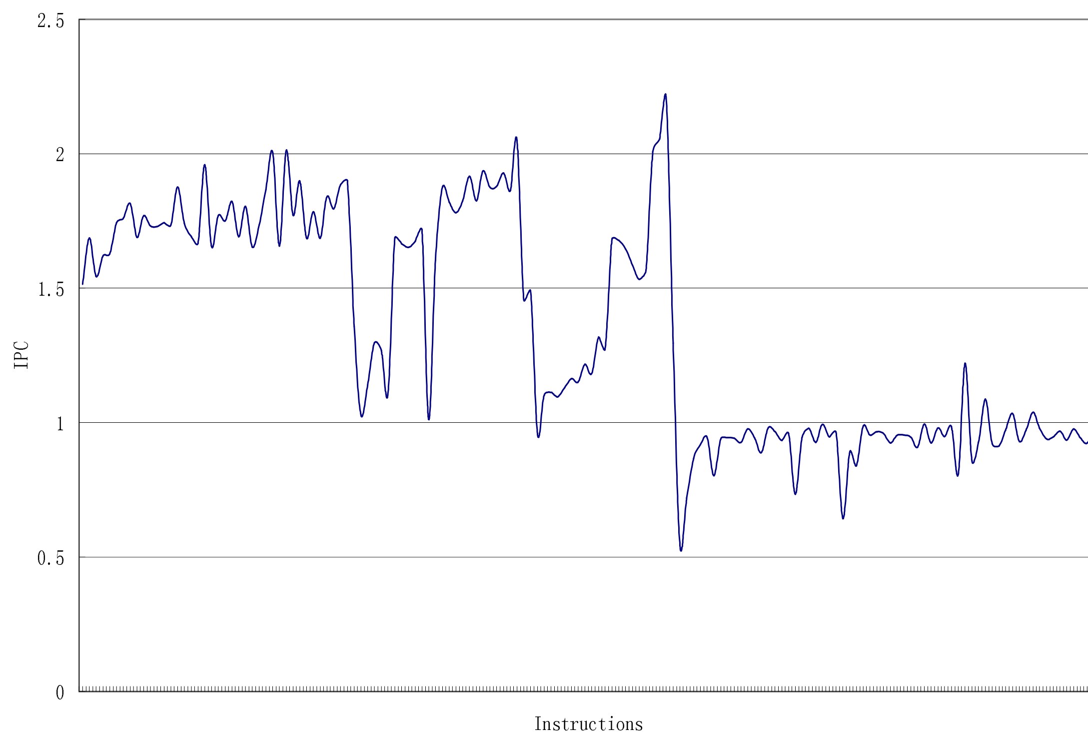


Fig. 5.1: IPC variations during a short execution period of "gcc".

Moreover, as noted by some researchers, this variability of IPC can be extended to other resources of the microprocessor. For example, D. W. Wall [87] investigated the

range of IPC values in direction to the depth level of instruction parallelism within a single application. D. Albonesi [88] also noted the similar characteristic of IPC in applications and used it in an IPC/clock rate tradeoff design. In our method, we shall employ IPC variations to indicate different performance requirements of application runtime periods to implement a power-performance tradeoff design.

5.1.2 IPC Indicator

As is well-known, for a task of an application, the most straightforward way to quantify the performance requirement is to make use of its required execution time. When executed by a processor, the CPU time (T) needed to finish a task is estimated by

$$T = \frac{Workload}{Speed} = \frac{N_i}{IPS} = \frac{N_i}{IPC \times f} = \frac{N_i \times \tau}{IPC} \tag{5.1}$$

where *IPS* is instructions per second. From the above formula, a new formula for IPC can be deduced as the following:

$$IPC = \frac{N_i}{N_c} = \frac{N_i \times \tau}{T} = \frac{N_i}{T \times f} \tag{5.2}$$

where N_i is the number of instructions in the task, N_c is the amount of cycles, τ is the clock cycle time, and f is the frequency of a processor.

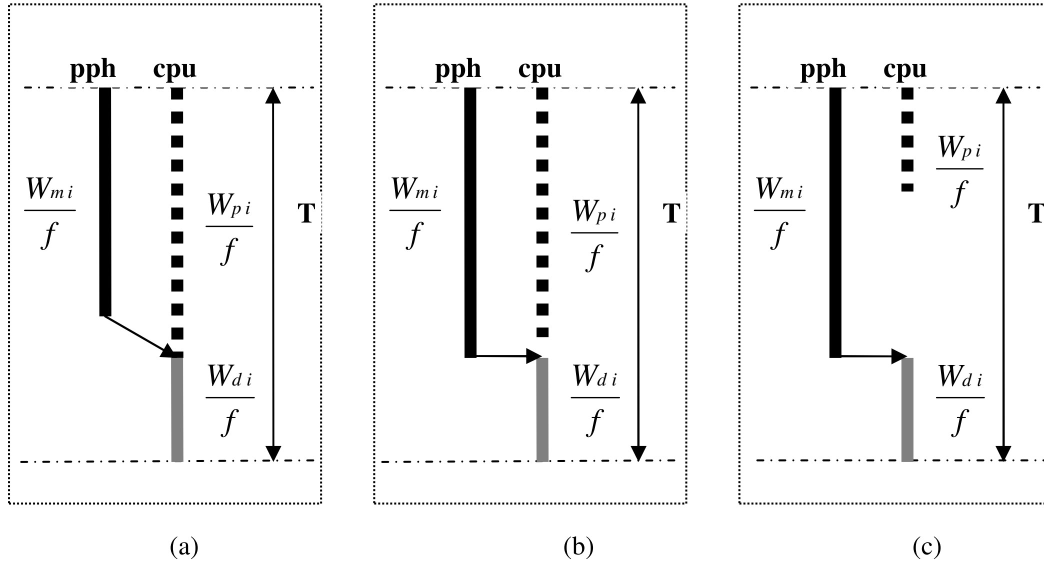


Fig. 5.2: Analysis model

In Chapter 3, from the analysis model shown in Figure 5.2, we derived the formula for the execution time of a task, which is given as:

$$T = \max\ (W_{mi}/f\ , W_{pi}/f\ ) + W_{di}/f \qquad (5.3)$$

where W_{mi} is the peripheral operations' workload, W_{pi} is the parallel computing operations' workload, and W_{di} is the dependent computing operations' workload (all the three workloads are quantified in cycles). Therefore, the IPC parameter can be estimated by the following formula:

$$IPC = \frac{N_i}{T \times f} = \frac{N_i}{\max\ (W_{mi}, W_{pi}) + W_{di}} \qquad (5.4)$$

From the above formula, it is obvious to see that for a task with a fixed number of workload (N_i), and executed on a processor with a fixed frequency (f), when the execution time (T) becomes longer, the IPC value for this task will consequently become lower.

Furthermore, from the analysis model, we can deduce scenarios and reasons for the longer execution time and the lower IPC value. As shown in Figure 6.2(c), when

the peripheral operations (W_{mi}) dominate the overlap part, they usually take much more time than the parallel computing operations (W_{pi}), and consequently result in a period of CPU idle time during the task execution. Therefore, in this case, it often leads to a much lower IPC value for the execution interval.

In general, peripheral operations are a combination of both memory-bound and I/O-bound operations. In our analysis model, we have assumed that peripheral components are asynchronous with the CPU and the processing speed of peripheral components is usually much slower than that of the CPU. Thus, some CPU computing operations have to be stalled while waiting for data from the peripheral operations if they are dependent on the peripheral operations.

In memory-bound operations, especially when a cache miss operation happens, the time will be prolonged as the CPU computing operations have to wait for the necessary data to be read from memory. In this case, the IPC value can become lower if very few parallel computing operations are simultaneously executing during this execution period.

I/O-bound operations may be implemented in use of two different ways: polling and interrupt. When using a polling method, the CPU is in a busy-waiting state. In such a case, the IPC value is not going to be lowered since these polling operations usually belong to the parallel computing operations and could execute concurrently with the I/O-bound operations. When using an interrupt method, the CPU is idle and waiting for an interrupt signal from the I/O controller to notify that it has finished the I/O task. In our analysis model, we have assumed that the clock is gated when the processor is idle. Thus, in this case, the IPC value is becoming lower since the CPU computing operations have to be

stalled while waiting for data from the I/O and very few parallel computing operations are executed simultaneously.

In summary, the IPC value will become much lower for an execution interval when the peripheral operations dominate the execution interval, especially for memory-bound operations when encountering cache miss operations and I/O bound operations when using interrupt style. This finding also explains the wide variations of the IPC values during application execution period.

This observation motivates us to employ the IPC parameter as a performance indicator to scale individual application execution intervals as if they could be run at their own performance-specific frequency and voltage. When an executed interval has lower performance activity level, which quantified by a lower IPC value, it is likely to have opportunities to reduce power dissipation of the microprocessor by slowing it at a lower frequency and voltage.

5.2 Methodology

In this section, we will present the detailed implementation of our method. To schedule the voltage and frequency at which a microprocessor is running, there are three tasks to be done in this method, including identification, prediction and speed-setting. We shall describe them in the following.

5.2.1 Identification

In this task, we are going to estimate the micro-architecture parameter, IPC, for the

current or some recent execution intervals. To achieve this goal, we employed a modified SimpleScalar tool for measuring the IPC value of every interval with a fixed amount of instructions. In this method, choosing an appropriate interval length for an application is important, since the following three tasks are all performed at the fixed length interval throughout application execution period. In Section 5.3.3, we shall discuss the impact of different interval lengths of an application on the final results of our method and show how to determine the appropriate interval length for an application.

To obtain the probability distribution for intervals' performance, we attempted to measure a quantifiable estimation of the IPC value for every interval during application runtime period. Firstly, we executed the benchmark application on our modified SimpleScalar at the baseline processor setting. Then, for each interval during application execution, we designed an algorithm to estimate its performance activity level in terms of the IPC value, which is illustrated in Algorithm 5.1.

1. Set an interval: a fixed amount of instructions N_i
2. For an interval, glean its execution results
   a) Consumed cycles N_c;
   b) Consumed time T;
3. For an interval, IPC = N_i / N_c;
4. For an interval, Performance = IPC

Algorithm 5.1: Interval performance estimation algorithm

As is shown in Algorithm 5.1, for each interval with a fixed amount of instructions, we first collected a trace of primitive execution results, including its consumed cycles and spent time. Then, by using Formula 5.2 presented in the

previous section, we calculated the IPC value for the executed intervals. Finally, we associated the measured IPC value with these executed intervals as their performance indicator.

5.2.2 Prediction

In the second task, the future interval's performance is predicted based on the current interval's or some recent intervals' performance in terms of their IPC values.

In general, the performance demand for the coming interval could be predicted based on the traced performance of one or some recent executed intervals. In [68], Govil et al. compared different predicting policies for DVS algorithms and found that simple algorithm rather than "smart" predicting algorithm may be most effective. Possible reason could be that there are usually more overheads, such as complex settings and complicated computations, in these smart predicting policies, thereby resulting in mistaken or ineffective predictions for future performance demand. Therefore, in this design, we employed PAST, a simple prediction algorithm proposed by Weiser et al. [67], as our heuristic policy for this task. Basically, the main idea of PAST algorithm is to look a fixed interval into the past, and assume the next interval will be like the previous one.

Following the PAST prediction policy, our method limits itself to only analyzing the performance of the immediately preceding interval to predict that for the future interval: if the IPC value for the previous interval was high, the coming interval could also have high performance demand and should be executed in a high speed; while if

the IPC value was low, the next interval could also have low performance demand and should be executed in a low speed.

5.2.3 Speed-setting

In the third step, the goal of the speed-setting task is to decide whether to scale the voltage and frequency of the processor and by how much.

As found, there is a wide range of IPC values in our traced results, which implies that the executed intervals have many different performance activity levels. However, in our design, we do not intend to continuously scale the processor's voltage and frequency over such a wide range. Furthermore, it is infeasible to use continuously variable voltage and frequency scale because the supply voltage and clock frequency transition of processors has a cost in terms of both time and energy consumption. Therefore, in this design, to achieve power reduction by the processor slowdown, we employed a simple DVS setting with two supply voltages and frequencies. Accordingly, we defined only one transition threshold, a boundary value of IPC indicator, to scale between the two voltage/frequency settings.

The decision to scale the frequency and voltage is determined by the defined IPC threshold. If the predicted IPC value for the next interval drops below the threshold, the voltage and frequency of microprocessor is scaled down; otherwise, if it is higher than the threshold, the voltage and frequency is scaled up. Thus, it is important to choose an appropriate IPC threshold value for achieving a successful design. In Section 5.3.4, we will discuss the effect of different IPC threshold values on our

experiment results, and attempted to determine the appropriate IPC threshold value for our method.

After making the decision of whether to scale the clock up (or down), the next problem is to decide how much to scale the processor voltage and frequency. In our method, we used the supply voltage/frequency setting of the Intel XScale processor since it is a typical application of actual DVS implementations and is widely used today. Thus, in this method, a two-mode setting for the processor is illustrated as below:

$$Mode = \begin{cases} Slow: & 1.1\text{v} \ \& \ 150 \text{ MHz} \\ Normal: & 1.6\text{v} \ \& \ 600 \text{ MHz} \end{cases}$$

5.3 Results

5.3.1 Evaluation metric

In experiments, we make use of three metrics to evaluate results of our method, including: energy saving, performance degradation, and energy-performance product improvement. Detailed experiment setup and benchmark information to measure the three metrics are already discussed and presented in Chapter 4.

Energy saving is used to evaluate how much energy could be reduced by using our low-power design. To estimate the energy saving result of an application, we perform it on our modified Wattch simulator at both dynamic voltage scaling and fixed voltage settings, and calculate the difference of its energy consumption.

Performance degradation is used to evaluate how much performance penalty

would be generated by using our design. To determine the performance degradation result of an application, we examine the difference of its execution time at both fixed voltage and dynamic scaled voltage.

As is discussed in Chapter 3, to achieve a successful power-performance trade-off optimization, our design is subject to a soft time constraint: $T(\delta) \leq (1+r)T$, where δ is the slowdown factor of the processor, T is the application total execution time without CPU slowdown, and r is the user acceptable performance penalty ratio.

In our experiment results, we employed the performance degradation (PD) metric to indicate the resulting performance penalty caused by our method. Thus, to meet the defined soft time constraint, our PD value must to be less than rT . Moreover, to evaluate our results, we defined the user acceptable performance penalty ratio (r) to be 10%. Therefore, for any application, when its performance degradation is more than 10% of the original execution time, it means that our method does not achieve its goal since it can not meet the time constraint.

Energy-performance product improvement is used to evaluate how much both execution latency and energy are affected by using our method. To determine the energy-performance product improvement of an application, we calculated the difference of its energy*performance results at both fixed voltage and dynamic scaled voltage. By using energy*performance improvement (EPI), it actually helps us understand the overall trade-off between the energy reduction and the induced performance slowdown from our mechanism.

5.3.2 Principal results

Figure 5.3 shows our principal results: energy saving (ES), performance degradation (PD), and energy*performance improvement (EPI) for the applications in our benchmark suite. To optimize the power-performance trade-off, we employed an IPC threshold value (1.3) in experiments and all results shown in this section are gathered by using this threshold. For the reason to select this value as our standard threshold, we will discuss and present in the next section.

To determine these results, we compared the simulation results obtained at dynamical voltage scaling with that measured at a fixed clock speed and voltage, both on the same baseline processor setting. Furthermore, we normalized results obtained at voltage scaling to those measured at fixed voltage. Therefore, in the following figure, all the bars represent results with dynamical voltage scaling, which are normalized to those results obtained at fixed voltage (the normal level setting in our case).

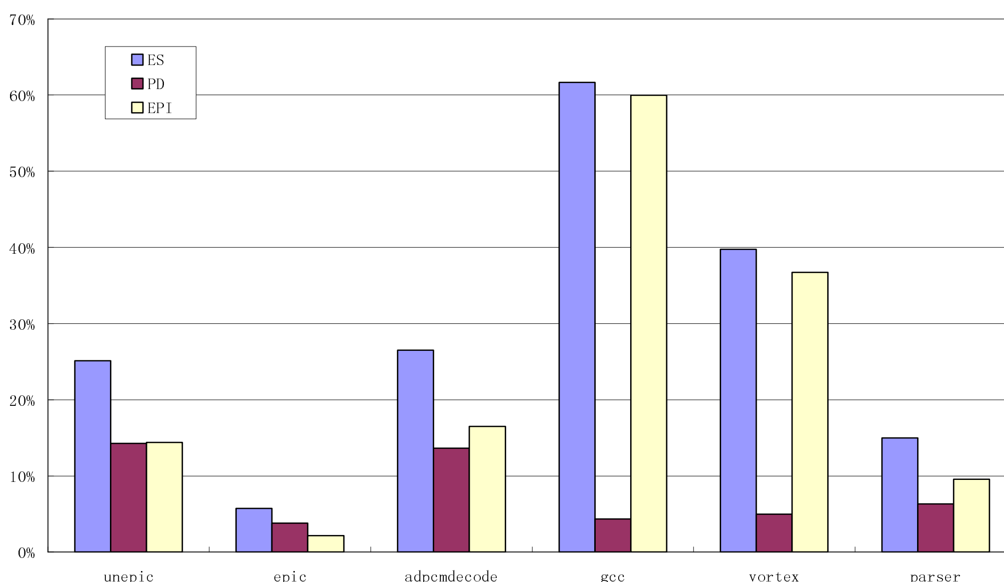


Fig. 5.3: Principal results: ES, PD, and EPI.

5.3.2.1 Energy savings

Our simulation results indicate that the potential for energy savings from our method is high. Figure 5.3 shows energy saving for all the benchmark applications. Compared with the energy consumption obtained at fixed voltage, energy savings from our mechanism vary from 5% to 60%, with an average of 29%. In contrast, "*gcc*" achieved the highest energy saving (61.7%), while "*epic*" only saved the least energy (5.8%). In general, the energy saving in our mechanism is decided by the slack time of intervals available at runtime. For the application that has less energy saving, the ratio of slack time to its whole execution time must be smaller, whereas the application that achieves higher energy saving must have a bigger ratio of slack to trade speed for energy reduction. For example, "*gcc*" could have much more slacks than "*epic*" since it saved more energy by dynamic voltage scaling reconfiguration.

5.3.2.2 Performance degradations

We do show that the overall performance of applications obtained from our mechanism does not degrade significantly compared with the baseline. As shown in Figure 5.3, almost all the performance degradations of applications are less than 10%, with an average of 8% slowdown. This finding implies that almost all applications meet their soft time constraints by using our method. Therefore, we can see that our low-power design did not adversely hurt execution latency very much for all these applications in our simulation. The reason is that we only scaled the clock speed of processor when there is slack in the interval execution, which means we attempted to

finish the task by a slow but adequate speed to reduce energy as well as maintain its performance. Thus, for these intervals having slacks, their goal performance can be achieved while running at slow clock rates.

*5.3.2.3 Energy*performance improvements*

As shown in Figure 5.3, the improvement of energy*performance (EP) varies from 2%-60% (average is 23%). Therefore, for each application, although there is undesirable but comparatively minor performance degradation, the overall improvement of energy and performance product is advantageous, with respect to the baseline results. At the same time, from Figure 5.3, we can see that the improvement of energy*performance of all applications is almost the same as their energy savings. The reason is that the energy saving of an application is much more than its performance degradation and thus it dominates the overall result.

Based on the three results presented in the above sections, we can draw a conclusion that our online identification and prediction mechanism, which dynamically scales the voltage and frequency of the processor in response to the IPC value of fixed intervals, is an effective way to reduce processor energy consumption and maintain the application performance with little extra overhead.

5.3.3 Impact of interval length

In this section, we shall discuss the impact of different length settings of an application execution interval on the final simulation results: energy saving (ES), performance degradation (PD), and energy*performance improvement (EPI). To

evaluate the impact of different interval lengths, we only changed the interval length setting in the first step described in Section 5.2.1, and kept the other settings to be the same in the following steps during simulations (the IPC threshold is set to be 1.3 for this experiment). Figure 5.3 shows the experiment results of "gcc" when using four different interval lengths of (10k, 100k, 500k, 1M).

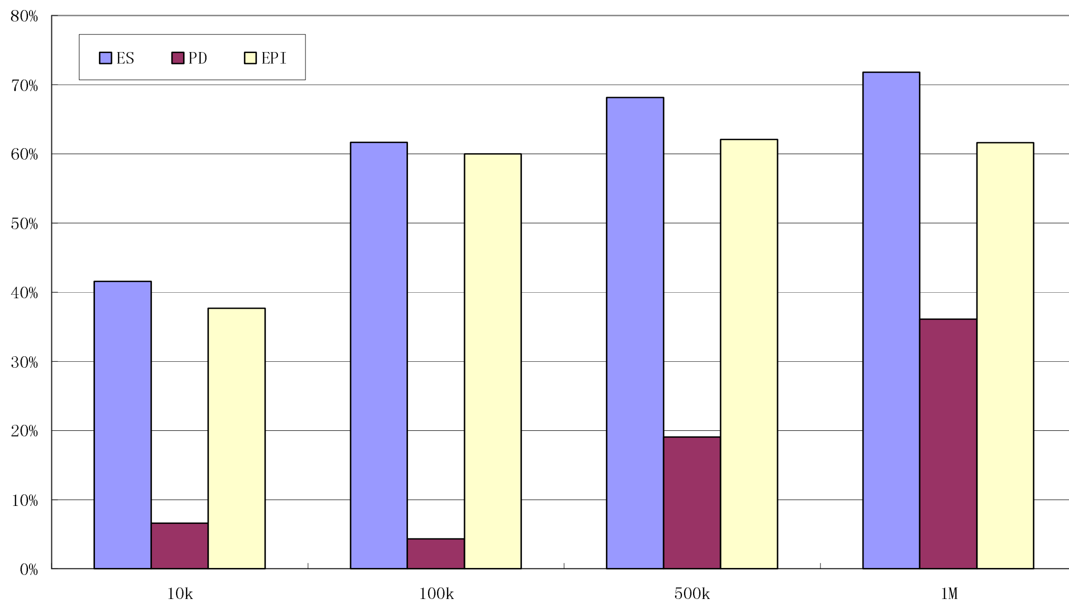


Fig. 5.4: The impact of different interval length on "gcc"

As shown in Figure 5.4, when using different interval length settings, there is actually much impact on our method and the final experiment results are quite different. Obviously, when the interval length varied from 10k to 1M, results of energy saving become higher, increasing from 40% to 70%, which implies that longer interval length setting will get more energy saving. But, as we can see from the figure, when the interval length increased from 100k to 1M, results of performance degradation in our method also increased very much, from 4% to 35%. This finding indicates that longer interval length will cause longer performance penalty although it

can achieve more energy saving. When the interval length is set to be 10k, comparing with the results obtained at the interval length of 100k, its energy saving is much less but its performance degradation is a bit more because of more runtime identification and prediction cost. Thus, its overall energy*performance improvement (37%) is much less that (60%) obtained at the 100k interval length setting, which means that shorter interval length setting will achieve less energy saving as well as less performance degradations.

Based on the experiment results, we finally decided to set the interval length of “gcc” to be 100k and our simulation results proved that this interval length setting is efficient. For different applications, they use different input files and consequently have different execution lengths. Thus, we have to follow the above experiment and analysis to determine appropriate interval lengths for them and achieve a good tradeoff between the energy savings and performance degradations.

5.3.4 Sensitivity to Slowdown Threshold

The experiment we discussed in this section shows the impact of different IPC thresholds on our final simulation results: energy saving, performance degradation and energy*performance improvement. For a clear comparison, we only presented the simulation results by using three IPC thresholds values (1.0, 1.3 and 1.6) to scale runtime intervals into distinct execution modes of microprocessors. Thus, for a given application, we can explore the impact of different switching thresholds by comparing the collected results.

5.3.4.1 Energy savings

As expected, for all applications, there is more energy savings when using a higher IPC threshold value. Figure 5.5 shows energy savings of all applications when executing with the three IPC thresholds. Obviously, for all benchmarks, their energy savings are increased when the IPC threshold value is raised. The reason for the results is evident: for a higher IPC threshold, there will be more runtime intervals to

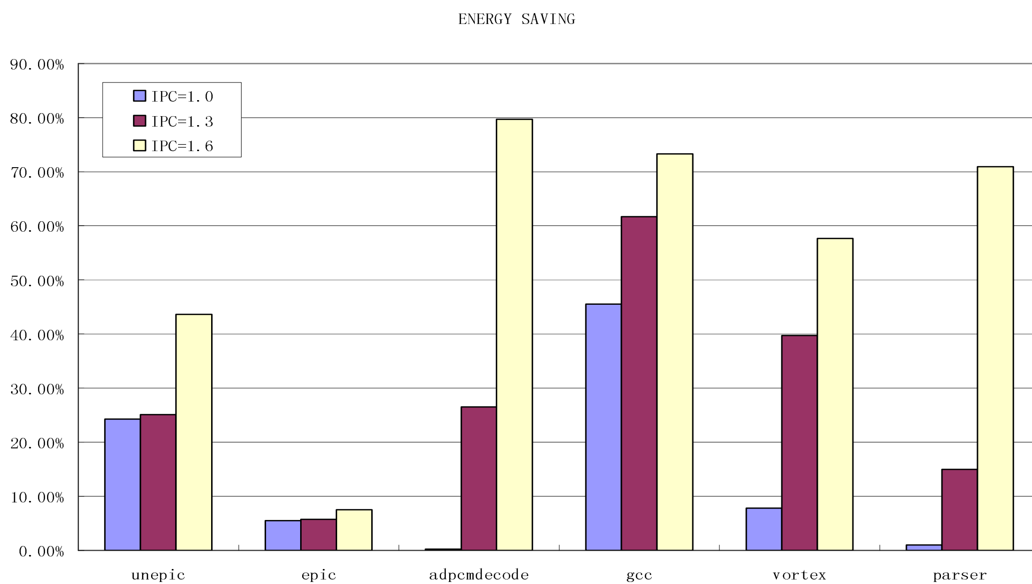


Fig. 5.5: Energy savings for different IPC thresholds.

be scaled into slow processor execution mode, which means that the CPU spends longer execution time in a low power mode; therefore, the total energy consumption will consequently be reduced. For some benchmarks, such as "*adpcmdecode*" and "*parser*", when using a higher IPC threshold, the increment of energy saving is very large, nearly double and even more.

5.3.4.2 Performance degradations

Our experiment results show that performance degradations of all applications

also increase with the raising of the IPC threshold. Furthermore, when the IPC threshold exceeds a certain value, the performance degradation is becoming very large, which means an unacceptable longtime performance penalty. As is shown in Figure 5.6, when IPC threshold is at both 1.0 and 1.3, the performance degradations of all applications are not very long, which is acceptable compared to the original execution time and can still meet the soft time constraint; while the IPC threshold is 1.6, for all

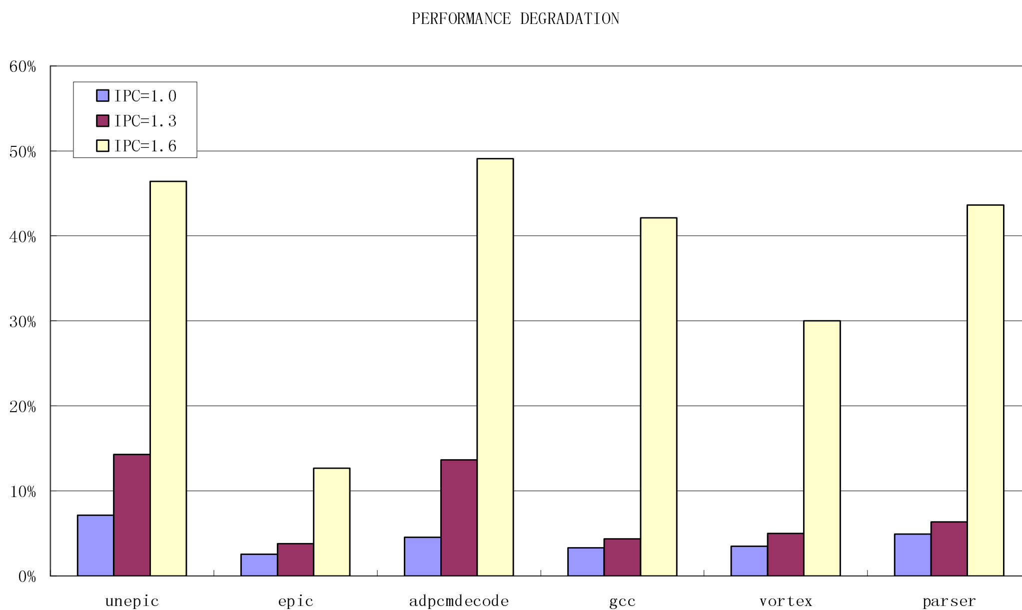


Fig. 5.6: Performance degradations for different IPC thresholds.

benchmarks, the performance degradations become very large, with an average of 37% performance degradation. For such a large performance penalty, one possible reason can be that: some intervals without slack but with an IPC value below the threshold are scaled into slow execution mode, resulting in the longer execution time and improper performance degradation.

*5.3.4.3 Energy*performance improvements*

As believed by many researchers, the improvements of energy-performance

product could well indicate the overall system energy-performance trade-off achievement. The *EPI* results shown in Figure 5.7 actually revealed the trade-off differences according to the three IPC thresholds. As shown in the figure, when the IPC threshold is low (1.0), there will be very little energy savings and also very little performance degradation, thus the overall *EPI* is consequently very small. However, for some applications, the *EPI* result is negative because the DVS overhead dominates

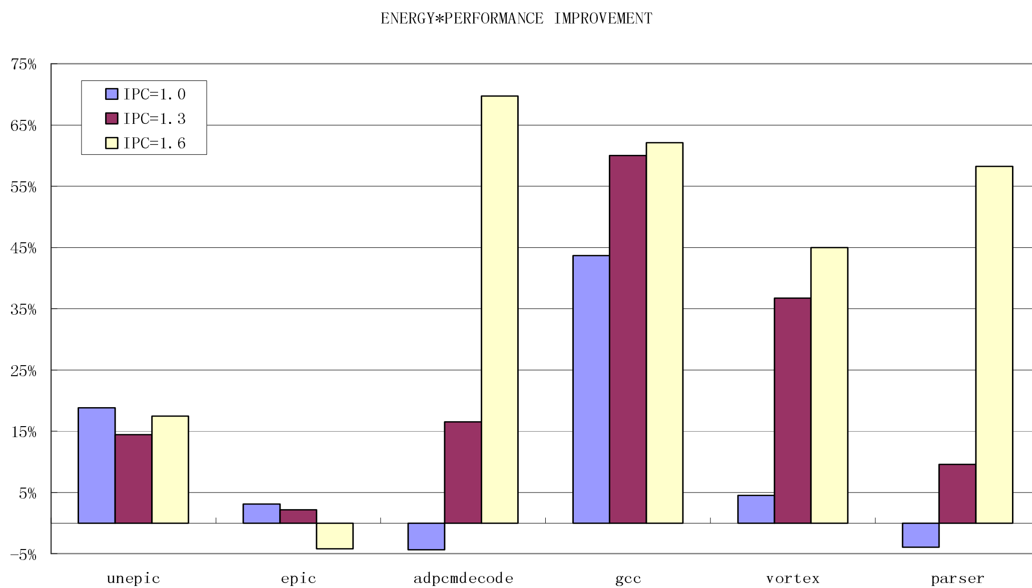


Fig. 5.7: Energy*performance improvement for different IPC threshold

the overall results. When the IPC threshold becomes appropriate (1.3), energy saving dominates the total result and the performance degradation is still very little, thus the overall *EPI* becomes good. When the IPC threshold is as high as 1.6, although there are very large energy savings, the performance degradation of applications are also very big because of the improper slowdown, thus the overall *EPI* results are unfavorable. For example, overall *EPI* of "*epic*" is negative because of its too big performance delay.

Based on the above discussions, we can see that the results of our method are sensitive to different IPC thresholds. However, to our knowledge, there is no good way to estimate the best threshold, and we can only select an experiential threshold that could achieve relatively good results in simulations. In experiments, we actually investigated a wide range of IPC values, varying from 0.5 to 2.5. Then we followed the above approach to compare the overall simulation results, and finally decided to choose the IPC threshold (1.3) as our standard experiment setting. As shown in the principal results, when the IPC threshold is 1.3, our mechanism achieved good energy savings, at the same time, with minor performance loss.

5.3.5 Overhead

In this method, the overhead could be broadly split into two types: one is the runtime cost to estimate the IPC value for the current interval and decide when to execute the DVS mode transition; the other one is the switch cost of microprocessor dynamic voltage and frequency scaling.

Unfortunately, for the first overhead related to the run-time identification and prediction, we do not yet have a good way to estimate the cost value in terms of execution time and energy consumption. However, the actual performance results measured in our experiments had already comprised the time loss caused by the dynamic judgment cost.

For the DVS transition cost, we referred to [81] and estimated one time DVS switch overhead with a transition time of 12 μs and transition energy of 1.2μJ. Therefore, for a

complete execution of applications, the overall impact of DVS cost is already considered in the total execution time and energy consumption of our simulation results.

As known, frequent or heavyweight DVS switches will have significant time and energy cost, and thus a power reduction approach is less likely to achieve a finer power-performance trade-off result. In our method, as is discussed in the previous section, the IPC threshold can affect the DVS switch times very much. Table 5.1 lists the total transition times for the three applications from MediaBench when finishing their execution by using different IPC thresholds.

	IPC=1.0	IPC=1.3	IPC=1.6
unepic	138	147	308
epic	172	184	283
adpcmdecode	2	174	523

Table 5.1: Transition times for different IPC threshold.

As shown in the Table 5.1, when the IPC threshold is raised, the transition times for all applications correspondingly increased. As is noted, when IPC threshold raised from 1.3 to 1.6, the transition times of some applications, such as "*adpcmdecode*", increased very much. Usually, much more transition times also mean much more DVS switch cost in both performance degradation and energy cost. As is discussed in the previous section, the result proves again that too high IPC threshold will not improve the overall power-performance trade-off optimization.

5.3.6 Comparison

To verify the efficiency of our IPC-driven online DVS algorithm, we compared it with the algorithm proposed by Weiser et al. [67], which is the first and most commonly used OS level algorithm to scale the frequency and voltage of the processor based on how busy the CPU was during an interval execution.

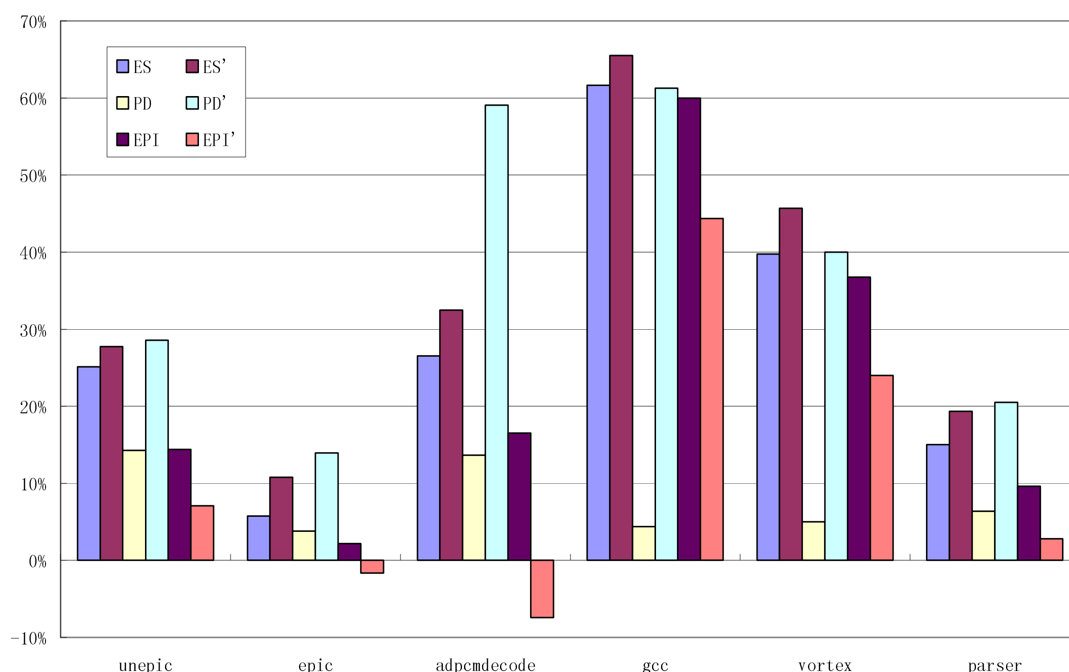


Fig. 5.8: Comparison between Our algorithm and Weiser's algorithm

Figure 5.8 shows the comparison of simulation results between the two algorithms, including energy saving (ES), performance degradation (PD) and energy*performance improvement (EPI). As shown in the figure, ES, PD and EPI present the measured results for our algorithm, while ES', PD' and EPI' present the estimated results for Weiser's algorithm. Also, all the results presented by the bars are normalized to the results obtained form application executions with fixed voltage on the same baseline setting.

As we can see from Figure 5.8, Weiser's algorithm usually saved more energy than our algorithm, but the difference between the two algorithms is not too big, only varying from 2% to 6%. However, as shown in all cases, Weiser's algorithm generated much more performance degradation than our algorithm. It can be seen that in some benchmarks such as "*gcc*" and "*adpcmdecode*", applying Weiser's algorithm resulted in very high performance degradation (more than 50%). Moreover, when using Weiser's algorithm, the performance degradations for all applications are more than 10%, which means they all could not meet the soft time constraint. As a result, for the metric of energy*performance improvement, in all cases, our algorithm achieve better results than Weiser's algorithm. This implies that: our IPC-driven online algorithm can obtain high energy savings while keeping low performance degradations to meet the soft time constraint, thereby leading to a good results for the overall energy*performance improvement.

The above findings indicate that our algorithm could achieve a considerable good result by using the micro-architecture parameter (IPC) to avoid the impact of the OS on the prediction results in former OS level DVS algorithms and obtain a reliable prediction for future performance requirements.

5.4 Summary

In this chapter, we have successfully presented a detailed interval-based identification and prediction mechanism for processor power reduction. Based on data obtained from the current interval, we calculated its performance activity level in terms of the

IPC value, and then predicted the next coming interval's performance activity to dynamically scale the voltage and frequency of the processor at appropriate level. In simulations, our approach achieved energy saving by an average of 29% with minor performance degradation, compared to a processor running at a fixed voltage and speed. Our simulation results revealed that our mechanism provides a practical and effective way to save significant amounts of energy while almost maintain the original performance.

Chapter 6

IPC-Driven Offline Power Reduction Method

In recent years, power dissipation is increasingly becoming a limiting factor for microprocessor design due to many reasons, for example, increased demand of mobile computing and high operating temperature of the microprocessor as a result of pushing its operating frequency. Therefore, various techniques are employed by modern microprocessors to reduce power dissipation, typically involving voltage reduction and activity reduction.

Voltage reduction, which usually implies a frequency reduction, could produce a significant energy saving, since energy is proportional to the square of the voltage. Typically referring to as Dynamic Voltage Scaling (DVS), this is a widely used implementation of voltage reduction in concert with clock frequency change.

Many algorithms have been proposed to dynamically set the frequency and voltage of processors to match the changing demands for processing speed and to achieve power reduction. In general, these algorithms can be divided into two categories. Some of them use the OS to monitor the system load and estimate the future processing demand at regular time interval. Control algorithms employed by the OS usually use heuristic scheduling. Examples are the interval-based algorithms of Lorch [69] and the task-based algorithms of Luo [70]. However, for some OS arbitrated DVS algorithms,

researchers found noticeable performance loss in their actual executions [71]. One possible reason is inaccurate predictions of future performance demand by the OS due to the long lapse between the low level activities at the processor and high level detection at the OS. This leads to wrong DVS decisions and the accompanying performance penalty. To overcome the problem, researchers proposed that applications themselves should provide additional information about their future performance demands, such as deadlines of real-time tasks. In these algorithms, applications must be designed with awareness of their processing demands. Then the optimal processor voltage and frequency can be selected to minimize power dissipation and still meets performance requirement of applications.

In this chapter, we introduced a micro-architecture parameter (IPC) to indicate the performance requirement for different code sections in an application to determine the value of IPC at different parts of the code. The motivation for using IPC as the performance indicator has been presented and discussed in the previous chapter. By using IPC, we implemented a traced-based code analysis and reconfiguration mechanism. The basic idea in our mechanism is to use IPC to identify opportunities in the various application code sections at which certain voltage and frequency scaling would be profitable. To do this, we first collected performance activity levels of different code sections during application training runs. We then performed an off-line analysis of the traced statistics to obtain the performance indicators in terms of the IPC value. Thirdly, we grouped these identified code sections into distinguished processor running modes with certain voltage and frequency setting. Finally,

throughout the application execution, we dynamically change processor running modes by scaling its voltage and frequency based on the profiled performance indicators. The low-power design presented in this chapter is built upon my earlier work in [103].

The rest of this chapter is organized as follows: Section 6.1 gives a brief motivation of our mechanism. Section 6.2 describes the detailed steps to implement our mechanism, which is the essential basis of this design. Section 6.3 demonstrates our simulation results, and we summarized in Section 6.4.

6.1 Methodology

The primary idea of this method is to divide the program execution code into different groups, and then each group has a uniform reaction to processor running mode adaptation. Basically, our trace-based code analysis and reconfiguration mechanism can be divided into three steps, including: phase identification, code matching, and slowdown process. We shall describe them in the following three subsections.

6.1.1 Phase Identification

In this step, our goal is to divide an application execution into individual intervals and identify some favorable intervals which have possible power reduction opportunities. To do this, we first identify performance activity levels of these intervals, and then classify them into different groups according to their performance levels, finally choose some phase groups having power reduction opportunities.

6.1.1.1 Phase Performance

In this method, a phase is defined as an execution interval with a fixed amount of instructions. In Section 6.3.3, we will give a detailed discussion about the impact of different interval length settings on our final results.

In this step, we modified the SimpleScalar tool to estimate the performance activity levels of application runtime phases. To obtain performance activity levels of each phase, there are three tasks to be done during application training runs. Firstly, we executed a benchmark application on the modified SimpleScalar tool at the baseline processor setting. For each application, we used the training data set supplied by the benchmark suits as its training input file. Then, for every phase interval throughout application execution, we collected a trace of primitive statistics, including its consumed cycles and spent time. Lastly, we employed Formula 5.2 presented in Chapter 5 to calculate the IPC value of every phase. Thus, for each phase interval throughout application execution, we estimated its performance activity level in terms of an IPC value.

The final output of our on-line trace is a performance activity level vector (PALV) for all phase intervals of an application, which also could be used as referenced performance requirements.

6.1.1.2 Phase Groups

As found, there is a wide range of IPC values in PALV, which means that application runtime phases have very many different performance activities. Since our mechanism is

not going to continuously scale voltage and frequency of processors over such a wide range, it is necessary to divide these phases into several distinct performance-specified groups according to their IPC values.

To identify phase intervals at which a CPU slowdown is feasible to achieve power reduction, we attempted to divide these phase intervals into two groups: a slow group and a normal group. Each of them represents a phase group with approximate IPC values in a specific range. Based on the knowledge obtained from our experiments, we selected the IPC value (1.3) as a threshold for classifying the two groups. We will present the reason for choosing 1.3 as the IPC threshold in Section 6.3.4. Then, by using the IPC threshold, we defined the two phase groups, which are illustrated in the following:

$$Group = \begin{cases} Slow \rightarrow IPC < 1.3 \\ Normal \rightarrow IPC \geq 1.3 \end{cases}$$

Following the above phase group definition, we clustered these phase intervals, which have similar performance activity levels, into different phase groups. Eventually, based on these classified groups with distinctive IPC values, we could identify these favorable phases, which might have power reduction opportunities.

6.1.2 Code Matching

To activate voltage and frequency scaling, we must identify the advisable code sections which could divide the program execution into different running modes. Thus, step two is to estimate such code sections in an application binary file through matching them with runtime track record of these phases identified in the first step. In

the following, we shall first show how to get the runtime track record of phases, and then present the detailed implementation of code section identification.

6.1.2.1 Runtime Track Record of Phases

In [89], Sherwood et al. have presented an efficient scheme for detecting application runtime execution behaviors. In this task, we modified their SimPoint [90] scheme and make use of their defined Basic Block Vectors (BBV) to detect runtime contents of phase intervals.

Basically, BBV represents the proportion of basic block executions through a given phase interval. To keep track of run-time information within every phase, we assign a static numeric identity (ID) to every basic block in the application execution code, starting from 1. Thus, in an interval with a fixed amount of instructions, we used BBV to record the number of times each basic block executed during the sampling period. Simultaneously, for the same interval, we employed a basic block instruction vector (BBIV) to trace detailed contents of each basic block, such as opcodes, instructions and their addresses.

6.1.2.2 Code Section Identification

Based on our collected runtime basic block clusters of phases, we attempted to match them with the binary code of an application to identify these favorable code sections. The detailed method to locate and mapping runtime execution contents of phases and static code sections could be divided into three steps, which are described as below:

- ✓ **BB ratio**: As found, although there might be hundreds of basic blocks within one phase interval, only very a few basic blocks are executed frequently and takes most of execution time. Therefore, we make use of a ratio to quantify the importance of these basic blocks, which is calculated as below:

$$Ratio = \frac{N_{bb} \times T_{bb}}{N_{phase}} = \frac{N_{bb} \times T_{bb}}{\sum_{BBID} N_{bb} \times T_{bb}} \quad (6.1)$$

  Where N_{bb} is the amount of instructions in a basic block, T_{bb} is its occurrence frequency in a phase, and N_{phase} is the total amount of instructions in a phase.

- ✓ **ID clusters**: During the application execution, there is an assigned ID for each basic block, and some continuous or nearby basic blocks usually execute together, thereby resulting in some small ID clusters. Thus, based on identified important basic blocks, we can obtain some potential basic block clusters for each phase interval.

- ✓ **Section mapping**: After identifying these potential basic block clusters by their ID, the next step is to match them with static code sections in an application binary file by using the traced opcodes and addresses of basic block clusters.

Following the above three steps, we finally identified these favorable code sections in an application binary file, which could divide the whole program execution into different processor runtime modes and contribute to power reduction. Most often, one of such identified code sections could also be defined as a small function unit.

6.1.3 Slowdown Process

After successfully identifying these code sections that have low performance requirements and could make possible contributions to runtime power reduction, Step three attempts to dynamically scale the voltage and frequency of the processor to reduce power dissipation. For this aim, there are two tasks to be finished, which will be described in the following.

Firstly, we should estimate how much to scale the processor voltage and frequency during execution period of these code sections with low performance requirements. Following the typical settings of both the Transmeta Crusoe and the Intel XScale processor, which are widely used implementations of DVS, we defined a two-mode setting with different clock frequencies and supply voltages, which is shown as below.

$$Mode = \begin{cases} Slow : 1.1\text{v} \ \& \ 150 \ \text{MHz} & \text{Slow Group} \\ Normal : 1.6\text{v} \ \& \ 600 \ \text{MHz} & \text{Normal Group} \end{cases}$$

Then, the next step is to profile indicators to direct microprocessor voltage and frequency scaling. Actually, one assumption for our method is that the application is not allowed to be modified, including both its source code and binary code. As is mentioned in the previous section, we have already obtained the detailed contents of identified code sections, including their opcodes and addresses. Thus, for one code section, we used the opcodes and addresses of its beginning and ending points as the performance indicators, and saved them into some registers of the processor simulator to direct the voltage and frequency scaling. We developed a mode selection and

profile algorithm for these identified code sections, which is shown in Algorithms 6.1.

As shown in Algorithms 6.1, Line 2 through 9 show the main loop of the algorithm applied for each code section to select its mapping mode. The main loop includes two subroutines to differentiate code sections and their mapping modes.

```
1:   Mode-Select (Code_Section(C1,...,Ci), Mode(M1, M2)){
2:       for each (C1,...,Ci) in Code_Sections {
3:           for Modes M1, M2 {
4:               if Ci only belongs to one Mode
5:                   OneToOne(Ci);
6:               else Ci belongs to two Modes
7:                   OneToMany(Ci);
8:           }
9:       }
10:      OneToOne( Cj ){
11:          if ( IPC(Cj) < 1.3 )
12:              save Indicator Ij for Cj;
13:      }
14:      OneToMany( Cj ){
15:          if ( IPC(Cj) < 1.3 && IPC(Cj) >=1.3 )
16:              set IPC(Cj) >= 1.3;
17:      }
18: }
```

Algorithms 6.1: Pseudocode for the mode selection and profile algorithm

From line 10 to 13, the *OneToOne* function deals with these code sections keeping the same performance activity level throughout the application execution, which means that a code section only belongs to one execution mode. Another function *OneToMany*, between line 14 and 17, will treat with these code sections, which have many performance activity levels throughout the application execution, to find out their mapping modes. This case is possible since the same code section might be in different phases which have different performance activity levels. For this case, we

select a group with the highest performance activity level for the code section.

6.2 Results

6.2.1 Evaluation metric

In this chapter, we still made use of the three metrics (energy saving, performance degradation, and energy*performance improvement) to evaluate the results of our method obtained in experiments. Detailed experiment setup and benchmark information to measure the three metrics are already discussed and presented in Chapter 4. Energy saving is used to evaluate how much energy could be reduced by using our low-power design. Performance degradation is used to evaluate how much performance penalty would be generated by using our design. Energy*performance improvement is used to evaluate how much both execution latency and energy are affected by using our method.

Furthermore, to achieve a successful low-power design, the method proposed in this chapter is still subject to the soft time constraint: $T(\delta) \leq (1+r)T$, where δ is the slowdown factor of the processor, T is the application total execution time without CPU slowdown, and r is the user acceptable performance penalty ratio. In this section, we still defined the user acceptable performance penalty ratio (r) to be 10% to evaluate our results, thus the soft time constraint is 1.1 times of the original execution time. Therefore, when using our design, if the performance degradation of an application is more than $0.1T$, it implies that our method does not achieve its goal for this application since it can not meet the soft time constraint.

6.2.2 Principal results

In this chapter, for achieving a good power-performance trade-off optimization, we set the IPC threshold value to be 1.3 in experiments, and all results shown in this section are gathered under this threshold. In Section 6.3.4, we will evaluate our results by using different IPC threshold values and present the reason for choosing 1.3 as the IPC threshold in our method.

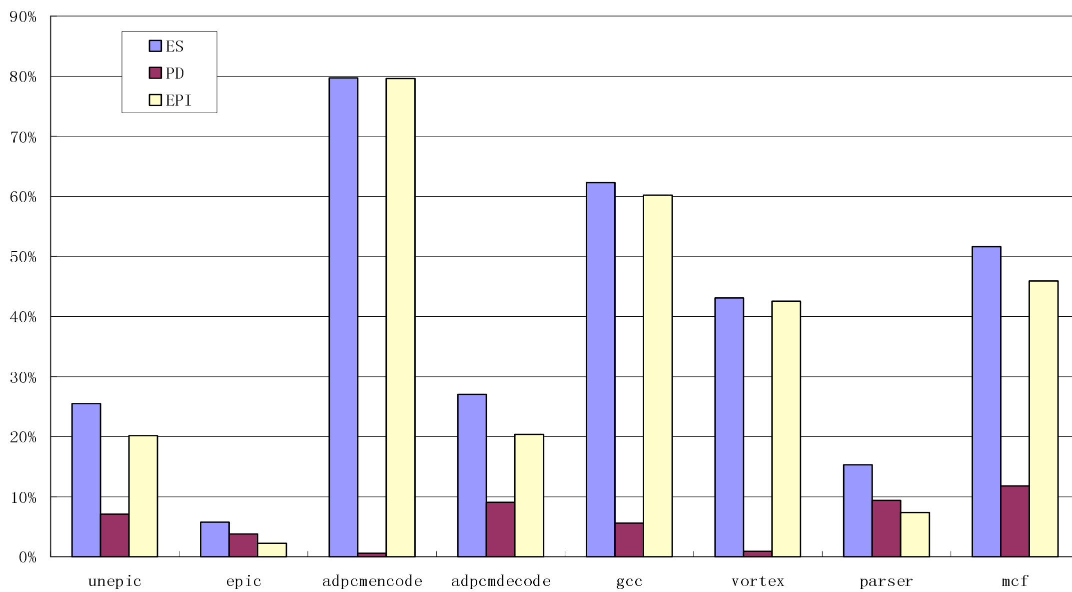


Fig. 6.1: Principal results: ES, PD, and EPI.

Figure 6.1 shows our principal results, including energy saving (ES), performance degradation (PD), and energy*performance improvement (EPI) for our chosen benchmark applications. To validate our method, we compared the experiment results obtained with dynamical voltage scaling with that measured at a fixed clock speed and voltage, both on the same baseline processor. Furthermore, we normalized all simulation results with voltage scaling to those with fixed voltage. Thus, in following figures, all the bars represent results with dynamical voltage scaling, which are

normalized to those results obtained with fixed voltage (the normal level setting in our case).

6.2.2.1 Energy Saving

In experiments, we obtained the energy saving by first running each application under our modified Wattch simulator at both dynamic voltage scaling and fixed voltage and then calculating the difference between the two results. Our simulation results indicate that the potential for energy savings of our code analysis and reconfiguration mechanism is very high.

Figure 6.1 shows energy saving for all the benchmark applications. Compared with energy consumption of the baseline processor, energy savings of applications by using our mechanism vary from 5% to 80%, with an average of 40%. By contrast, "*adpcm encode*" achieved the highest energy saving by 79.7%, while "*epic*" only saved the least energy by 5.8%. Generally, the energy saving in our code analysis and reconfiguration mechanism is dominated by slack, i.e. the CPU idle time, of applications available at the runtime. For the benchmark that has less energy saving, the ratio of slack time to its whole execution time must be less, whereas the application that achieves higher energy saving must have a bigger ratio of slacks to trade speed for energy reduction. For example, "*adpcmencode*" must have much more slacks than "*adpcmdecode*" since it saved more energy by dynamically scaling voltage of the processor.

6.2.2.2 Performance Degradation

To determine the runtime performance penalty of an application, we compared the total execution time that a benchmark spent in both fixed voltage and dynamical scaled voltage, and then made use of the difference between the two cases as the performance degradation.

We do show that the overall performance of applications based on our mechanism does not degrade significantly compared with that at the fixed voltage. As shown in Figure 6.1, almost all the performance degradations of applications are less than 10%, with an average of 6% slowdown. It implies that all the applications meet their soft time constraints (1.1*T*) when using our method. Therefore, we can draw a conclusion that our low power design did not adversely hurt the execution latency very much for all these applications in our simulation. The reason is that we only scale the clock speed of processor when there is slack during the application execution, which means we attempted to finish the task by a slow but adequate speed to reduce power dissipation as well as maintain its performance. Therefore, for these benchmarks having enough slacks, their original performance would not be degraded while running at slow clock rates.

*6.2.2.3 Energy*Performance Improvement.*

We make use of the metric energy*performance improvement (EPI) to show how both execution latency and energy consumption of an application are affected by dynamic voltage scaling. By using *EPI,* Figure 6.1 helps to understand the trade-off between

the energy reduction and the induced slowdown by using our code analysis and reconfiguration mechanism.

As shown in Figure 6.1, the improvements of energy*performance varied from 2%–80% (average is 35%). Therefore, for each application, although there is undesired but comparatively minor performance degradation, the overall improvement of energy and performance product is advantageous, with respect to the baseline configuration. Also, from Figure 6.1, we can see that the improvement of energy*performance of all applications is almost the same to their energy savings. The reason is straightforward since the energy savings of applications are all much more than their performance degradation.

From the above discussion about the three metrics, we can draw a conclusion that our code analysis and reconfiguration mechanism with dynamical voltage and frequency scale in response to IPC variations during application execution period is an effective way to reduce processor energy consumption and maintain the application performance.

6.2.3 Impact of Phase Interval Length

In this section, we shall study the impact of different length settings of phase intervals on our final simulation results: energy saving (ES), performance degradation (PD), and energy*performance improvement (EPI). In experiments, to evaluate the impact of different interval length settings, we only changed the interval length in the first step of our method described in Section 6.1.1, and kept the other settings to be the same (the IPC threshold is set to be 1.3 for this experiment). Figure 6.2 shows the

experiment results of "gcc" when using four different interval length settings of (10k, 100k, 500k, 1M).

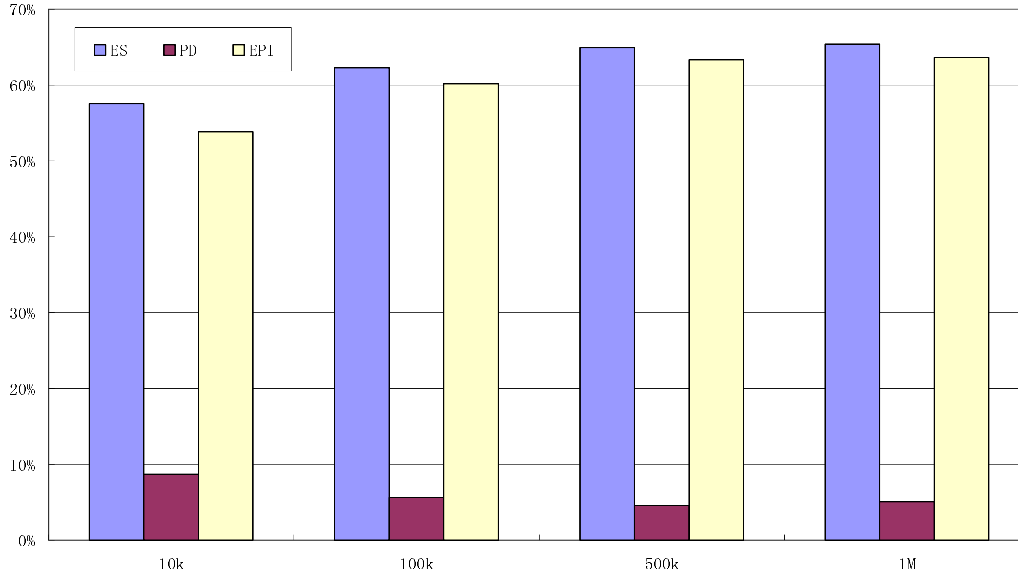


Fig. 6.2: The impact of different interval length on "gcc"

As shown in Figure 6.2, when using different interval lengths in our method, the experiment results are not affected very much. For energy saving, the four results are all around 60% for the four interval length settings. For performance degradation, there are also not very big differences in the four cases when using different interval lengths, only varying from 4% to 9%. As a result, when using four different interval length settings, there are not very much changes in the energy*performance improvement results for the four cases. This observation is reasonable: in this code analysis and reconfiguration method, we only directed the DVS scheduling during execution periods of some favorable code sections, which are identified in the application binary file through code-match process with these phase intervals. When using different interval length settings in the method, we found that these identified code sections are almost the same, and thus there is little impact on the final

experiment results. This finding also proves that only 10% program codes consumes more than 90% execution times (known as the 90/10 law).

Although different interval lengths will not affect our final results very much, there are still some problems needed to be considered. If using a too small interval length, for a big input file of an application, there will be a huge amount of phase intervals to be handled in the following processes in this method, thereby resulting in a quite long process time. Therefore, for different applications, according to their input files and their complete execution length, we have to choose appropriate interval length settings for them to achieve a trade-off between the process time and the amount of phase intervals.

6.2.4 Sensitivity to Slowdown Threshold

The experiment we discussed in this section is going to show the impact of different values of the IPC threshold on our final simulation results: energy saving, performance degradation and energy*performance improvement. For comparison, we present simulation results of three IPC threshold values (1.0, 1.3 and 1.6) used in our experiment to group code sections into distinct microprocessor execution modes. Thus, for a given application, we can explore the impact of different switching thresholds by comparing the experiment results.

6.2.4.1 Energy Savings

As expected, for all applications, there is more energy savings when using a higher IPC threshold value. Figure 6.3 shows energy savings of all benchmark applications

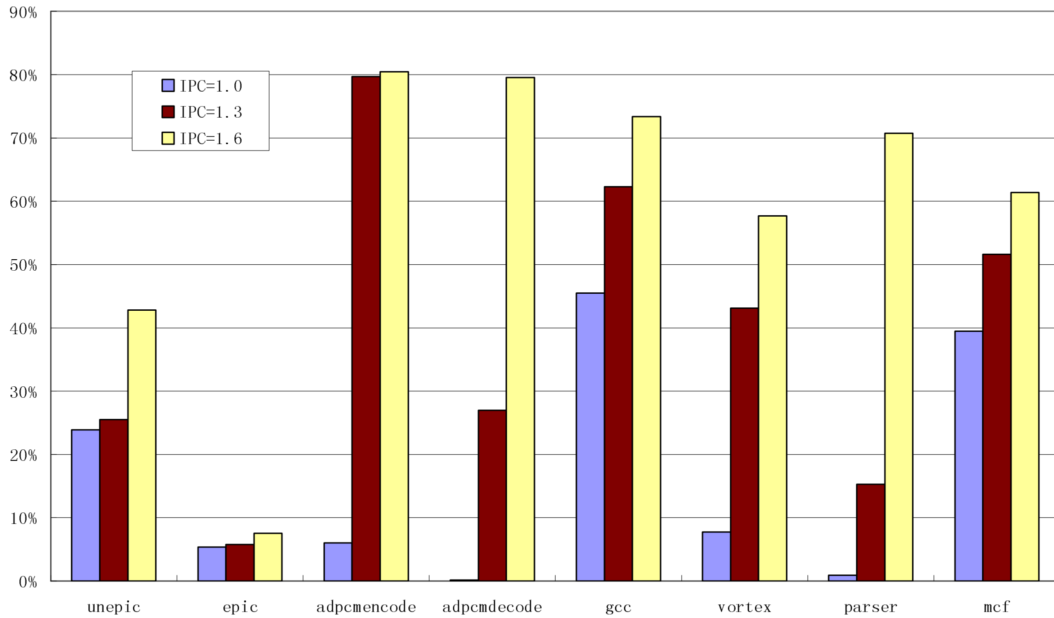


Fig. 6.3: Energy saving results

when executing by using the three IPC thresholds. Obviously, for all benchmarks, their energy savings are increased when the IPC threshold value is raised. The reason is evident: for higher IPC threshold, there will be more code sections to be grouped into slow processor execution mode, which means that the CPU executes longer time in a low power mode; therefore, the total energy consumption will consequently be reduced. For some benchmarks, such as "*adpcmdecode*" and "*parser*", when shifting to a higher IPC threshold, the increment of energy saving becomes to be very large, nearly to be double.

6.2.4.2 Performance Degradations

Our experiment results show that performance degradations of all applications also increase with the increase of the IPC threshold. Furthermore, when the IPC threshold exceeds a certain value, the performance degradation will become very large, which means an unacceptable long time postpone for the application execution.

As shown in Figure 6.4, when IPC threshold is at 1.0 and 1.3, the performance

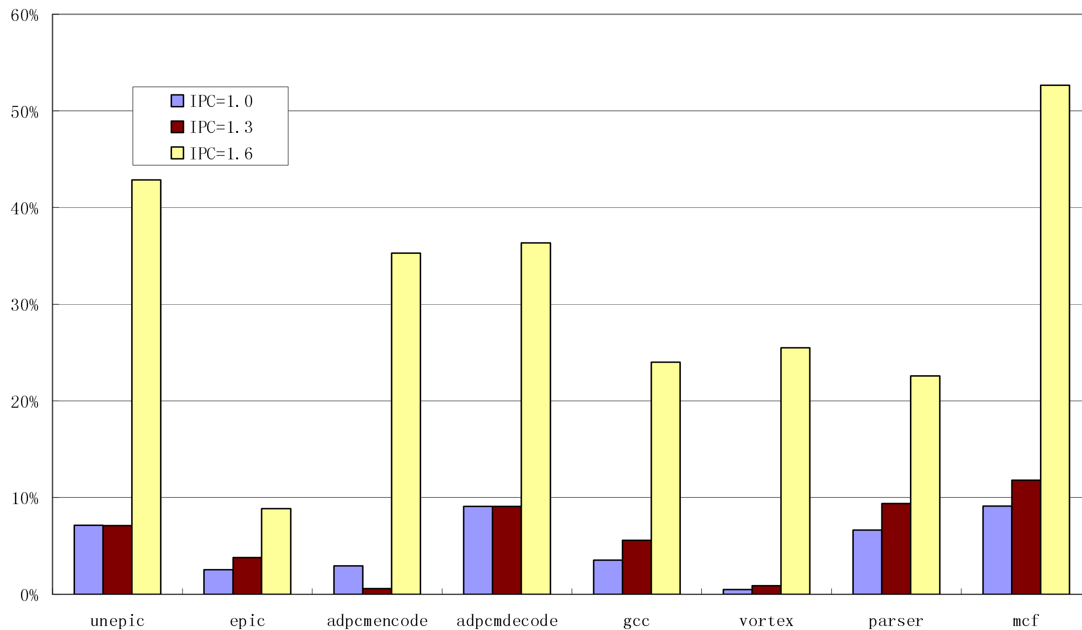


Fig. 6.4: Performance degradation results

degradations of all applications are not very big, which is acceptable compared to the original execution time and can meet the soft time constraint; while the IPC threshold is at 1.6, for all benchmarks, the performance degradation becomes very large, with an average of 30% execution degradation, thus all applications can not meet the soft time constraint. For such a large degradation, possible reasons may be: some code sections without slack but with an IPC indicator value below the threshold are grouped into slow execution mode and result in the longer execution time and improper performance degradation.

*6.2.4.3 Energy*Performance Improvements*

As believed by many researchers, the improvements of energy-performance product could well indicate the overall system energy-performance trade-off. As

shown in Figure 6.5, the *EPI* results actually revealed the trade-off differences when using the three IPC thresholds. As we can see from the figure, when the IPC threshold is low at 1.0, there will be very little energy savings and also very little performance

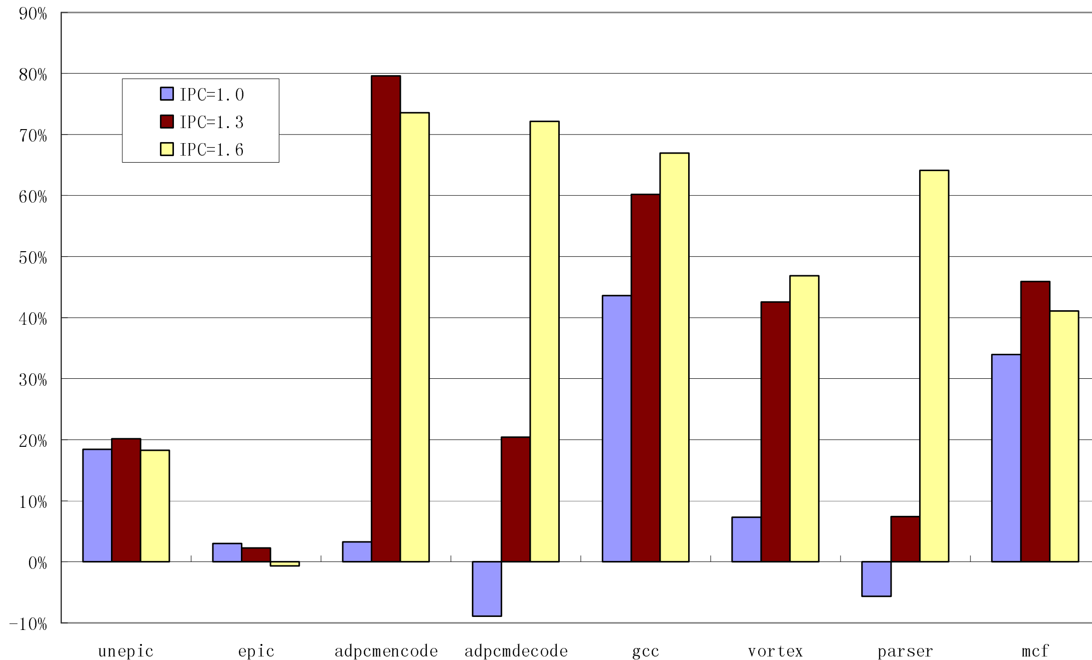


Fig. 6.5: Energy*performance improvement results

degradation, thus the overall *EPI* results for all applications are small. As found, for some applications, the *EPI* result is negative because of the DVS overhead; when the IPC threshold becomes higher at 1.3, energy saving dominates the overall result and the performance degradation is still very small, thus the overall *EPI* results of all applications become good; when the IPC threshold is high at 1.6, although there are very large energy savings, the performance degradation of some applications are also very large because of the improper slowdown, thus the overall *EPI* results are unfavorable.

Therefore, based on the above discussions, we can draw a conclusion that the

results of our method are sensitive to different IPC thresholds. However, as is discussed in the previous, there is no good way to estimate the best threshold, so in this method, we still have to select an experiential threshold that could achieve good results. In experiments, we investigated a wide range of IPC values, varying from 0.5 to 2.5, and then we followed the above approach to compare the overall simulation results, Thus, based on our experiment results for a wide range of IPC values, we also chose the IPC threshold (1.3) as our standard experiment setting for this method. As shown in the principal results, when the IPC threshold is 1.3, our mechanism achieved good energy savings, at the same time, with minor performance loss.

6.2.5 Overhead

In our code analysis and reconfiguration method, the overhead could be broadly divided into two types: one is the runtime judgment cost to monitor the profiled indicator and decide when to execute the DVS mode transition; the other one is the switch cost of microprocessor dynamical voltage and frequency scaling.

Unfortunately, the first type of cost is related to the run-time DVS decision, and we do not yet have a good estimate for the time delay and the power dissipation. However, a partial runtime judgment cost, i.e. the execution time loss, has already been comprised into the performance degradation results in our simulations.

For the DVS transition cost, we referred to [81] and estimated one time DVS switch overhead with a transition time of 12 μs and transition energy of 1.2μJ. Therefore, when an application finished its execution, the overall impact of DVS cost is included in the

total execution time and energy consumption of our simulation results.

As well known, frequent DVS switches will have a significant cost in terms of both execution time and energy consumption, and thus the power-aware approach is less likely to achieve a finer power-performance trade-off result. As is discussed in the previous section, we found that our experiments results will affected when using different IPC threshold values. Thus, in this following, we are going to investigate how much the DVS switch times will be affected by the IPC threshold.

Figure 6.6 shows the total transition times for the four applications selected from MediaBench when completing their execution under different IPC thresholds.

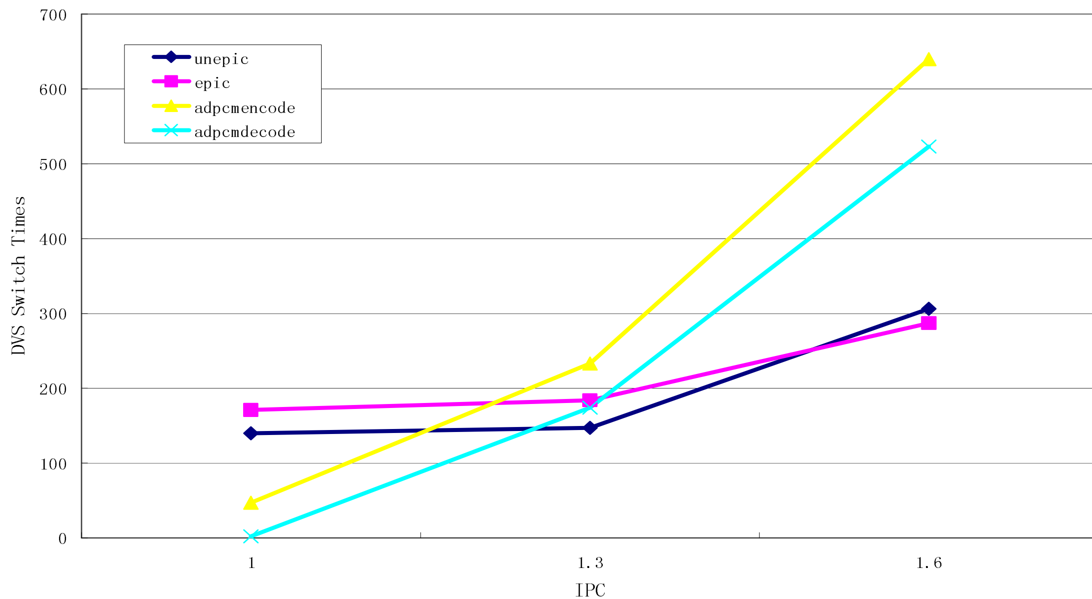


Fig. 6.6: Transition times for different IPC threshold.

As shown in the Figure 6.6, when the IPC threshold increases, the transition times for all applications correspondingly increased. In particular, when IPC threshold is raised from 1.3 to 1.6, the transition times of some applications, such as

"*adpcdecode*", increased very much. Obviously, much more transition times also mean much more DVS switch cost, both in performance degradation and energy cost. Thus, we can conclude that the DVS switch times are also highly affected by the IPC threshold. The observation again proves that too high IPC threshold will not improve the overall power-performance trade-off optimization.

6.2.6 Comparison with the IPC-driven online Method

In this section, we compared our IPC-driven offline DVS method with the IPC-driven online DVS method, which is proposed in the previous chapter. Although the two designs both use the same micro-architecture parameter (IPC) to address power reduction opportunities in applications, they employed two different mechanisms to identify appropriate chances for DVS scheduling. The offline method uses a code analysis and reconfiguration mechanism based on application training runs; On the other hand, the online method uses an interval-based identification and prediction mechanism during application execution period.

Figure 6.7 shows the comparison of simulation results between the two methods, including energy saving (ES), performance degradation (PD) and energy*performance improvement (EPI). As shown in the above figure, ES, PD and EPI present the measured results for the online method, while ES', PD' and EDI' present the estimated results for the offline method. Also, all the results presented by the bars are normalized to the results obtained form application executions with fixed voltage on the baseline processor.

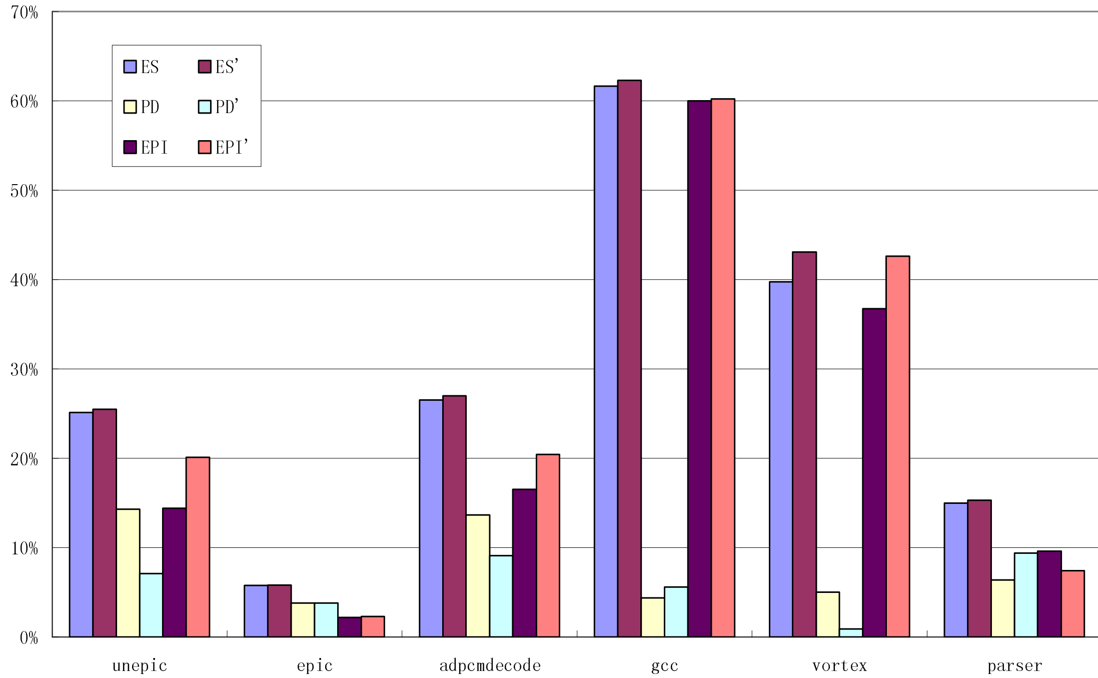


Fig. 6.7: Comparison between the online and offline algorithms

As we can see from Figure 6.7, the offline method usually saved more energy than the online method, but the difference of energy saving between the two methods is not very big, only varying from 1% to 4%. Furthermore, as shown in Figure 6.7, for most cases, the performance degradation from the offline method is smaller than that of the online method, and the difference of PD is also not big, varying from 1% to 7%. As a result, for energy*performance improvement, the results obtained from the offline method are better that that from the online method. The reason could be that: The offline code analysis and reconfiguration method could identify more power reduction opportunities than the online prediction method, and the decision made by the offline method might be more reliable than that by the online method, therefore, the overall energy-performance product results of the offline method are better than that of the online method.

The above findings indicate that both the online and offline methods proposed by us could achieved good results and it is an effective way to employ micro-architecture parameter (IPC) to identify power reduction opportunities in applications for a successful low-power design .

6.2.7 Comparison with Other Offline Methods

In this section, we compared our IPC-driven offline DVS method with an offline DVS method proposed in [74]. In their design, C. H. Hsu et al. identified memory-bound regions at source code level and made use of compiler to insert their defined instructions to direct DVS. Therefore, we compared offline DVS algorithms implemented at different implementation level.

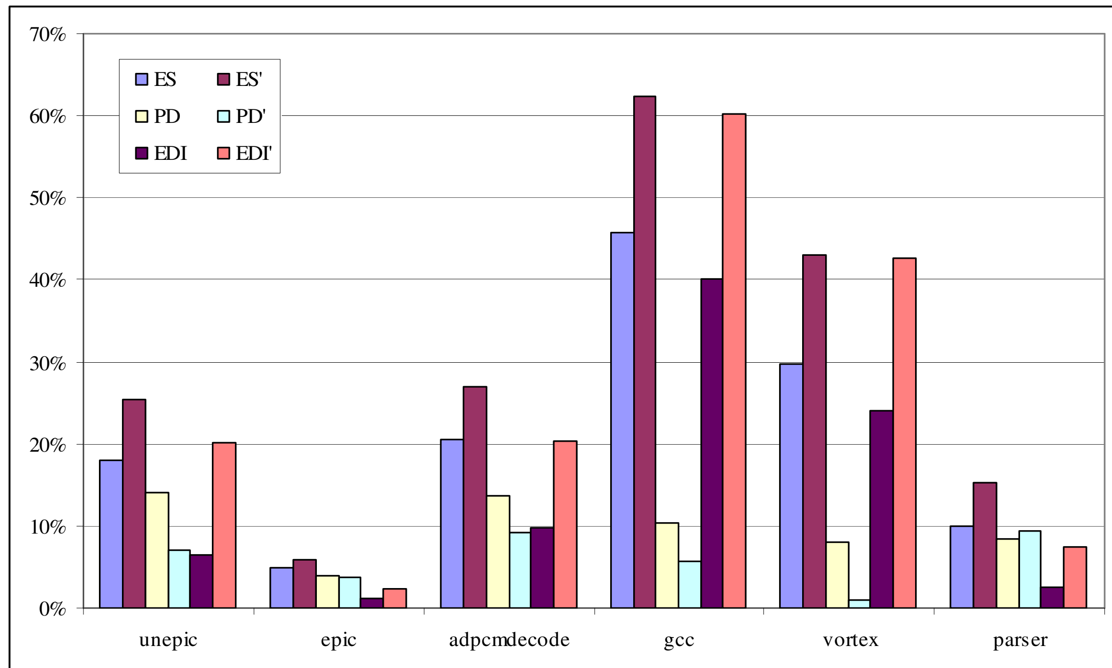


Fig. 6.8: Comparison between our and Hsu's offline DVS algorithm

Figure 6.8 shows simulation results of the two offline DVS algorithms, including ES, PD and EPI. In the figure, ES, PD and EPI present the measured

results for Hsu's offline DVS algorithm, while ES', PD' and EDI' present the estimated results for our offline algorithm.

As we can see from the above figure, our IPC-driven method achieved better results than Hsu's source code level algorithm in all of ES, PD and EPI. As shown in all cases, Hsu's algorithm usually achieved less energy savings than our algorithm. The reason might be that our micro-architecture level mechanism could find out not only memory-bound but also I/O-bound code sections, thereby identifying more opportunities to save energy. Furthermore, their algorithm most often generated more performance degradation for all benchmark applications. Possible reason could be that Hsu's identified coarser grain size of code sections at source code level. Therefore, it is straightforward that our IPC-driven offline algorithm obtained a better energy*performance improvement.

The above findings imply that: our IPC-driven offline algorithm can achieve considerable higher energy savings with lower performance degradations by identifying more energy saving opportunities and finer grain DVS scheduling at the micro-architecture level.

6.3 Summary

In this chapter, at the micro-architecture level, we have successfully implemented a code analysis and reconfiguration mechanism for processor power reduction. Based on execution results obtained during application training runs, we identified code sections with slack in terms of their IPC value, and then profile applications to dynamically

scale the voltage and frequency of processor at appropriate points during later execution. In simulations, our approach achieves significant energy savings by an average of 40% with minor performance degradation (6%), compared to a processor running at a fixed voltage and speed. Our simulation results showed that our code analysis and reconfiguration mechanism provides a practical and effective way to reduce energy consumption in many applications while nearly maintain the original performance.

Chapter 7

Methods to Identify Related Micro-architecture Parameters

In the previous two chapters, we have proposed two effective low-power designs by using the micro-architecture parameter (IPC) to identify power reduction opportunities during application execution period to direct DVS scheduling and eventually achieve a profitable power-performance trade-off.

In this chapter, we shall present two methods to identify two related micro-architecture parameters. In Section 7.1, we shall describe a method to identify application runtime power behaviors during its execution period. In Section 7.2, we shall present a method to identify data dependence length (DDL) through application runtime instructions. The two micro-architecture parameters, application runtime power behaviors and DDL characteristics, could be helpful to evaluate our proposed micro-architecture level low-power designs. For example, application runtime power behaviors could reveal runtime power dissipation characteristics for different code sections in an application, and it can be used to verify these power reduction opportunities identified in our methods. In Section 7.3, we will summarize this chapter.

7.1 Application Power Behavior Identification Method

7.1.1 Introduction

In recent years, power dissipation issues in modern processors have led to significant research efforts in power optimization technologies, such as power-aware and temperature-aware low power designs. Usually, these power optimization designs focus on not only reducing the overall energy consumptions, but also minimizing the runtime power dissipations. Therefore, runtime power dissipation behavior is becoming one of the important metrics to evaluate such power optimization designs.

Recently, researchers have proposed some methods to identify runtime power dissipation information of a computer system and then characterize its power behaviors. In 2003, Isci et al. [91] presented a practical measurement approach to monitor runtime power dissipation of a computer system. They also proposed an efficient technique in a further study [92], which could offer lower average errors for classifying system power behaviors. In [93], to identify runtime power behaviors, C. Hu et al. used the SimPoint tool to find representative program execution slices, and validated the feasibility of using them to estimate system power dissipation information. Later, in [94], they suggested to characterize system runtime power behaviors instead of the detailed power dissipation information, and proposed an effective method to characterize system runtime power behaviors.

In general, all the fore-mentioned methods measured the runtime information of a computer system, and then calculated its power dissipation to characterize the runtime

power behaviors. Therefore, the power behavior characteristics identified by them usually is for a whole computer system but not for a single application.

In this section, we will present an efficient method to identify fine-grained runtime power behaviors for a single application by using a phase classification technique. To identify runtime power behaviors of an application, first of all, we detect its representative execution phases, which is a small set of runtime instruction intervals that can be used for analysis instead of the complete executed program. Then, we perform the application on a power/performance simulator to estimate its runtime power dissipation information of these representative phases. Finally, we characterize the whole program power behaviors based on runtime power dissipation information of the representative phase. Our simulation results demonstrate that our method is efficient and speedy to identify application power behaviors and the results estimated by these selected phases is very close to that measured from the complete application execution.

7.1.2 Methodology

In this method, our primary idea is to capture the detailed power behaviors for an application by using the phase classification technique. There are three steps to implement our method. Firstly, we employ the SimpleScalar tool to perform an application and gather its runtime information. Then, for phase identification, we make use of the SimPoint tool to identify representative execution phases for an application. Finally, we estimate both the detailed runtime power behaviors and the total energy consumption for these identified phases, which are used to represent the

power behavior characteristic for the whole application. We shall describe them in the following sections.

7.1.2.1 Phase identification

This step is going to identify these representative phases, which are a small set of execution slices but could represent the power behavior characteristics of the whole program.

As found during execution period, most applications show that many runtime phases have similar behaviors in some metrics, such as instructions-per-cycle (IPC), cache miss rate and branch mis-prediction rate. Some previous works have investigated various issues to identify program runtime phase behaviors [95, 96, 97, 98]. As they suggested, instead of working over the complete program execution, only a small set of representative phase intervals is measured and analyzed to characterize the whole program behavior, which could achieve significant savings in both experiment time and storage space.. This observation motivates us to make use of the phase classification technique to speedily identify application power behaviors.

In this step, we make use of Basic Block Vector (BBV), which could show the proportion of basic block executions during a given interval, to detect representative phases. In addition, we have implemented a modified version of the SimPoint tool for identifying these representative phases, which is described in the following:

***1) Profiling basic block*:** this task is to obtain the basic block vector (BBV) for each phase interval with a fixed amount of instructions. For the interval length setting, we still followed the discussions described in the previous chapter to

choose appropriate settings for different applications and achieve a good tradeoff between the total process time and the amount of phase intervals. For all intervals throughout a program execution, we employed BBV to record the number of times each basic block executed in a phase interval.

2) BBV comparison**:** this task is to compare BBV which are identified in the previous task. We employed the Manhattan distance of basic blocks to compare how closely related two phase intervals are to one another, and find out their differences. For vector *a* and vector *b* in D-dimensional space, the distance can be computed as:

$$\text{Manhattan Dist}(a,b) = \sum_{i=1}^{D} | a_i - b_i |$$

3) Phase classification**:** this task is to classify runtime phase behaviors. In order to detect the amount of resemblance between different phase intervals, a basic block similarity matrix was defined to represent the Manhattan distance between all pairs of basic block vectors. And we made use of the similarity matrix to classify phase-based behaviors.

4) Picking representative phases**:** this task is to identify representative phases from each cluster. By checking the similarity between BBV, we generated similarity groups among these executed phase intervals. Eventually, we clustered these intervals, which have similar runtime behaviors, into phase groups.

By using these identified phases, we can represent the full program's execution characteristics through analyzing only a single sample from each cluster.

7.1.2.2 Runtime power behavior identification

This step is going to estimate the detailed runtime power behaviors for these representative phases, which are identified in the previous step, to represent the whole program power behaviors with a reasonable accuracy.

As known, when an application is executed by a microprocessor, its power dissipation could be calculated by

$$Power = \frac{Energy}{Time} = \frac{E}{S} = \frac{E}{C} \cdot \frac{C}{S} = EPC \cdot CPS \quad (7.1)$$

Where *Power* is the average power dissipation for an execution period, *EPC* is the average energy per cycle, and *CPS* is the cycles per second ratio for a CPU.

As shown in the above formula, it is obvious to see that: for a processor with a fixed frequency (*f*), the *CPS* is the same at anytime during an application execution, and the power dissipation is in a direct proportion to the *EPC*. Therefore, in our method, we make use of the *EPC* metric to show the detailed runtime power behaviors of an application.

In the first step, we have already identified representative phases for a complete program execution. Now, for every one of these identified phase, we can estimate its detailed runtime power behavior by measuring its *EPC* parameter. The detailed implementation is described in the following.

Firstly, we employ our modified Wattch tool to calculate the *EPC* parameter for small sample slices within a phase. Supposing that in an application, there are *M* representative phases, which are identified in the first step. For each phase *Pj* (*j* =

1,...,*M*), there are *N* small sample slices to measure its runtime power dissipation value. For each sample *Si* (i = 1,...,*N*), there is a corresponding data [*EPC*] measured during its execution. Then, for every phase, we make use of a power vector to save the detailed runtime power behavior in terms of *EPC*. Thus, for a phase *Pj*, we collect a "power vector", PV_j, as the detailed runtime power values for the total *N* samples belonging to the phase P_j. Eventually, for the complete application execution, we can obtain the total representative power behaviors, by orderly combining the corresponding power vector for the M representative phases.

7.1.2.3 Total energy consumption estimation

This step is to estimate the total energy consumption for a complete application execution based on our selected representative phases.

For every identified phase, we not only gather a power vector for its detailed runtime power behaviors, but also measured its total energy consumption. Therefore, for an application with *M* representative phases, the whole program energy consumption is estimated by

$$E_{est} = \sum_{i=1}^{M} E_i \times W_i \qquad (7.2)$$

Where *Ei* is the measured energy consumption of the *i*th phase, W_i is the weight for the *i*th phase, and *M* is the total number of phases.

To evaluate the accuracy of the total energy consumption estimated by our identified phases, we compare it with the real energy consumption measured during the complete application execution. To do this, we make use of an error rate to present

the difference between the estimated and measured total energy consumption, which is calculated as

$$error = \frac{|\,energy_estimated - energy_measured\,|}{energy_measured} \qquad (7.3)$$

In the next section, we will demonstrate the overall error rates between the estimated and measured total energy consumption in our simulation results.

7.1.3 Results

We employed a modified version of Wattch to perform our experiments and collected the execution results. Our modifications to Wattch provide detailed runtime power dissipation information sampling. The micro-architectural parameters of the baseline model of Wattch are shown in Table 4.2 in Chapter 4. In order to validate the feasibility and accuracy of our identified phases, we selected two benchmark applications from the MediaBench and two benchmark applications from the SPEC CPU2000 to perform experiments. And the reference workloads supplied by the two benchmark suites were executed in experiments.

7.1.3.1 Runtime power behavior estimation

To verify the accuracy of runtime power behaviors estimated by identified phases, we compare it with the real power behavior measured during the complete application execution. Figure 7.1 and 7.2 show our experiment results for the runtime power behaviors: Figure 7.1(a) and 7.2(a) present the measured runtime power behaviors for the complete program execution, and Figure 7.1(b) and 7.2(b) present the estimated

runtime power behaviors by the representative phases. In these figures, the curve demonstrates the power behaviors of benchmark applications; each power value presents the power dissipation of a sample interval with a fixed amount of continuous instructions. Obviously, Figure 7.1(b) and 7.2(b) have much fewer samples than Figure 7.1(a) and 7.2(a) because our chosen phases are only a small part of the complete application execution workload.

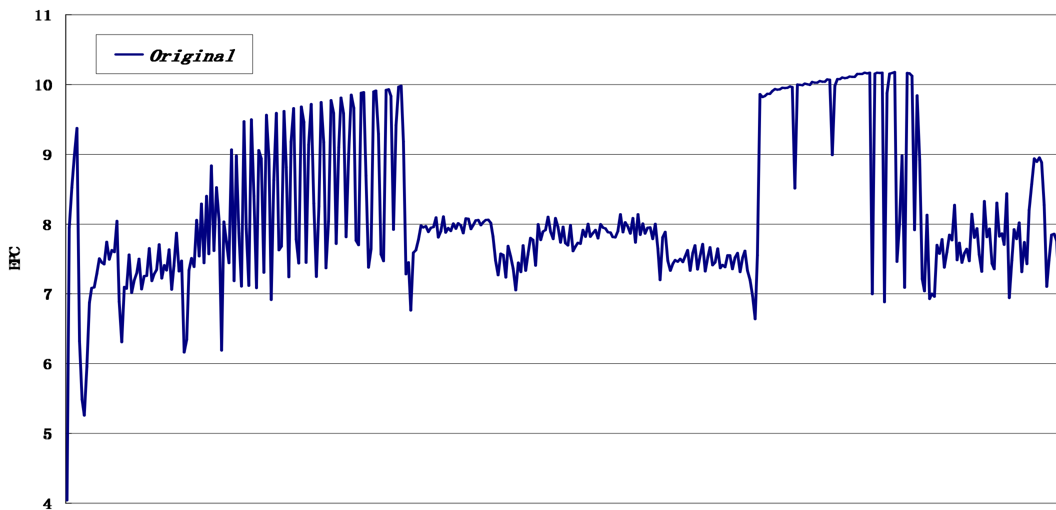


Fig. 7.1(a): Measured runtime power behavior for “vortex”

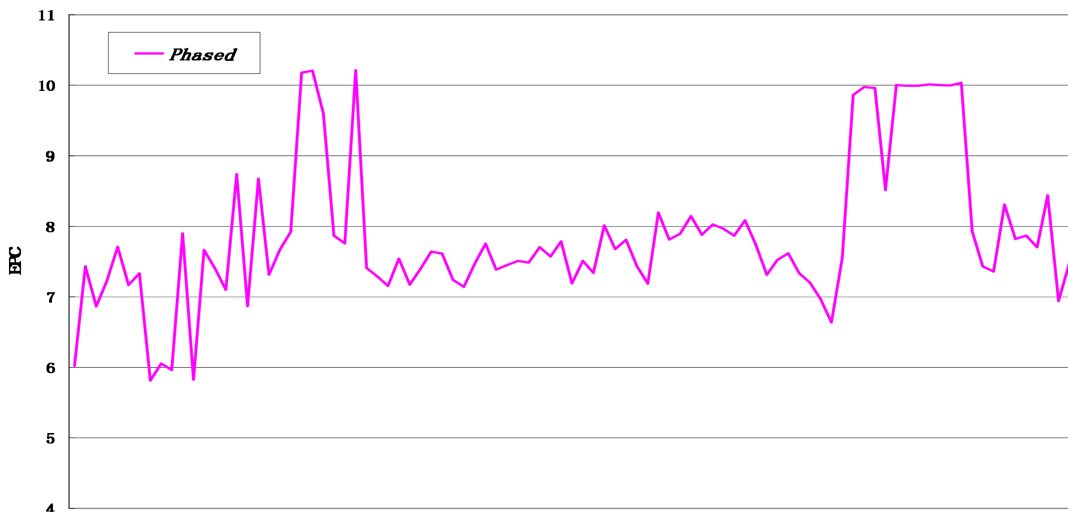


Fig. 7.1(b): Phase estimated power behavior for “vortex”

From Figure 7.1(a) and 7.1(b), we can see that: the runtime power behaviors estimated by these representative phases match well with that measured through the complete application execution.

As shown in Figure 7.1(a), for “*vortex*”, there are some frequent and distinct changes in its real runtime power behaviors. In Figure 7.1(b), our estimated runtime power behaviors highly repeated all the distinctive characteristics. Figure 7.1(a) shows that: the whole program execution can be roughly partitioned into 4 stages according to its power dissipation value. In Figure 7.1(b), as expected, the power behaviors estimated by the representative phases well captured the 4 stages of power behaviors. In addition, our phase-based power estimation well represented the periodic and highly varying areas in the real measured power behaviors. Lastly, we can see that: In both figures for “*vortex*”, they have nearly the same value range for power behaviors, varying from 6 to 10.

As shown in Figure 7.2(a) and 7.2(b), for “*adpcmencode*”, there is infrequent and small change in its real power behavior. Nevertheless, our phase estimated results also well represent its real power dissipation very well. As shown in Figure 7.2(a), for the real power dissipation value, there is only a small range between 7.0 and 7.6. Then, as we can see from Figure 7.2(b), our phase estimated data exactly repeated the range of power value. Moreover, as shown in the figure, our estimated power behaviors captured not only the smooth period with small variations (7.1-7.2), but also the big change area with peak power dissipation (7.6).

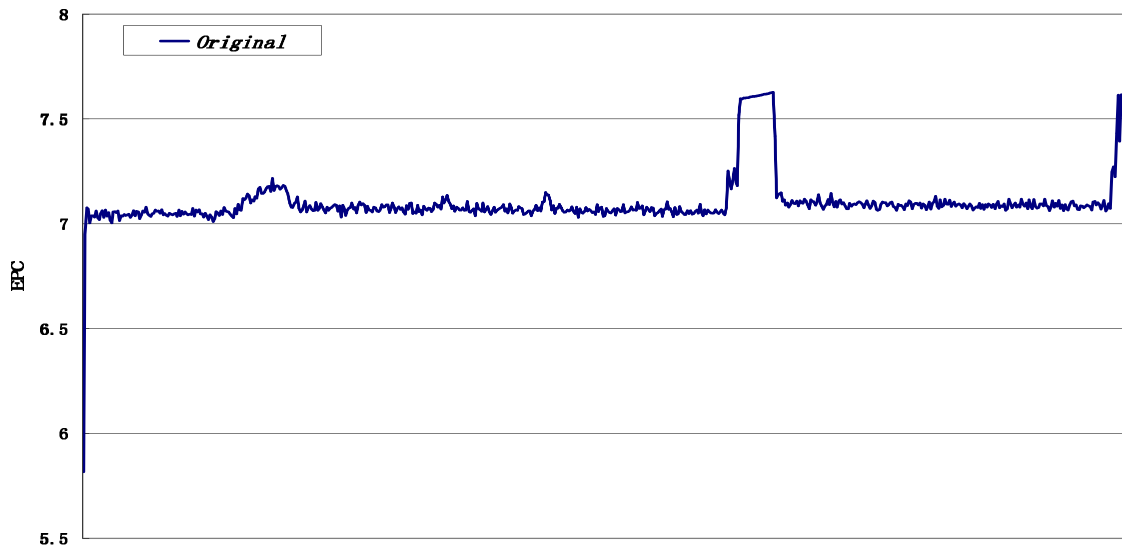


Fig. 7.2(a): Measured runtime power behavior for "adpcmencode"

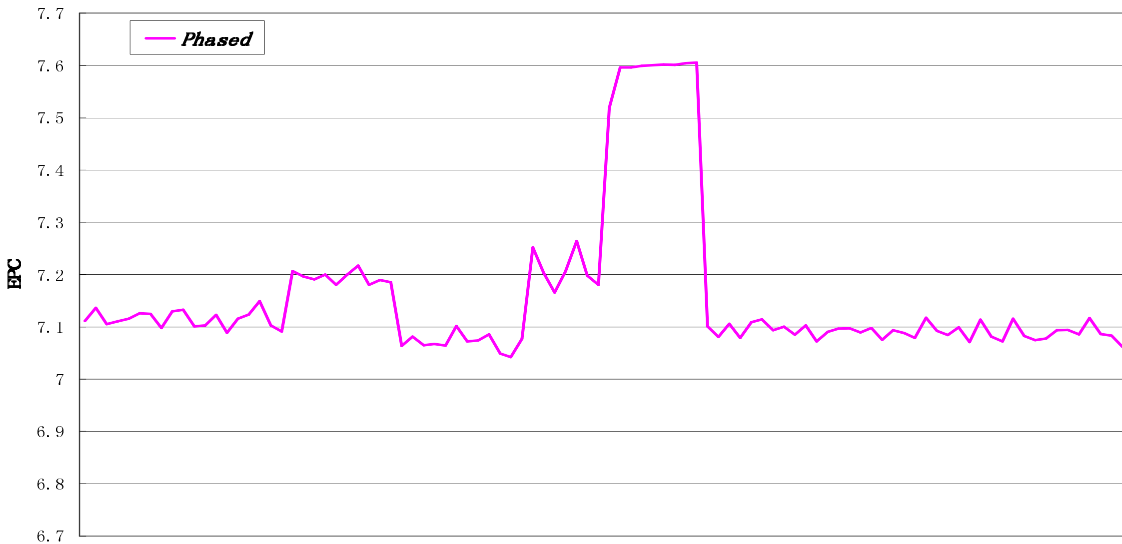


Fig. 7.2(b): Phase estimated power behavior for "adpcmencode"

All the above findings indicate that our phase-based methodology could achieve a considerable good result not only in the power data accuracy, but also in its representation of the full program power behavior characteristics. Therefore, we can draw a conclusion: it is an accurate and to efficient way to reconstruct the full program runtime power behaviors by estimating the power behaviors of a small set of

representative phases.

7.1.3.2 Total energy consumption estimation

As discussed in Section 7.1.3.3, to determine the accuracy, we compared the total energy consumption results estimated by our selected phases against that measured in the complete application execution. Using the formula presented in Section 7.1.2.3, we calculated an error rate, which is based on the comparison between the estimated and measured total energy consumption of applications.

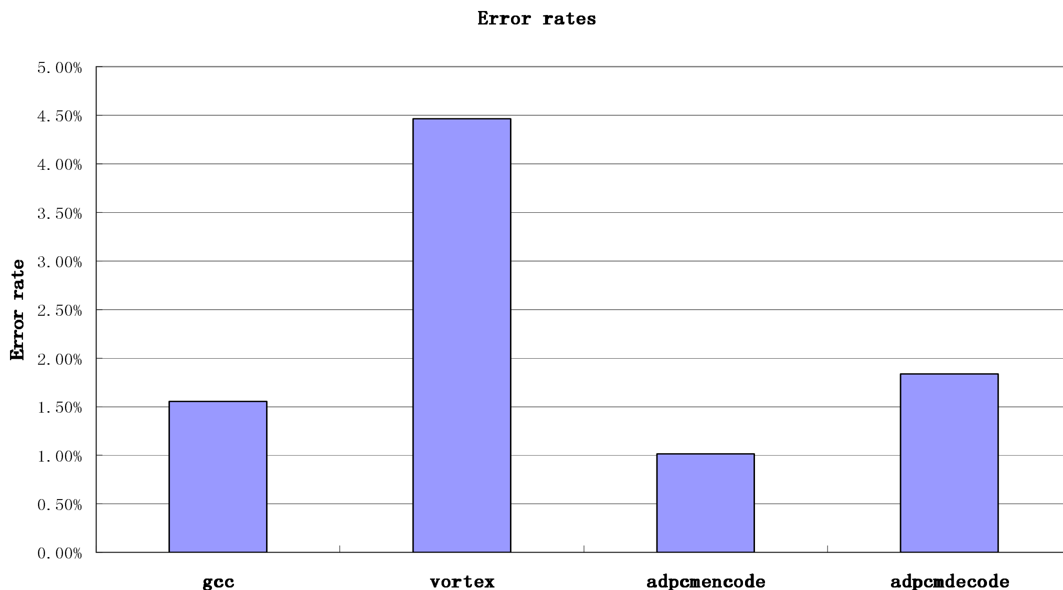


Fig. 7.3: Error rates

Figure 7.3 demonstrates the calculated error rates of our estimated total energy consumption by using phase-based estimation. As shown in the figure, the result implies that our phase-based methodology could estimate the total energy consumption for a complete application execution with very high accuracy.

For all the four applications, comparing to real measured energy consumption for

the complete application execution, the error rate of the total energy consumption estimated by our mechanism vary from 1% to 4.5%, with an average of 2.2%. As shown, "*vortex*" estimated the results with the highest error rate by 4.47%, while "*adpcmencode*" did with the lowest error rate by 1%. The observation is consistent to the finding of the runtime power behaviors in the above section. As shown in Figure 7.1 and 7.2, "*vortex*" shows highly varying behavior in its runtime power dissipation while "*adpcmencode*" shows a very flat behavior in its runtime power dissipation. Thus, the total energy consumption estimation for "*vortex*" is more difficult than that for "*adpcmencode*", resulting in a higher error rate. Overall, for all applications, the error rates of the total energy consumption estimation are very small, which indicates that the total energy consumption measured by our phase-based methodology is an exact estimation to the real measured results.

7.1.4 Conclusion

Application power behavior is helpful to evaluate the power optimization designs. In this section, we successfully presented an efficient power behavior identification methodology for a single application based on their runtime phase estimation. Firstly, we made use of a phase analysis technique to identify a small set of application execution intervals, which could represent characteristics of the complete application execution. Then, based on identified representative phases, we estimated the detailed runtime power behavior and total energy consumption to represent the power characteristics of a whole application. Our simulation results revealed that our

phase-based mechanism provides a practical and effective way to analyze application power behavior and assure the accuracy of results. We believe that our mechanism can be used to evaluate and observe power optimization opportunities in microprocessor power optimization designs.

7.2 Data Dependence Length Identification Method

7.2.1 Introduction

As known, data dependence analysis was a fundamental technique employed in compilers to perform optimization transformations. In recent years, data dependence analysis became an attractive topic for microprocessor architecture research to exploit micro-architecture characteristics in a program, such as dynamic branch prediction, memory access estimation, and out-of-order superscalar execution.

In the past years, the exploitation of data dependence information usually was estimated by static analysis at the source code level [99], such as FORTRAN, C and Assembly code [100]. However, only static data dependence analysis in the source code is insufficient for the micro-architecture level research, thus researchers proposed to perform data dependence analysis during a program execution period to identify its dynamic data dependence information. For example, Lei Chen at al [101] developed a mechanism to dynamically track data dependence of pipeline instructions when a program is executing.

In this paper, we present an efficient methodology to identify dynamic data dependence characteristic among program runtime in-flight instructions by using a phase analysis technique. To identify dynamic data dependence information of a program, first of all, we detect its representative execution phases, which is a small set of execution intervals with a fix amount of instructions. These representative phases are used for data dependence analysis instead of the complete execution of a program. We

employed a modified version of SimPoint to identify these representative phase intervals. Then, based on the tracked runtime information of these phases, we implemented an approach to exploit the data dependence length (DDL) within both basic blocks and phases. Finally, we characterized dynamic data dependence characteristic for the whole program based on our identified DDL information of the representative phases. This data dependence length identification method presented in this section is built upon my earlier work in [104].

7.2.2 Methodology

The goal of this method is to identify dynamic data dependence information of a program at runtime. As shown in Figure 7.4, there are three steps for our DDL identification method. Firstly, we perform a program on the SimpleScalar tool and gather its runtime information. Then, based on the traced execution results, we attempt to detect some representative phases for the complete application execution. Finally, we identify the DDL through analyzing these representative phases. In the following, we will describe the detailed implementations of these steps.

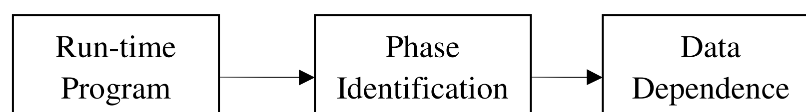


Fig. 7.4: Steps for DDL identification method

7.2.2.1 Phase Identification

This first step is to identify some potential phases, which could be a representative of the complete program execution. As is presented in Section 7.1.2.1, we have

implemented an approach to identify such a small set of execution intervals that are representative of the entire application execution. Thus, we also use it as the phase identification method in this step, and give a briefly description about it in the following.

Firstly, we use basic block vector (BBV) to estimate the proportion of basic block executions for each phase interval throughout a program execution. At the same time, for each interval, we employed a basic block instruction vector (BBIV) to trace runtime instructions of each basic block, which will be used for data dependence length analysis later. Then, based on the identified BBV, we employed the Manhattan distance of basic blocks to compare how closely related two execution intervals are to one another, and find out their differences. Thirdly, we made use of a basic block similarity matrix to identify phase-based behaviors. Finally, we generated similarity groups of phases by clustering these intervals, which have similar runtime behaviors, into groups.

By using the above phase analysis method, we finally detect these representative phases and employ them to identify dynamic data dependence information in the next step.

7.2.2.2 DDL identification

After identifying these representative phases, the second step is to analyze their dynamic data dependence information, which could be representative to that of the complete program execution.

In general, data dependence shows an ordering relationship between sequences of instructions, and data dependence length (DDL) indicates the length of a data dependence chain relative to a particular instruction. In the previous step, we have obtained a basic block instruction vector (BBIV), which is used to trace detailed instructions of each basic block. Thus, we could identify the DDL for each basic block through analyzing its BBIV.

Our traced BBIV is a two dimensional array [X, Y], where X recodes each basic block executed in a phase, while Y recodes all instructions of basic blocks. In the following, we began our data dependence length identification by analyzing instructions contained within Y of each BBIV. Figure 7.5 shows a simple example of a DDL analysis in a basic block.

```
1. ld d1, d2
2. ld d3, d4
3. add d1 d3, d5
4. ld d6, d7
5. ld d8, d9
6. mul d6, d8, d6
7. sub d5, d6, d5
8. ret d10, (d5)
```

	d1	d2	d3	d4	d5	d6	d7	d8	d9	d10	DDL
1	D	X									0
2			D	X							0
3	X		X		D						2
4						D	X				0
5								D	X		0
6						D		X			2
7					D	X					4
8					X					D	1

Fig. 7.5: DDL Example

As is shown in Figure 7.5, we made use of a DDL table to present array Y of every BBIV. The depth of DDL table is the number of instructions of Y, and the width of DDL is the number of real data of Y. Thus each instruction occupies a row in the DDL table. For clarity, we will refer to a data in the column of the DDL as a data entry.

Also, we refer to the instruction information occupying a row in the DDL as an instruction entry. Entries with an 'X' indicate that the data is read and the instruction depends on it, while entries with a 'D' indicate that the data is written and it depends on that instruction.

As is shown in the last column of DDL table, for each instruction, we identify its data dependence chain length. The DDL table contains all instructions of a basic block as nodes with a data dependence value. However, when an instruction commits, it must be eliminated from all data dependence chains length analysis because its data value is now ready for immediate use. Therefore, DDL is defined as the distance from one instruction to the latest one if there is data dependence between them. For an instruction that depends on more than one data, its DDL is defined as the maximum length of its data. For example, the third instruction depends on both d1 and d3, and d1's dependence length is 2 whereas d3's dependence length is 1. So, DDL of the third instruction is 2. For a basic block, the minimum execution time usually is to finish an instruction with the maximum DDL. Therefore, the DDL for a basic block is defined as the maximum dependence length of its instructions. Furthermore, the total DDL of a basic block is the sum of each instruction's dependence length.

However, identifying data dependence length of all basic blocks in a phase is a very computationally expensive task since a phase may have hundreds of basic blocks. Therefore, we developed a data dependence length identification algorithm, which is presented in Figure 7.6.

As shown in Figure 7.6, Lines 1 through 13 shows the main loop of the algorithm

which will be applied for each basic block. In lines 2 through 4 we create a data dependence node of the basic block for an instruction. In lines 5 through 10, a reverse search function is performed to get instruction's dependence length. In lines 11 through 13, it is to calculate the maximum data dependence length and the total dependence length of a basic block. The subroutine in lines 14 through 19 will identify each data of an instruction and figure out its dependence length.

```
1 for each (basic block) {
2           for each instruction {
3                 add instruction as node
4           }
5           for each node {
6                 for each register {
7                       get_register_length(cur_register)
8                 }
9                 node_length = maximum_register_length
10          }
11          maximum_basic_block_length = maximum_node_length
12          total_basic_block_length = sum(node_length)
13 }
14 get_register_length(cur_register) {
15          if cur_register has no predecessor
16                cur_register_length = 0
17          else cur_register has a predecessor
18                predecessor_length = get_register_length(predecessor)
19 }
```

Fig. 7.6: Pseudocode for DDL Identification Algorithm

7.2.3 Results

In experiments, we used the SimpleScalar tool to perform an application and collected its execution results. From experiment results, we identified data dependence length of basic blocks within phases as well as the entire program execution. To determine

these results, we compared the experiment results obtained by representative phases with that measured through the complete program execution, both on the same baseline processor. Detailed experiment setup information is presented in Chapter 4.

7.2.3.1 MAX_DDL

We make use of the metric MAX_DDL to show the maximum DDL of one basic block. The value of MAX_DDL is calculated by the DDL identification algorithm,

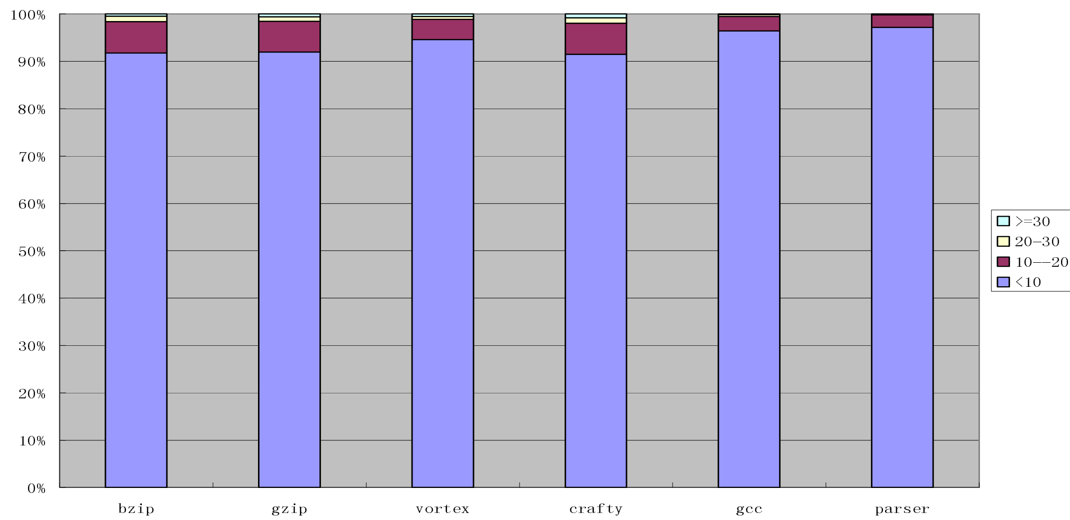


Fig. 7.7(a): MAX_DDL of the complete application execution

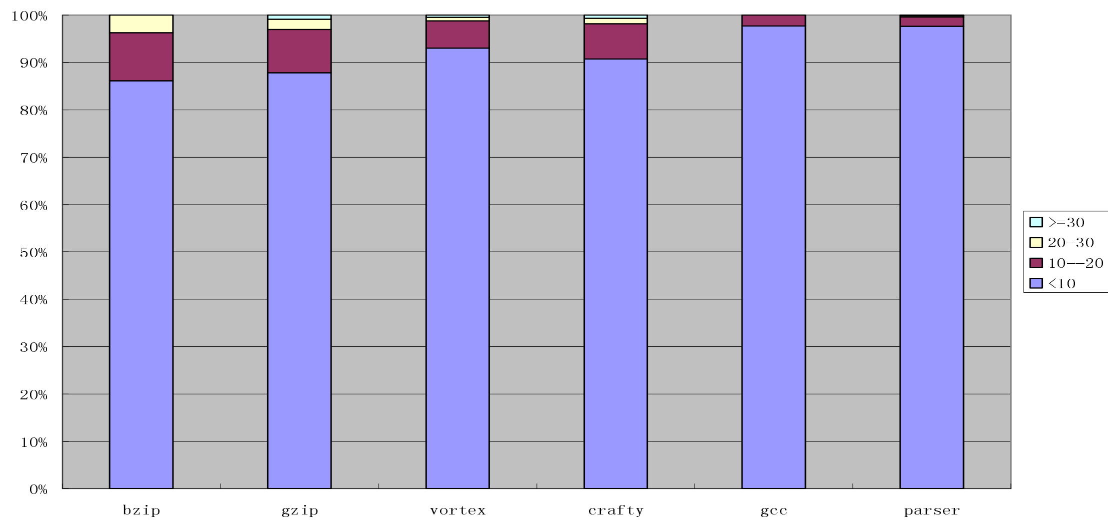


Fig. 7.7(b): MAX_DDL of the representative phases

which is presented in the previous section. Figure 7.7 (a) and (b) show the MAX_DDL results obtained form both the representative phases and the complete application execution, for different benchmark applications.

From both Figure 7.7(a) and Figure 7.7(b), we can see that: the largest fraction of basic blocks has a maximum DDL lower than 10; some basic blocks have maximum DDL lower than 20, and only very few basic blocks have maximum DDL high than 20. In general, the MAX_DDL is directly proportional to the amount of instructions in a basic block. As found, most basic blocks have less than 10 instructions, and there are very few basic blocks that have more than 20 instructions. Thus, it is obvious that increasing the amount of instructions of a basic block will consequently increase its MAX_DDL characteristic.

7.2.3.2 TOTAL_DDL

To determine the dynamic data dependence information of an application, we employed a metric of TOTAL_DDL to demonstrate the sum of DDL value for an application execution interval. The results are also obtained by our DDL identification algorithm. Figure 7.8(a) and 7.8(b) show the difference of the total DDL results between the representative phases and the complete application execution.

As we can see from Figure 7.8(a) and 7.8(b), for all these benchmark applications, they have similar TOTAL_DDL results to their MAX_DDL results: most of basic blocks has the total DDL value lower than 10; some basic blocks have total DDL lower than 20, and very few basic blocks have summed DDL higher than 20. The

result is reasonable since the TOTAL_DDL is the sum of DDL value within one basic block.

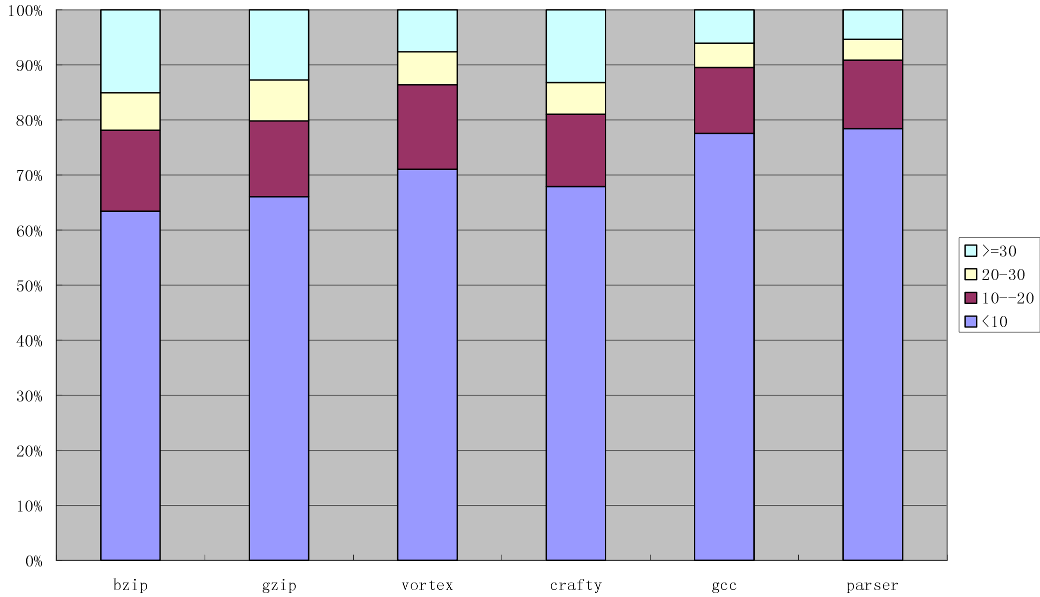


Fig. 7.8(a): TOTAL_DDL of the complete application execution

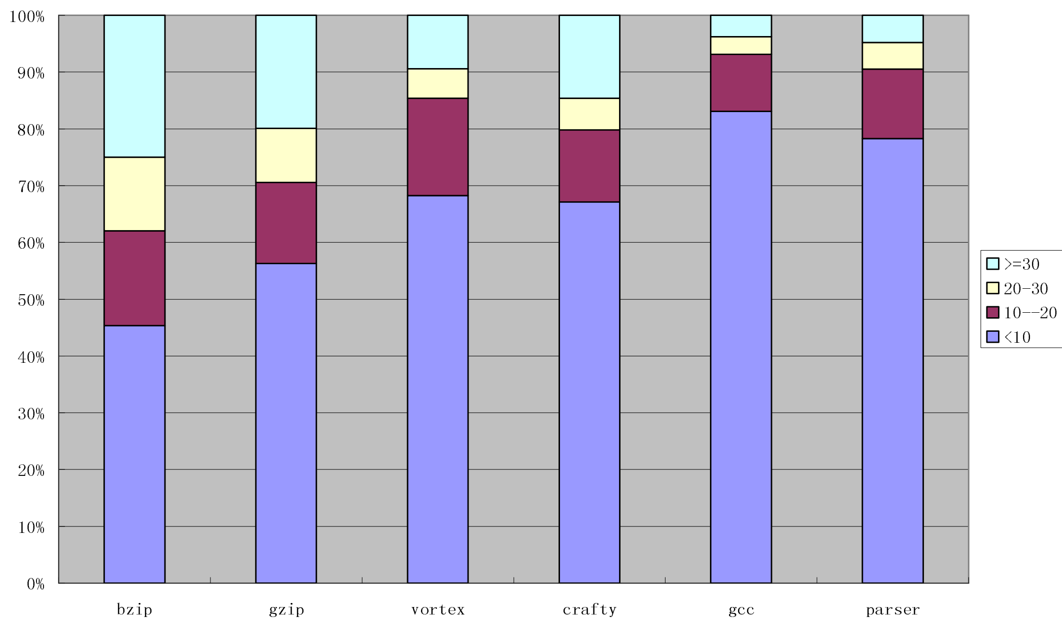


Fig. 7.8(b): TOTAL_DDL of the representative phases

Another observation is that the similarity between the DDL results obtained from the representative phases and those from the complete program execution. From all

the above four figures, we can see that: for any application, its DDL results of chosen phases are very similar to that of its complete execution. This finding indicates that our phase-based methodology could achieve a considerable good result not only in the DDL accuracy, but also in its representation of dynamic data dependence character of the full program.

7.2.6 Conclusion

In this section, we have presented an efficient dynamic data dependence identification methodology by using a phase analysis technique. Firstly, we used a phase identification technique to detect representative execution phases for a program. Then, based on these representative phases, we employed a DDL identification approach to obtain runtime data dependence information among in-flight instructions of a program. Finally, we characterized dynamic data dependence for the whole program by using our identified DDL information obtained from the representative phases. Our simulation results revealed that our phase-based data dependence length identification methodology can improve analysis efficiency at the same time assuring accuracy of results.

7.3 Summary

In this chapter, we have presented two efficient methods to identify runtime power behaviors and data dependence length (DDL) of an application, which are two useful micro-architecture parameters to evaluate our proposed low-power designs in the previous chapters. Both of the two methods employed a phase analysis technique to rapidly identify the target micro-architecture parameters, and demonstrated a considerable accuracy in their experiment results.

Chapter 8

Conclusions and Future Work

In this chapter, we shall summarize our work in Section 8.1. In Section 8.2, we shall briefly describe our main contributions of this thesis. Finally, in Section 8.3, we shall suggest some possible directions for future work.

8.1 Summary of Work

In this thesis, our main aim is to implement new low-power design schemes for the microprocessor to reduce its runtime power dissipation. For this goal, we first investigated various technologies for reducing power dissipation of the microprocessor, and then we proposed to address the issue of microprocessor power reduction at the micro-architecture level, finally we presented a realistic analysis model to discuss potential opportunities for power reduction during application execution period. Based on our analysis model, we have designed two low-power schemes at the micro-architecture level: the IPC-driven online identification and predication power reduction method and the IPC-driven offline code analysis and reconfiguration power reduction method. The two designs employed the same micro-architecture parameter-IPC, as the performance indicator to identify appropriate points during application execution to scale the voltage and frequency of the microprocessor for reducing power dissipation.

Our first design, the IPC-driven online power reduction method, has achieved

good results in experiments with high energy savings and small performance degradation. In this design, based on execution results obtained from the current interval, we calculated its performance activity level in terms of the IPC value, and then predicted the coming interval's performance requirement to dynamically scale the voltage and frequency of the processor at an appropriate level. Our simulation results revealed that the interval-based identification and prediction approach successfully achieved energy savings by an average of 29% with small performance degradation (8%).

Our second design, the IPC-driven offline power reduction method, has also been proved to be a practical and effective way to save significant amounts of energy while maintaining the original performance. In this design, based on execution results obtained in application training runs, we first identified code sections having opportunities in terms of their IPC value to reduce power dissipation, and then profiled applications to dynamically scale the voltage and frequency of the processor at appropriate points during their execution. As shown in our experiment results, this code analysis and reconfiguration design finally achieved energy savings by an average of 40% with little performance degradation (6%).

Beside the two micro-architecture level low-power designs, we also presented two methods to identify two micro-architecture parameters: runtime power behavior and data dependence length of applications. The two micro-architecture parameters are helpful to evaluate the two low-power designs proposed by us. Both methods employed a phase-based analysis technique to speedily identify the interesting

micro-architecture parameters through a small set of representative execution phases for a program, and the experiment results demonstrated both methods could identify the target micro-architecture parameter accurately.

8.2 Summary of Contributions

The main contributions presented in this thesis are as follows: Firstly, focusing on the micro-architecture level, a closer level to microprocessor, we have addressed the microprocessor power dissipation issue and accomplished two effective low-power design schemes to reduce power dissipation of the processor. We also showed that the micro-architecture level is a practical and efficient level to address power optimization opportunities for the microprocessor. Secondly, we successfully demonstrated a realistic analytical model to discuss how to estimate potential code regions in an application which have opportunities to optimize power dissipation of the microprocessor. Lastly, as shown in our experimental results, to achieve a successful power-performance trade-off optimization, the micro-architecture parameter (IPC) employed in our designs is shown to be a good performance indicator for DVS scheduling.

In summary, working at the micro-architecture level and using a useful and realistic analytical model, our proposed low-power designs have successfully employed the micro-architecture parameter (IPC) as a performance indicator to direct DVS scheduling at appropriate points and eventually achieved good power-performance trade-off.

8.3 Future work

Although our methods only focused on addressing the power dissipation and optimization issues of the microprocessor, the micro-architecture parameters identified and employed in our designs, such as IPC and DDL, could also be used in software power/energy optimization. From our proposed methods and designs, there are several future/potential research directions for low-power design in the high level software domain.

- Software compiling optimization

  There are many conventional optimization techniques in the process of compiling a program into a binary execution file, such as critical path and data dependence identification. However, these former strategies mostly focus on improving the performance of a program, but do not consider its energy consumption. Therefore, during the compiling period, our proposed micro-architecture parameters, such as our defined IPC and DDL, could be used to identify opportunities for energy savings in a program and optimize the final binary code to reduce the whole program energy consumption.

- Software architecture optimization

  As believed by researchers in the area of low-power designs, the efficiency of analysis and the amount of energy savings obtainable are much larger at higher levels. Thus, there should be many opportunities to address low power design at a higher level of software architecture. For example, since we could estimate opportunities among program code sections by using the micro-architecture

parameter-IPC, we could also identify some experiential code sections group with power reduction opportunities as potential “code bank”, which could be referred to by later software designers, and let them know possible chances to save energy in the architecture design period.

Bibliography

[1] E. Grochowski and M. Annavaram. Energy per Instruction Trends in Intel® Microprocessors. Intel Technology Magazine, Mar 2006.

[2] T. Starner. The challenges of wearable computing. IEEE Micro, pp. 44-52, July 2001.

[3] M. Wilson, C. Zawodzinski, and M. Daugherty. Small battery-fuel cell alternative technology development. In Proceedings of DOE Hydrogen Program, 2001.

[4] K. Lahiri, A. Raghunathan, S. Dey, and D. Panigrahi. Batter-driven system design: A new frontier in low power design. In Proceedings of Asia South Pacific Design Automation Conference/International Conference on VLSI Design, January 2002.

[5] C. Small. Shrinking devices put the squeeze on system packaging, EDN, vol. 39, no. 4, pp. 41-46, February 1994.

[6] S. H. Gunther, F. Binns, D. M. Carmean, and J. C. Hall. Managing the impact of increasing microprocessor power consumption. Intel Technology Journal, Q1, 2001.

[7] (2001) International Technology Roadmap for Semiconductors. International SEMATECH, Austin, TX. http://public.itrs.net/.

[8] K. Roy, S. Mukhopadhyay, and H. Mahmoodi-Meimand. Leakage Current Mechanisms and Leakage Reduction Techniques in Deep-Submicrometer CMOS Circuits. In Proceedings of the IEEE, Vol. 91, No.2, February 2003.

[9] R. Pierret. Semiconductor Device Fundamentals. MA: Addison-Wesley, 1996,

CH. 6, pp. 235-300.

[10] Y. Taur. CMOS scaling and issues in sub-0.25 um systems, in Design of High-Performance Microprocessor Circuits. A. Chandrakasan, W.J. Bowhill, and F. Fox, Eds. Piscataway, NJ: IEEE, 2001, CH. 2, pp. 27-45.

[11] S. Thompson, P. Packan, and M. Bohr. MOS scaling: Transistor challenges for the 21st century. Intel Technology Journal, Q3, 1998.

[12] D. Fotty. MOSFET Modeling with SPICE. Englewood Cliffs, NJ: Prentice-Hall, 1997, CH. 11, pp. 396-397.

[13] T. Iwamoto et al. A highly manufacturable low power and high speed HfSiO CMOS FET with dual poly-Si gate electrodes. In Digest of Technical Papers of International Electron Devices Meeting (IEDM), 2003, pp.27.5.1-27.5.4.

[14] S. Thompson, P. Packan, and M. Bohr. Linear versus saturated drive current: Tradeoffs in super steep retrograde well engineering. In Digest of Technical Papers of Symposium on VLSI Technology, 1996, pp. 154-155.

[15] S. Venkatesan, J.W. Lutze, C. Lage, and W. J. Taylor. Device drive current degradation observed with retrograde channel profiles. In Proceedings of International Electron Devices Meeting, 1995, pp. 419-422.

[16] J. Jacobs and D. Antoniadis. Channel profile engineering for MOSFET's with 100 nm channel lengths. IEEE Transaction of Electron Devices, vol. 42, pp. 870-875, May 1995.

[17] W. Yeh and J. Chou. Optimum halo structure for sub-0.1 μm CMOSFET's. IEEE Transaction of Electron Devices, vol. 48, pp. 2357-2362, Oct. 2001.

[18] V. De, Y. Ye, A. Keshavarzi, S. Narendra, J. Kao, D. Somasekhar, R. Nair, and S. Borkar. Techniques for leakage power reduction, in Design of High-Performance Microprocessor Circuits. NJ: IEEE, 2001, CH.3, pp. 48-52.

[19] Y. Ye, S. Borkar, and V. De. New technique for standby leakage reduction in high-performance circuits. In Digest of Technical Papers of Symposium on VLSI Circuits, 1998, pp. 40-41.

[20] M. C. Johnson, D. Somasekhar, and K. Roy. Leakage control with efficient use of transistor stacks in single threshold CMOS. In Proceedings of ACM/IEEE Design Automation Conference, 1999, pp. 442-445.

[21] M. Powell, S.H. Yang, B. Falsafi, K. Roy, and T.T. Vijaykumar. Gated-V_{DD}: A Circuit Technique to Reduce Leakage in Deep-Submicron Cache Memories. In Proceedings of ACM/IEEE International Symposium on Low Power Electronics and Design, 2000.

[22] S. Mukhopadhyay, C. Neau, T. Cakici, A. Agarwal, C. H. Kim, and K. Roy. Gate Leakage Reduction for Scaled Devices Using Transistor Stacking. IEEE Transaction on VLSI Systems, Vol. 11, Issue 4, pp. 716-730, Aug. 2003.

[23] Z. Chen, M. Johnson, L. Wei, and K. Roy. Estimation of standby leakage power in CMOS circuits considering accurate modeling of transistor stacks. In Proceedings of International Symposium on Low Power Electronics and Design, 1998, pp. 239-244.

[24] S. Narendra, S. Borkar, V. De, D. Antoniadis, and A. Chandrakasan. Scaling of Stack Effect and its Application for Leakage Reduction. In Proceedings of the

International Symposium on Low Power Electronics and Design, 2001, pp. 195-200.

[25] Y. Liu and Z. Q. Gao. Timing analysis of transistor stack for leakage power saving. In Proceedings of 9th International Conference on Electronics, Circuits and Systems, 2002, pp.41- 44.

[26] N. Sirisantana, L. Wei, and K. Roy. High-performance low-power CMOS circuits using multiple channel length and multiple oxide thickness. In Proceedings of International Conference of Computer Design, 2000, pp. 227-232.

[27] Y. Taur and T. H. Ning. Fundamentals of Modern VLSI Devices. New York: Cambridge Univ. Press, 1998, CH. 4, pp. 194.

[28] H. Makino, Y. Tsujihashi, K. Nii, C. Morishima, Y. Hayakawa, T. Shimizu, and Arakawa. An auto-backgate-controlled MT-CMOS circuit. In Proceedings of International Symposium on Low Power Electronics and Design, 1998, pp.42-43.

[29] N. Tripathi, A. Bhosle, D. Samanta, and A. Pal. Optimal assignment of high threshold voltage for synthesizing dualthreshold CMOS circuits. In Proceedings of 14th International Conference on VLSI Design, 2001, pp. 227-232.

[30] T. Inukai, T. Hiramoto and T. Sakurai. Variable threshold voltage CMOS (VTCMOS) in series connected circuits. In Proceedings of International Symposium on Low Power Electronics and Design, 2001, pp. 201-206.

[31] S. Mutoh, T. Douseki, Y. Matsuya, T. Aoki, S. Shigematsu, and J. Yamada. 1-V power supply high-speed digital circuit technology with multi-threshold voltage CMOS. IEEE Journal of Solid-State Circuits, vol. 30, pp. 847-854, Aug. 1995.

[32] J. Kao, A. Chandrakasan, and D. Antoniadis. Transistor sizing issues and tool for multi-threshold CMOS technology. In Proceedings of ACM/IEEE Design Automation Conference, 1997, pp. 495-500.

[33] F. Assaderaghi, D. Sinitsky, S. Parke, J. Bokor, P. K. Ko, and C. Hu. A dynamic threshold voltage MOSFET (DTMOS) for ultra-low voltage operation. In Digest Technique Papers of International Electron Devices Meeting, 1994, pp. 809-812.

[34] C. H. Kim and K. Roy. Dynamic V_{th} scaling scheme for active leakage power reduction. In Proceedings of Conference on Design, Automation and Test Europe, 2002, pp. 163-167.

[35] A. J. Bhavnagarwala, B. L. Austin, K. A. Bowman, and J. D. Meindl. A minimum total power methodology for projecting limits on CMOS GSI. IEEE Transaction on VLSI System, Vol. 8, pp. 235-251, June 2000.

[36] S. Tyagi et al. A 130 nm generation logic technology featuring 70nm transistors, dual Vt transistors and 6 layers of Cu interconnects. In Digest Technique Papers of International Electron Devices Meeting, 2000, pp. 567-570.

[37] M. Takahashi et al. A 60-mw MPEG4 video codec using clustered voltage scaling with variable supply-voltage scheme. IEEE Journal of Solid- State Circuits, vol. 33, pp. 1772-1780, Nov. 1998.

[38] Y. Kanno, H. Mizuno, K. Tanaka, and T. Watanabe. Level converters with high immunity to power-supply bouncing for high-speed Sub-1-V LSI's. In Digest Technique Papers of Symposium on VLSI Circuits, 2000, pp. 202-203.

[39] T. Fuse, A. Kameyama, M. Ohta, and K. Ohuchi. 0.5 V power-supply scheme for

low power LSI's using multi-Vt SOI CMOS technology. In Digest Technique Papers of Symposium on VLSI Circuits, 2001, pp. 219-220.

[40] L. R. Carley and A. Aggarwal. A completely on-chip voltage regulation technique for low power digital circuits. In Proceedings of International Symposium on Low Power Electronics and Design, 1999, pp. 109-111.

[41] T. D. Burd, T. A. Pering, A. J. Stratakos, and R. W. Brodersen. A Dynamic Voltage Scaled Microprocessor System. IEEE Journal of Solid-State Circuits, Vol. 35, No. 11, November 2000.

[42] N. S. Kim, T. M. Austin, D. Blaauw, T. N. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. T. Kandemir, and N. Vijaykrishnan. Leakage current: Moore's law meets static power. IEEE Computer, vol. 36, no. 12, pp. 68-75, 2003.

[43] E. Rohou and M. D. Smith. Dynamically managing processor temperature and power. In Proceedings of 2nd Workshop on Feedback-Directed Optimization, 1999.

[44] S. M. Martin, K. Flautner, T. Mudge, and D. Blaauw. Combined dynamic voltage scaling and adaptive body biasing for lower power microprocessors under dynamic workloads. In Proceedings of the 2002 IEEE/ACM International Conference on Computer-Aided Design, 2002, pp. 721-725.

[45] T. Sakurai, H. Kawaguchi, and T. Kuroda. Low-power CMOS design through V_{th} control and low-swing circuits. In Proceedings of the 1997 International Symposium on Low Power Electronics and Design, 1997, pp. 1-6.

[46] H. Zhang, V. George, and J. Rabaey. Low-swing on-chip signaling techniques:

Eectiveness and robustness. IEEE Transactions on VLSI Systems, 2000, vol. 8, pp. 264-272.

[47] F. Worm, P. Ienne, P. Thiran, and G. D. Micheli. An adaptive low-power transmission scheme for on-chip networks. In Proceedings of the 15th International Symposium on System Synthesis, 2002, pp. 92-100.

[48] W. Jeong, B. C. Paul, and K. Roy. Adaptive supply voltage technique for low swing interconnects. In Proceedings of the 2004 Conference on Asia South Pacific Design Automation, 2004, pp. 284-287.

[49] M. Ferretti and P. A. Beerel. Low swing signaling using a dynamic diode-connected driver. In Proceedings of the 27th European Solid-State Circuits Conference, 2001. pp. 369-372.

[50] A. Narasimhan, M. Kasotiya and R. Sridhar. A Low-Swing Differential Signaling Scheme for On-Chip Global Interconnects. In Proceedings of the 18th International Conference on VLSI Design, 2005, pp.634-639.

[51] Intel ® Pentium ® M Processor on 90 nm Process with 2-MB L2 Cache Datasheet, June 2004.

[52] T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In Proceedings of International Symposium on Low Power Electronics and Design, 1998, pp.197-202.

[53] I. Hong, D. Kirovski, G. Qu, M. Potkonjak, and M. Srivastava. Power optimization of variable-voltage core-based systems. IEEE Transaction on Computer Aided Design of Integrated Circuits and Systems, 18(12), December

1999.

[54] K. Flautner, S. K. Reinhardt, and T. N. Mudge. Automatic performance setting for dynamic voltage scaling. In Proceedings of International Conference on Mobile Computing and Networking, 2001, pp.260-271.

[55] A. Azevedo, I. Issenin, R. Cornea, R. Gupta, N. Dutt, A. Veidenbaum, and A. Nicolau. Profile-based Dynamic Voltage Scheduling using Program Checkpoints. In Proceedings of the Conference on Design, Automation and Test in Europe, 2002.

[56] K. Choi, R. Soma and M. Pedram. Fine-Grained Dynamic Voltage and Frequency Scaling for Precise Energy and Performance Trade-off Based on the Ratio of Off-chip Access to On-chip Computation Times, In Proceedings of the conference on Design, automation and test in Europe, 2004.

[57] T. Pering, T. Burd, and R. Brodersen. Voltage scheduling in the lpARM microprocessor system. In Proceedings of International Symposium on Low Power Electronics and Design, 2000.

[58] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In Proceedings of International Symposium on Operating Systems Principles, 2001.

[59] J. Lorch and A. J. Smith. Operating system modifications for task-based speed and voltage scheduling. In Proceedings of International Conference on Mobile Systems, Applications, and Services, 2003.

[60] S. Haga, N. Reeves, R. Barua, and D. Marculescu. Dynamic Functional Unit

Assignment for Low Power. In Proceedings of the Conference on Design, Automation and Test in Europe, 2003.

[61] Q. K. Zhu and M Zhang. Low-voltage swing clock distribution schemes. In proceedings of IEEE International Symposium on Circuits and Systems, pp.418-421, 2001.

[62] B. Gunning, L. Yuan, T. Nguyen, and T. Wong. A CMOS Low-Voltage-Swing Transmission-Line Transceiver. In Proceedings of IEEE International Solid-state Circuits, pp. 58-61, 1992.

[63] W. Kim, J. Kim and S. L. Min. A Dynamic Voltage Scaling Algorithm for Dynamic-Priority Hard Real-Time Systems Using Slack Time Analysis. In Proceedings of the conference on Design, automation and test in Europe, 2002.

[64] A. Sinha and A. P. Chandrakasan. Dynamic voltage scheduling using adaptive filtering of workload traces. In Proceedings of International Conference on VLSI Design, pp. 221-226, 2001.

[65] Transmeta Corporation. TM5400 processor specifications, Jan., 2000.

[66] Intel. Intel StrongARM 1100 microprocessor developers' manual, Aug., 1999.

[67] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In Proceedings of the First Symposium on Operating Systems Design and Implementation, pp.13-23, 1994.

[68] K. Govil, E. Chan, and H. Wasserman. Comparing algorithms for dynamic speed-setting of a low-power CPU. In Proceedings of the First ACM International Conference on Mobile Computing and Networking, pp.13-25, 1995.

[69] J. Lorch and A. Smith. Improving dynamic voltage algorithms with PACE. In Proceedings of the International Conference on Measurement and Modeling of Computer Systems, 2001.

[70] J. Luo and N. K. Jha. Power-profile driven variable voltage scaling for heterogeneous distributed real-time embedded systems. In Proceedings of the 16th International Conference on VLSI Design, 2003.

[71] D. Grunwald, P. Levis, K. I. Farkas, C. B. Morrey III, and M. Neufeld. Policies for dynamic clock scheduling. In Proceedings of the 4th Symposium on Operating Systems Design and Implementation, 2000.

[72] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod. Using SimOS machine simulator to study complex computer systems. ACM Transaction of Modeling & Computer Simulation, 7(1):78-103, 1997.

[73] J. Pouwelse, K. Langendoen, H. Sips. Dynamic voltage scaling on a low-power microprocessor. In proceedings of the 7th annual international conference on Mobile computing and networking, pp.251-259, 2001.

[74] C.H. Hsu, U. Kremer, and M. Hsiao. Compiler-directed dynamic frequency/voltage scheduling for energy reduction in microprocessors. In proceedings of International Symposium on Low Power Electronics and Design, 2001.

[75] M. Huang, J. Renau, and J. Torrellas. Profile-based energy reduction for high-performance processors. In 4th Workshop on Feedback-Directed and Dynamic Optimization, 2001.

[76] D. Mosse, H. Aydin, B. Childers, and R. Melhem. Compiler-assisted dynamic power-aware scheduling for real-time applications. In Workshop on Compiler and Operating Systems for Low Power, 2000.

[77] J. L. Henning. SPEC CPU2000: Measuring CPU Performance in the New Millennium. Computer, pp.28-35, 2000.

[78] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communications Systems. IEEE Micro, pp.330-335, 1997.

[79] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In Proceedings of the 27th International Symposium on Computer Architecture, pp.83-94, 2000.

[80] D. Burger and T. Austin. The Simplescalar Toolset, Version 3.0. Technical Report TR-97-1342, University of Wisconsin- Madison, June 1997.

[81] F. Xie, M. Martonosi and S. Malik. Compile-time dynamic voltage scaling settings: opportunities and limits. In the Proceedings of Programming Language Design and Implementation, June 2003.

[82] S. Ghiasi and D. Gurnwald. A comparison of two architectural power models. In Proceedings of the Workshop on Power-Aware Computing Systems, 2000.

[83] N. S. Kim, T. Austin, T. Mudge, and D. Grunwald. Challenges for architectural level power modeling. Power Aware Computing, 2001.

[84] P. Shivakumar and N. P. Jouppi. Cacti 3.0: An integrated cache timing, power, and area model. Technical Report 2001/2, Western Research Laboratory, 2001.

[85] C.H. Hsu and U. Kremer. Compiler-Directed Dynamic Voltage Scaling Based on Program Regions. Technical Report DCS-TR-461. Rutgers University, Nov 2001.

[86] D. Marculescu. On the use of microarchitecture-driven dynamic voltage scaling. In Workshop on Complexity-Effective Design, 2000.

[87] D.W. Wall. Limits of Instruction-Level Parallelism. In Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 176-188, 1991.

[88] D. Albonesi. Dynamic IPC/Clock Rate Optimization. In Proceedings of the 25th International Symposium on Computer Architecture, pp. 282-292, June 1998.

[89] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In Proceedings of 10th International Conference on Architectural Support for Programming Languages and Operating Systems, 2002.

[90] B. Calder, T. Sherwood, E. Perelman, and G. Hamerly. SimPoint web page. http://www.cse.ucsd.edu/~calder/simpoint/.

[91] C. Isci and M. Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, page 93, 2003.

[92] C. Isci and M. Martonosi. Phase characterization for power: Evaluating control-flow-based and event-counter based techniques. In Proceedings of 12th International Symposium on High Performance Computer Architecture, February 2006.

[93] C. Hu, D. A. Jimaenez, and U. Kremer. Toward an evaluation infrastructure for power and energy optimizations. In Proceedings of 19th International Parallel and Distributed Processing Symposium, April 2005.

[94] C. Hu, D. A. Jimaenez, and U. Kremer. Power Phase Identification for Efficient Simulation. DCS-TR-575, Rutgers University. April 2005.

[95] B. Fields, S. Rubin, and R. Bodik. Focusing Processor Policies via Critical-Path Prediction. In Proceedings of 28th Annual International Symposium on Computer Architecture, pp. 74-85, June 2001.

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

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

[98] C. Isci and M. Martonosi. Identifying Program Power Phase Behavior Using Power Vectors, In Proceedings of IEEE International Workshop on Workload Characterization, Oct., 2003.

[99] P.M. Petersen and D.A. Padua. Static and Dynamic Evaluation of Data Dependence Analysis Techniques, IEEE transactions on parallel and distributed systems, vol. 7, no. 11, November 1996.

[100] W. Amme, P. Braun, and E. Zehendner. Data Dependence Analysis of Assembly Code, In Proceedings of Parallel Architectures and Compilation

Techniques, pp. 12-18, October 1998.

[101] L. Chen, S. Dropshoy, and D.H. Albonesi. Dynamic Data Dependence Tracking and its Application to Branch Prediction, In Proceedings of the Ninth International Symposium on High-Performance Computer Architecture, 2003.

[102] Xia Xiao Xin and Tay Teng Tiow. IPC-Driven Energy Reduction for Low-Power Design, In Proceedings of IEEE international Symposium on Circuits and Systems, 2006.

[103] Xia Xiao Xin and Tay Teng Tiow. Trace-based Energy Reduction for Low-Power Microprocessor Design, In Proceedings of Very Large Scale Integration System-on-Chip (VLSI-SoC), 2005.

[104] Xia Xiao Xin and Tay Teng Tiow. Dynamic Data Dependence Identification Using Phases Analysis, In Proceedings of the Workshop of International Conference on Computational Intelligence, 2005.

[105] E. S. Seo, S. Y. Park, J. S. Kim and J. W. Lee. TSB: A DVS algorithm with quick response for general purpose operating systems Source, Journal of Systems Architecture, Volume 54, Issue 1-2, January 2008.

[106] J. Zhuo and C. Chakrabarti. System-level energy-efficient dynamic task scheduling. In Proceedings of the IEEE Design Automation Conference, 2005.

[107] P. Malani, P. Mukre, and Q. R Qiu. Profile-Based Low Power Scheduling for Conditional Task Graph: A Communication Aware Approach, In Proceedings of IEEE International Symposium on Circuits and Systems, 2007.