View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by ScholarBank@NUS

FUNCTIONAL UNIT SELECTION IN MICROPROCESSORS FOR LOW POWER

PAN YAN

NATIONAL UNIVERSITY OF SINGAPORE

2006

FUNCTIONAL UNIT SELECTION IN MICROPROCESSORS FOR LOW POWER

PAN YAN

(B.Eng., Shanghai Jiao Tong University)

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

NATIONAL UNIVERSITY OF SINGAPORE

2006

Acknowledgements

I would like to express my deepest gratitude to all those who have directly or indirectly provided advice and assistance during the course of my research work in the National University of Singapore.

Assoc. Prof. Tay Teng Tiow (NUS), who has led 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 research work a memorable part of my life.

Mr. Zhu Xiaoping and Mr. Xia Xiaoxin, for their times in several constructive discussions over technical and academic problems. These discussions often helped to clarify questions that are related to the research interest.

My parents, for their invaluable love.

Table of Contents

Acknowledgements i
Table of Contents ii
Abstract iv
List of Tables v
List of Figures vi
Chapter 1 Introduction 1
1.1 Background 1
1.2 Motivation and Contributions of this Thesis 2
1.3 Organization of the thesis 4
Chapter 2 Power Dissipation Sources and Prevention Techniques 5
2.1 Power Dissipation Sources 5
2.1.1 Static Power Dissipation 5
2.1.2 Dynamic Power Dissipation 10
2.2 Power Reduction Techniques 12
2.2.1 Static Power Dissipation Reduction 12
2.2.2 Dynamic Power Dissipation Reduction 19
2.3 Chapter Conclusion 23
Chapter 3 Hardware Basis for Functional Unit Selection 24
3.1 Processor Model 24
3.2 Power and Speed Trade-off for Functional Units 26
3.2.1 Circuit-level Tradeoff 26
3.2.2 An alternative: Voltage Scaling Driven Trade-off 28
3.3 Chapter Conclusion 29
Chapter 4 Technique for In-order Issue Processors 30
4.1 Overview 30
4.2 Static Instruction Filtering Algorithm 32
4.2.1 Basic Block Division 32
4.2.2 Instruction Filtering. 33
4.2.3 Simulation Results 37
4.3 A step forward: Static Instruction Scheduling 42
4.4 Chapter Conclusion 42
Chapter 5 Technique for Out-of-order Issue Processors 43
5.1 Overview 43
5.2 Implementation 43
5.2.1 Recording PI values by Pipeline Profiling 45
5.2.2 Statistical Analyzer 47
5.3 Pros and Cons of profiling based instruction filtering algorithm. 50
5.4 Simulation Results 51
5.4.1 System Configuration 51
5.4.2 General Performance 52
5.4.3 Impact of Threshold Ratio 58
5.4.4 Impact of the Number of Power-frugal FU 63

5.5 Chapter Conclusion 67
Chapter 6 Optimization: Static Instruction Scheduling 68
6.1 Scheduling Objective 69
6.2 Scheduling Algorithm 71
6.2.1 Inter-dependence Table Generation 71
6.2.2 Equivalence Check 73
6.2.3 Scheduling Algorithm 74
6.3 Discussions 79
6.3.1 Issue Scheme: In-order or Out-of-order? 79
6.3.2 FU Selection 80
6.4 Simulation Results 81
6.4.1 In-order issue processors 81
6.4.2 Out-of-order issue processors 85
6.5 Chapter Conclusion 87
Chapter 7 Conclusion 88
Bibliography 90

Abstract

With each new technology generation, transistor density doubled and the correspondingly increased transistor switching frequency dramatically increase on-chip power dissipation. To address this, we propose here in this thesis a low power design technique for microprocessors where multiple Functional Units (FU) of a same function but with different power and performance metrics are employed. Hence, by carefully assigning instructions to either fast or slow FU, power dissipation can be minimized while still providing high performance.

In this work, we focused on the algorithm of FU selection. For in-order and out-of-order issue processors, we developed two instruction filtering algorithms to make the FU choice without modifying the sequence of the object codes. Thus, programs can be optimized as given, and power dissipation is reduced when such codes are running on processors which include power-frugal FU.

To further reduce power dissipation, we also proposed a scheduling algorithm to re-order the instruction order so as to expose more instructions for power-frugal execution. The scheduling program aims at both efficient execution (first objective) and more power reduction. Simulation shows that the scheduling algorithm can improve the execution efficiency, as measured by Instruction Per Cycle (IPC), while still reduces significant amount of energy. Prospect of issuing 30% to 40% of integer ALU instructions to power-frugal ALUs has been shown with the benchmarks. This implies a power reduction of 15% to 20% of power reduction in the integer ALUs.

List of Tables

TABLE I Normalized Power and Delay of 32-bit Adders..27
TABLE II Per-execution Energy and Data Arrivals for Functional Units................................27
TABLE III Data Structures Used in In-order Scheduling..35
TABLE IV Processor Configuration Used in In-order Scheduling 39
TABLE V Code Analysis Results for In-order Processors 39
TABLE VI Data Structures for Profiling Out-of-order Processors..46
TABLE VII Out-of-order Processor Configuration...52
TABLE VIII Out-of-order Instruction Filtering Statistics 53
TABLE IX Execution Simulation Metrics for Modified Codes 54
TABLE X Impact of Threshold Ratio 59
TABLE XI Impact of the Number of Power Frugal ALUs...64
TABLE XII Interdependence Relationships...72
TABLE XIII Statistics Of Scheduled Codes...83
TABLE XIV Impact of the Number of Power Frugal ALUs 86

List of Figures

Fig. 1 ITRS projections for device power consumption [10] 6
Fig. 2 Leakage current mechanisms of deep-submicron transistors [11] 6
Fig. 3 Maximum Clock Frequency Vs. Supply Voltage [16] 11
Fig. 4 Static Power Reduction Techniques 13
Fig. 5 Static Power Reduction Techniques Scaling of Device [17] 14
Fig. 6 Retrograde Doping and Halo Doping [18] 14
Fig. 7 Transistor Stack 15
Fig. 8 Current Mode Signaling and Voltage Mode Signaling [32] 21
Fig. 9 Dynamic Functional Unit Assignment [9] 22
Fig. 10 Processor Pipeline Structure and Resources 24
Fig. 11. Functional Unit with Scaled Supply Voltage 28
Fig. 12. Sample PISA[34] Code & Visualization 31
Fig. 13 Algorithm for Performance Index Estimation 36
Fig. 14 Runtime Power-frugal ALU Issue Percentage (RPAIP) 40
Fig. 15 IPC of Original and Modified Programs 41
Fig. 16. Profiling Based Instruction Filtering System Structure 44
Fig. 17. Statistical Analyzer Screen Shot 49
Fig. 18. Runtime Power-frugal ALU Issue Percentage 54
Fig. 19. Execution Performance Comparison (IPC) 56
Fig. 20. Execution Performance Comparison (IPC) 57
Fig. 21. SIFP for GO.SS with varied Threshold Ratio 60
Fig. 22. SIFP for BZIP00.SS with varied Threshold Ratio 60
Fig. 23. RPAIP for modified GO.SS with varied Threshold Ratio 61
Fig. 24. IPC for modified GO.SS with varied Threshold Ratio 61
Fig. 25. RPAIP for modified BZIP00.SS with varied Threshold Ratio 62
Fig. 26. IPC for modified BZIP00.SS with varied Threshold Ratio 62
Fig. 27. RPAIP for modified GO.SS with varied Threshold Ratio 64
Fig. 28. IPC for modified GO.SS with varied Threshold Ratio 65
Fig. 29. RPAIP for modified BZIP00.SS with varied Threshold Ratio 65
Fig. 30. IPC for modified BZIP00.SS with varied Threshold Ratio 66
Fig. 31. Example: Original Code Sequence 68
Fig. 32. Example: Re-ordered Code Sequence 69
Fig. 33 Algorithm for IDT Generation 73
Fig. 34 Example for Ready and Quasi-Ready Instructions 76
Fig. 35 Processing Steps for Basic Block Scheduling 77
Fig. 36 Sample Solution Tree Aligned to Cycle Numbers 79
Fig. 37 Simulation Scheme for In-order Issue Processors 81
Fig. 38 SIFP Improvement of Scheduled code (compared with Filtered code) 83
Fig. 39 RPAIP Improvement of Scheduled code (compared with Filtered code) 84
Fig. 40 IPC of Scheduled code (compared with Filtered code) 84
Fig. 41 Simulation Scheme for Out-of-order Issue Processors 85

Chapter 1 Introduction

1.1 Background

Each generation of integrated circuit fabrication technology pushes the limit on the number of transistors that can be packed onto a single chip. This allows complex logic and massive memory to be integrated into a single chip in modern-day processors. Performance of microprocessors is thus improved to make various fancy applications possible.

However, this booming of on-chip function is accompanied with significant increase in power consumption by the chips. This causes problems in at least two aspects. Firstly, a large portion of microprocessor centered systems are battery driven, such as found in popular consumer electronics like mobile phones, PDAs and digital cameras. In contrast with the rapid progress of the microprocessor performance, the battery industry is slow in developing powerful batteries to match the need by these applications. Thus, the term “battery-life” is becoming a deciding factor for the overall performance of a product. Secondly, the high power consumption in the compact Integrated Circuit (IC) chips requires advanced packaging and cooling techniques to ensure proper operation. This may result in higher cost and limit some applications.

On a per-transistor basis, power consumption has been decreasing with the advancing of technology, which is mostly due to the lowered power supply voltage for shorter-channel devices. However, with the capacitance per unit area increasing, coupled with raised switching frequency, the overall power density keeps surging

[1][2][3]. At the same time, the ever more complex on-chip function also pushes up chip die sizes, which results in higher overall dynamic power consumption. What is more, as the threshold voltages of transistors are lowered for faster switching, off-state leakage current emerges to be a considerable power dissipation source. Obviously, low power techniques are thus necessary so as to make computer systems, especially portable ones, meet the commercial needs.

Low power techniques targeting at various levels of microprocessor systems have been proposed, ranging from device-level fabrication techniques to system-level scheduling techniques. We will review some of these low power techniques in Chapter 2.

1.2 Motivation and Contributions of this Thesis

Though we prefer techniques that provide high performance and low power at the same time, it is a matter of fact that usually higher performance comes at the price of higher power. Thus, one important branch of low power technique is based on the trade-off between performance and power. The basic idea behind is that maximum performance is not always necessary for many applications, especially applications that center on a user, and by cleverly lowering the performance where appropriate, the power consumption is reduced while the overall performance is still acceptable to the user. The power saving may be categorized into two parts: 1) Incorporating low-power working modes, which are usually associated with lower performance; 2) Making a decision on when to switch to low-power modes.

Several published and commercial low power techniques falls into this category.

Intel SpeedStep uses DVS to provide the multiple working modes and switches the modes based on IPC [4]. The Data Retention Gated-GND cache uses transistor stacks to provide the standby modes, which means less leakage, and switches whenever there is no access [5]. Offline code analysis [6] or real time scheduling [7] can both be used to direct DVS.

Obviously the efficiency of such mode-switching low power techniques depends on two things: 1) the amount of power that is to be saved in the low power mode compared to that in active mode. 2) The percentage of time we can switch the processor to low-power mode.

The method being presented here focuses on the Functional Units (FU) in microprocessors. None of the available low power techniques has taken into account the facts that: 1) the design of FUs is always aiming at providing the best performance; 2) the results of arithmetic and logic instructions are not always immediately needed upon their completion; 3) slower FUs, typically with a simpler circuit structure, consume significantly less energy than their faster counterparts [8]. Based on these facts, we present a novel power saving technique. Extra slow FUs with lower per-execution energy are introduced into a processor. Using code analysis and/or run-time pipeline profiling, certain instructions are then picked out to be issued to these power-frugal FUs. An instruction re-scheduling algorithm is developed which re-orders instructions to increase the number of instructions that may be issued to slower FUs without significant compromises on performance. With this method, simulations show that around 40% of all FUs instructions can be directed into slower

FU while incurring less than 0.4% performance degradation, as measured by IPC.

This technique provides a fine-grain mechanism for lowering performance at an instruction-by-instruction level, which is not possible in DVS or any other technique. It allows instructions of different urgency to be executed at different power cost. This technique can be implemented together with other power-saving techniques like DVS [6][7] and FU assignment [9]. The power saving achieved here is an extra gain. What is more, the overall performance is not noticeably degraded as a result of the algorithm that drives the instruction selection process. The advantage of this method also lies in its wide range of application and simplicity for practical implementation.

1.3 Organization of the thesis

The remainder of this thesis is organized as follows. Chapter 2 reviews the basic issues of processor power dissipation. Various types of power dissipation sources are identified. Available low power techniques are briefly reviewed. Chapter 3 presents a novel hardware basis for the FU selection scheme. The trade-off between power and performance in various FU are studied. The processor architecture to implement our scheme is also described. Chapter 4 focuses on in-order issue processors. Techniques specifically developed for these processors are proposed. Chapter 5 follows with techniques for out-of-order processors. Chapter 6 proposes a basic-block based instruction scheduling algorithm, which optimizes object codes for both in-order and out-of-order processors so as to improve the power reduction achievable with the proposed techniques. Chapter 7 draws the conclusions and projects future work.

Chapter 2 Power Dissipation Sources and Prevention Techniques

For CMOS circuits, leakage current in digital circuits has long been negligible in digital circuits. Thus, the switching-induced dynamic power dissipation has long been the sole target of low power processor design techniques. However, with finer feature sizes, leakage-induced static power dissipation emerges and is predicted to play a major role in future processors. In this chapter, we identify the power dissipation sources in both categories. Then, low power techniques at different levels to address both types of power dissipation are reviewed.

2.1 Power Dissipation Sources

Generally we can divide power dissipation into to two categories: 1) Static power dissipation, which is switching independent and mostly induced by various leakage currents; 2) Dynamic power dissipation, which arises from the switching activities of logic circuits. We examine both of them in detail here.

2.1.1 Static Power Dissipation

In deep sub-micrometer regimes, the high leakage current is becoming a significant contributor to the overall power dissipation of CMOS circuits, as threshold voltage, channel length and gate oxide thickness are reduced. Fig. 1 shows the projections done by the International Technology Roadmap for Semiconductors (ITRS) for the relative significance of static and dynamic power consumptions with respect to technology progress. It can be seen that the static power dissipation is expected to

overwhelm dynamic power dissipation unless effective static power reduction techniques are properly applied.

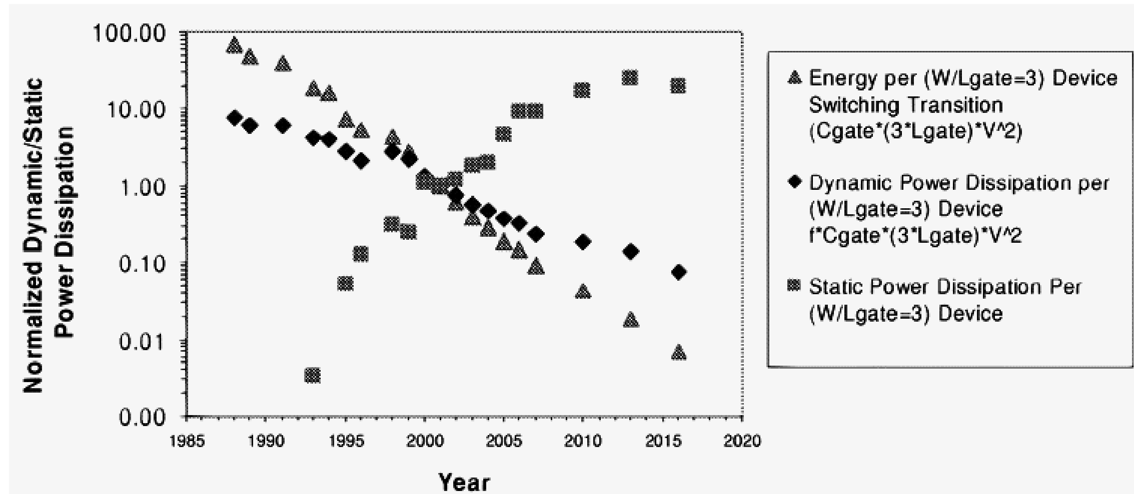


Fig. 1 ITRS projections for device power consumption [10]

For deep-submicron transistors, there are six major leakage mechanisms that contribute to the static power dissipation, as illustrated in Fig. 2 below.

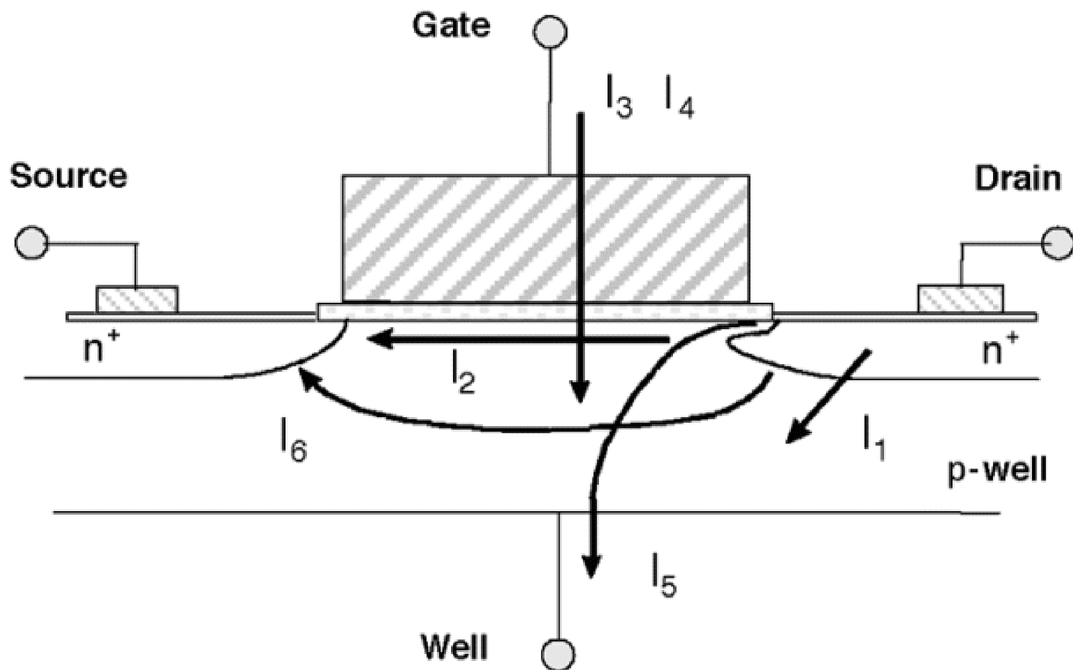


Fig. 2 Leakage current mechanisms of deep-submicron transistors [11]

In Fig. 2, the six leakage mechanisms are [11]:

1. PN Junction Revers-Bias Current (I_1)

2. Sub-threshold Leakage (I_2)

3. Tunneling into and through Gate Oxide (I_3)

4. Injection of Hot Carriers from Substrate to Gate Oxide (I_4)

5. Gate-Induced Drain Leakage (I_5)

6. Punch-through (I_6)

Currently, for a well-fabricated transistor, the major part of leakage comes from the first two leakage mechanisms: 1) PN Junction Reverse-bias Leakage (I_1); 2) Sub-threshold Leakage (I_2).

2.1.1.1 PN-Junction Reverse-Bias Current (I_1)

This leakage mechanism is incurred as drain and source to well junctions are typically reverse-biased. This leakage 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 [12]. PN-Junction reverse-bias leakage is a complex function of junction area and doping concentration [12]. If both p and n regions are heavily doped, band-to-band tunneling (BTBT) dominates the leakage current. The current density can hence be approximated by [13]:

$$J = A\frac{EV_{app}}{E_g^{1/2}}\exp\left(-B\frac{E_g^{3/2}}{E}\right) \quad (1)$$

$$A = \frac{\sqrt{2m^*}q^3}{4\pi^3\hbar^2}\text{, and } B = \frac{4\sqrt{2m^*}}{3q\hbar} \quad (2)$$

Where m^* is effective mass of electron; E_g is the energy-band gap; V_{app} is the

applied reverse bias; E is the electric field at the junction; q is the electronic charge; and $\hbar$ is $1/2\pi$ times Planck's constant. Assuming a step junction, the electric field at the junction is given by [13]

$$E = \sqrt{\frac{2qN_a N_d (V_{app} + V_{bi})}{\varepsilon_{si}(N_a + N_d)}} \tag{3}$$

where N_a and N_d are the doping in the p and n side, respectively; ε_{si} is permittivity of silicon; and V_{bi} is the built-in voltage across the junction. In scaled devices, the higher doping concentrations and abrupt doping profiles cause significant BTBT current through the drain-well junction.

2.1.1.2 Sub-threshold Leakage

The sub-threshold leakage is the leakage between source and drain in an off-state transistor. In modern MOSFETs, weak inversion leakage is the dominate part in sub-threshold leakage. Consider an NMOS where Vd > Vs, Vs=0 and Vg < Vth, the V_{DS} drops almost entirely across the reverse-biased substrate-drain pn junction. Here conduction is dominated by the diffusion current and is similar to charge transport across the base of bipolar transistors. 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 [11]. The threshold leakage including weak inversion, DIBL and Body Effect can be modeled as [14]

$$I_{sub_{th}} = A \times e^{1/mv_T (V_G - V_S - V_{th0} - \gamma' \times V_s + \eta V_{DS})} \times (1 - e^{-V_{DS}/v_T}) \tag{4}$$

where,

$$A = \mu_0 C_{OX}^{'} \frac{W}{L_{eff}} (v_T)^2 e^{1.8} e^{-\Delta V_{th}/\eta v_T} \quad (5)$$

V_{th0} is the zero bias threshold voltage, and $v_T = KT/q$ is the thermal voltage. The body effect for small values of source to bulk voltages is linear and is represented by the term $\gamma' Vs$, where γ' is the linearized body effect coefficient. η is the DIBL coefficient, C_{ox} is the gate oxide capacitance, μ_0 is the zero bias mobility, and *m* is the sub-threshold swing coefficient of the transistor. ΔV_{TH} is a term introduced to account for transistor-to-transistor leakage variations.

From the equation (4), it is important to note that the sub-threshold leakage increases exponentially with smaller threshold voltage and larger drain-source voltage. As feature size decreases with each generation of technology, the supply voltage is scaled down and the threshold voltage must be scaled down proportionally to maintain performance. Thus, smaller threshold induces exponentially increasing sub-threshold leakage. On the other hand, on a certain fabricated chip with a fixed threshold voltage, reducing supply voltage can also significantly reduce sub-threshold leakage. Equation (4) provides the guideline in designing leakage reduction techniques.

It can be seen the static power dissipation is very complex and not easy to model. The static power can be represented by:

$$P_{static} = I_{leak} \times V_{DD} \quad (6)$$

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

2.1.2 Dynamic Power Dissipation

For many years, efforts toward power reduction have been focused on reducing dynamic power dissipation, mainly due to the extensive use of CMOS technology where leakage in the static state is many orders of magnitude smaller compared to power consumed as a result dynamic switching of states.

Dynamic power dissipation mainly arises from two circuit behaviors: 1) transient short-circuit current; and 2) repeated charging and discharging of capacitive loads.

The short-circuit current is incurred due to transient conduction of both the pull-up and pull-down circuits in the CMOS circuit. Because transition cannot realistically be instant, it is possible that the shut-off network is turned on before the previously turned-on network is shut off. This current, however, is not significant in most circuits and is often ignored [3][15].

The major dynamic power consumption 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, the capacitance per unit area increases, accompanied by the increased switching frequency. These trends lead to significant dynamic power consumption in modern-day processors.

In conventional process technology, the dynamic power involved in the

switching is estimated by

$$P_{dynamic} = \alpha \bullet C_L \bullet V_{DD} \bullet \Delta V \bullet f_{CLK} \tag{7}$$

Where α is a circuit-dependent constant, 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 normal switching in a CMOS circuit, 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

$$T = \frac{N}{f_{CLK}} \tag{8}$$

Also, the fastest clock frequency achievable shows a nearly linear dependence upon supply voltage, due to the driving ability of transistors, which is illustrated in Fig. 3 below [16].

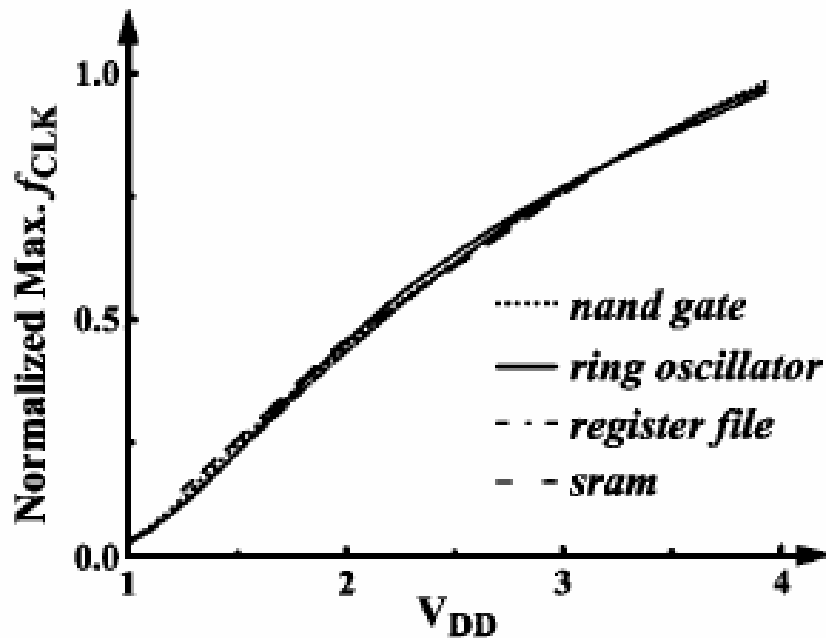


Fig. 3 Maximum Clock Frequency Vs. Supply Voltage [16]

Thus we can approximately put:

$$f_{CLK} = k \bullet V_{DD} \tag{9}$$

Thus, the dynamic power can be estimated by:

$$P_{dynamic} = \left(\alpha \bullet C_L \bullet k\right) \bullet V_{DD}^3 \tag{10}$$

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

2.2 Power Reduction Techniques

In this section, we review various techniques targeting at reducing both static and dynamic power dissipation. These techniques range from device fabrication level to system design level.

2.2.1 Static Power Dissipation Reduction

There are a wide range of low power techniques addressing static power dissipation, from the fabrication level engineering to the system level design. As a quick summary, we list some of them in Fig. 4. Each of these techniques will be examined in the following sub-sections.

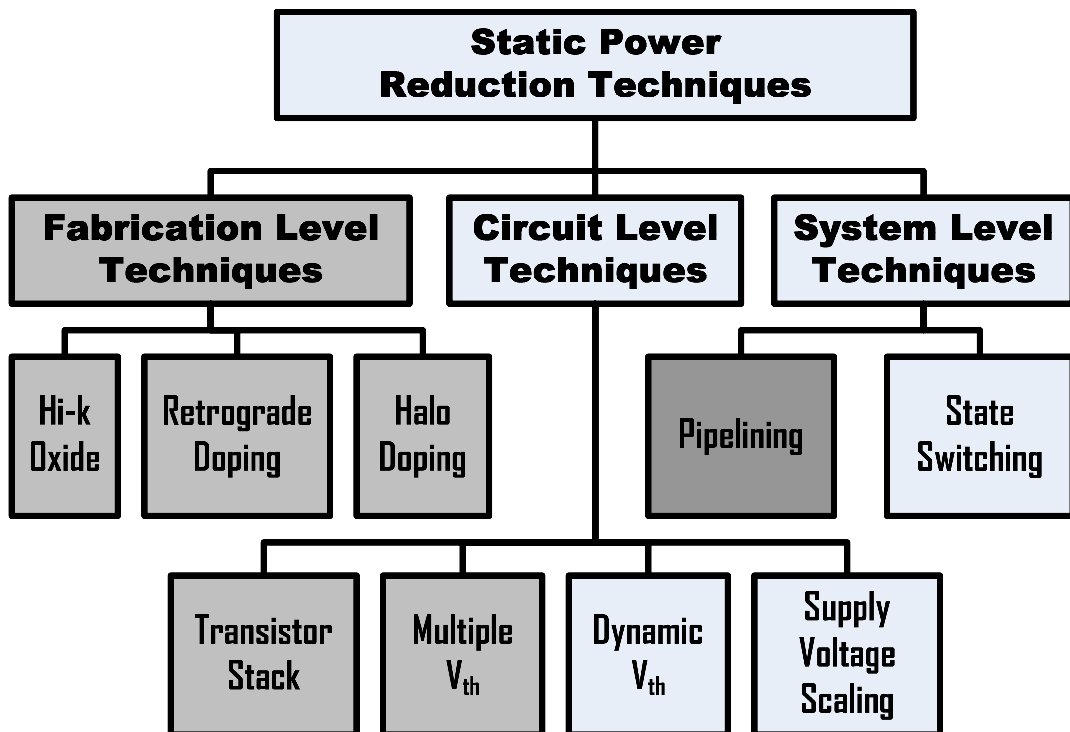


Fig. 4 Static Power Reduction Techniques

2.2.1.1 Fabrication Level Techniques for Static Power Reduction

To minimize the overall static power dissipation, a straight forward way is to minimize the leakage in each transistor. This can be done with fabrication techniques.

First of all, with deep submicron transistors, scaling happens not only in the lateral dimension (channel length), but also in the vertical dimension, doping concentration and supply voltage, so as to maintain performance. This is illustrated in Fig. 5 [17]. Thus, gate oxide thickness is getting thinner, which results in increased leakage through gate node. This can be solved by using High-k insulating materials, which increases physical thickness of the insulator while keeping reduced equivalent electrical thickness.

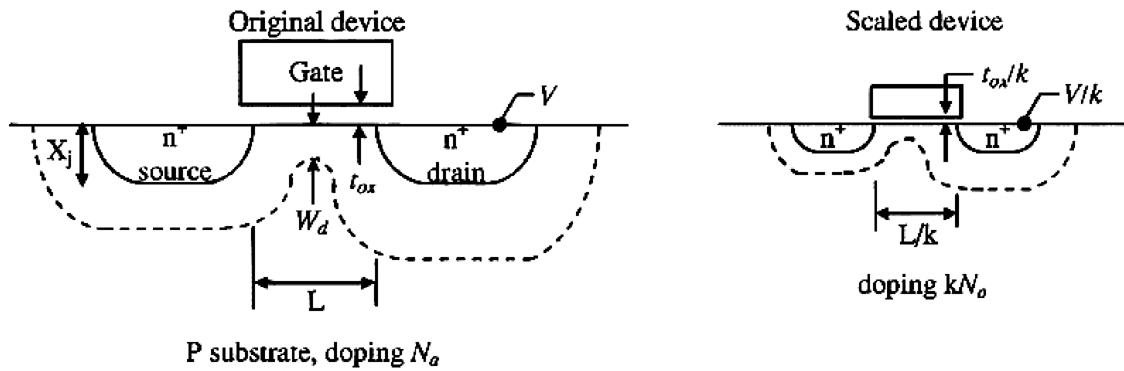


Fig. 5 Static Power Reduction Techniques Scaling of Device [17]

As the channel length is scaled down, punch-through becomes a significant 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". The low surface doping is to make sure less impurity is present in the surface and hence mobility will be higher. The higher sub-surface concentration can counteract the nearing of source and drain regions, which reduces punch-through leakage. The retrograde doping is illustrated in Fig. 6 [18].

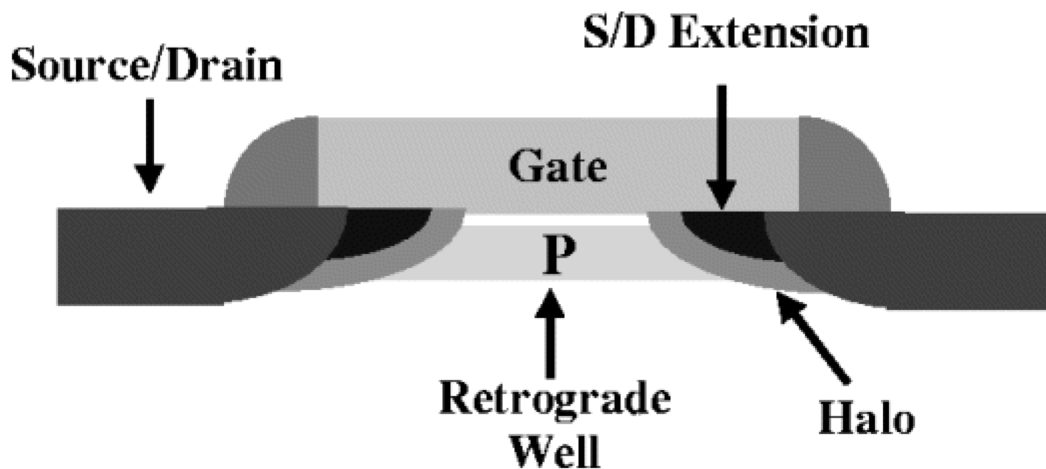


Fig. 6 Retrograde Doping and Halo Doping [18]

Below the edge of the gate, where is also the end of the source or drain region, additional doping of the substrate type is introduced. This will result in narrower depletion region, hence reduces the charge-sharing effect [19] and the threshold voltage degradation, and eventually reduces the sub-threshold leakage. Halo doping is also illustrated in Fig. 6.

These fabrication techniques are already in use to provide transistors with the best performance possible. More detailed discussion of these techniques can be found in [11].

2.2.1.2 Circuit Level Techniques for Static Power Reduction

With the fabrication level techniques applied to extremes, additional leakage power reduction can be achieved by carefully designing the circuit structures. Here we describe four popular circuit level techniques to reduce leakage.

A) Transistor Stack

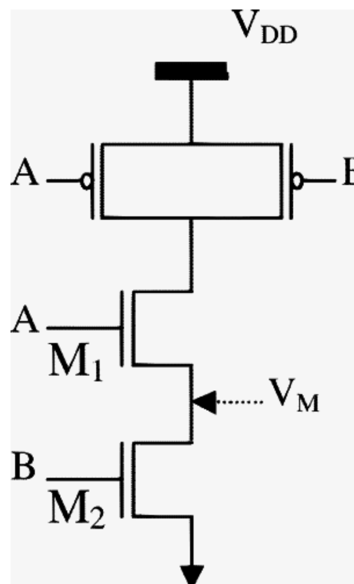


Fig. 7 Transistor Stack

One promising way of reducing standby leakage is by intentionally introducing a series-connected transistor. Sub-threshold leakage current can be reduced when more than one transistor in the stack is turned off. This is known as stacking effect [14]. Consider the NAND circuit in Fig. 7. When M1 and M2 are both turned off, the voltage at the intermediate node (V_M) is positive due to the small drain current that flows through M2. Positive potential at this node has three effects:

1) Due to the positive source potential V_M, gate-to-source voltage of M1 becomes negative; hence, the sub-threshold current reduces substantially.

2) Due to $V_M>0$, body-to-source potential of M1 becomes negative, resulting in an increase in the threshold voltage of M1 (body effect), and thus reducing the sub-threshold leakage.

3) Due to $V_M>0$, the drain to source potential of M1 decreases, resulting in the lessening of Drain Induced Barrier Lowering (DIBL), and reducing the sub-threshold leakage.

Apart from the above explanations, the situation here can be intuitively understood by taking the off-state transistors as non-linear resistors. An additional resistor will reduce leakage. According to [20], the leakage of a two-transistor stack is an order of magnitude less than the leakage in a single transistor. Thus, we have at least two ways to reduce leakage:

1) To carefully choose the input vector so as to allow more off-state transistors in series. This has been proved to be an effective way of controlling the

sub-threshold leakage [21].

2) To employ additional transistors to gate a circuit structure from the power supply, as done with the Gated-VDD circuit technique [22].

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. 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. Obviously, changing the channel doping [23], gate oxide thickness [23], channel length [24] and body bias can all affect the final threshold voltage of a transistor. Thus, we can change the V_{th} either statically or dynamically. Possible solutions include:

1) MT-CMOS. This is similar to transistor stack. Additional high-threshold transistors are put in series to low V_{th} circuity. These additional transistors reduce leakage in sleep mode of a circuit.

2) Dual threshold CMOS. We can fabricate transistors in critical paths with lower threshold to guarantee best performance while apply higher threshold elsewhere.

3) Variable threshold CMOS. By changing the body bias of transistors, the threshold voltage can be manipulated at run time.

C) Supply Voltage Scaling

Designed to reduce dynamic power dissipation, voltage scaling techniques are the most successful and widely used low power techniques. Interestingly, it is also an effective method for leakage reduction, since the sub-threshold leakage can be reduced because DIBL decreases as the supply voltage is scaled down [25]. [26] showed that supply voltage scaling achieved sub-threshold and gate leakage reduction in the orders of V^3 and V^4 respectively.

2.2.1.3 System Level Techniques for Static Power Reduction

Further static power reduction can be achieved by applying higher level low power techniques. The nature of static power dissipation indicates 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. Pipelining, though developed for improving the performance of processors, thus has an side effect of reducing static energy consumption. On the other hand, the operation of certain tasks can be divided into various phases in which the processors can be of different levels of activities. Identifying these phases helps in minimizing the static power dissipated.

A) Pipelining

Pipelining saves energy in a straight-forward way. It significantly reduces the overall execution time of a certain program. Thus, the time of leakage flowing is also reduced. N.S. Kim *et al* [3] compared the overall power consumption of pipelined

systems and series systems, and concluded that "pipelining's combined dynamic and static power leakage will be less than that of the serial case".

<u>B) Phase Switching</u>

Modern day processors are designed for 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. Gated-V_{DD} Caches [22] and DVS systems are examples of this. Then, identifying the phases itself is a system level effort toward low power design.

In summary, there are many trade-offs among cost, system complexity and power saving performance in applying the numerously mentioned static power reduction techniques. Careful design is needed. Even though we do not target on leakage reduction in our research work presented here in this thesis, it is important to know that we have so many available techniques to be combined to further reduce the overall power dissipation of a processor.

2.2.2 Dynamic Power Dissipation Reduction

Here we review the low power techniques that target dynamic power dissipation. These techniques are also grouped into either circuit-level or system-level.

2.2.2.1 Circuit-level Techniques for Dynamic Power Reduction

Dynamic power dissipation can be easily modeled by:

$$P_{dynamic} = \alpha \bullet C_L \bullet V_{DD} \bullet \Delta V \bullet f_{CLK} \quad (11)$$

It is natural to think of reducing the voltage swing and supply voltage to minimize the dynamic power. Low-swing signaling and current mode signaling aim at reducing the voltage swing while Dynamic Voltage Scaling reduces the supply voltage.

A) Low-swing Signaling

The first method is by reducing the signal swing. 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. This technique has been carefully studied in [27][28][29][30]. It is also employed in the arithmetic core of Pentium 4 Processors [31].

B) Current Mode Signaling

Another technique that also provides high speed and low power is current mode signaling. Compared with normal circuits where signal is represented by voltages, current mode circuits employ current to represent signal, especially for long transmission lines. As shown in Fig. 8, instead of driving the transmission line to full rail voltages, current mode circuits drive the transmission line with a current source and this signal is received by a matched low impedance current mode receiver. As the current pulse does not switch the capacitance of the transmission line, power consumption is considerably reduced [32].

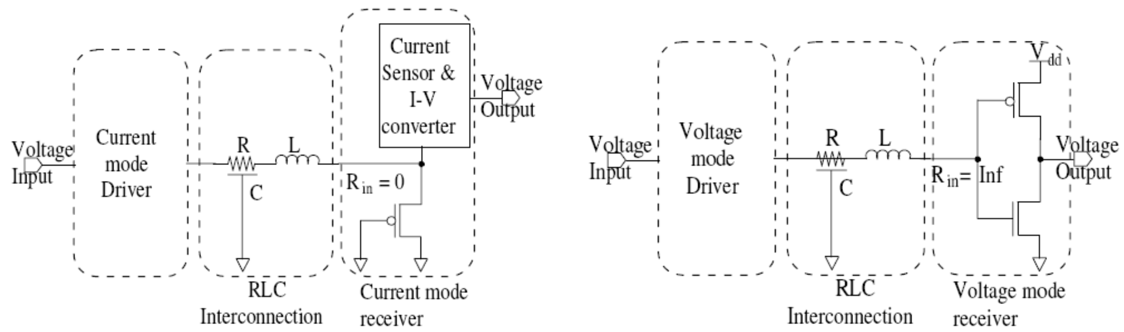


Fig. 8 Current Mode Signaling and Voltage Mode Signaling [32]

C) Dynamic Voltage Scaling

Dynamic Voltage Scaling (DVS) is by far the most popular technique in use. As deducted in section 2.1.2, dynamic power has a cubic relationship with supply voltage in conventional CMOS circuits, while the maximum clock frequency is approximately proportional to supply voltage. Thus, as a first order estimation, given a task that is to be finished in N clock cycles, if we apply a scaled supply voltage V_{DD}'=sV_{DD} (s<1), the total time needed to finish the task will be: T'=T/s and the dynamic power will be P'=s^3P. Summing them up, the total energy spent for the task will be E'=s^2E. That is, if we apply half the supply voltage, the total energy spent will be only one fourth of the original, but the price we pay is that the task will take double the time to finish. DVS has been widely used in commercial chips such as Pentium 4[31]. 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 consumption.

2.2.2.2 System-level Techniques for Dynamic Power Reduction

Higher level techniques are also being developed to achieve dynamic power

reduction. Typically these techniques make use of system level information to reduce either the voltage swing or supply voltage.

A) Dynamic Functional Unit Assignment

S. Haga *et al* [9] proposed to dynamically assign instructions to carefully selected Functional Units to minimize signal switching that happens in the FU. Instructions are preferably issued to FU where the previous operands are more similar to the current operands. This is illustrated in Fig. 9.

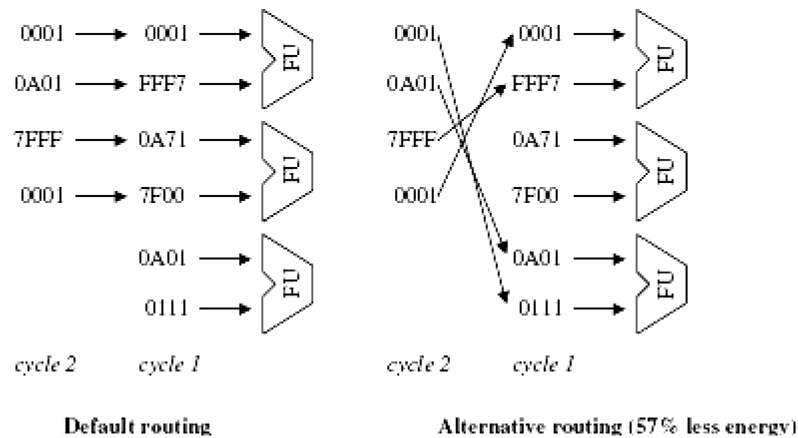


Fig. 9 Dynamic Functional Unit Assignment [9]

Thus, signal switching happening at the input ports, output port and inside of the FU are reduced. This is achieved at the price of extra hardware that carries out the comparing of the operands. Simplified algorithm helped to minimize the hardware cost. Simulation results showed an average of 17% to 26% reduction of switching activities in various FU [9].

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. This state switching decision can be made by either hardware or software. Additional hardware can be added to monitor the IPC and adjust the supply voltage accordingly. Otherwise, in real-time systems, the operation system can scale the supply voltage to make each task finish just within the deadline [7]. These approaches all lead to better power performance in microprocessors.

2.3 Chapter Conclusion

In this chapter, various existing techniques for static and dynamic power reduction have been described. Many of these static and dynamic power reduction techniques can be combined to minimize the overall power consumption. The technique we are to introduce in this thesis is at system-level that utilizes code information to adjust the FU selection statically to save dynamic power dissipation.

Chapter 3 Hardware Basis for Functional Unit Selection

In this chapter, the proposed hardware basis for the low power approach will be described. First, the processor model that we base our research on is presented. After that, based on the observation of a trade-off between performance and power dissipation, we will present the ways the same functional units can be implemented with varied power. As the focus of this thesis is on the software technique that utilizes these FUs with varied performance, detailed circuit designs are not included.

3.1 Processor Model

We target our research on a generic 6 stage pipelined microprocessor structure described in [33]. The structure of the pipeline is illustrated below:

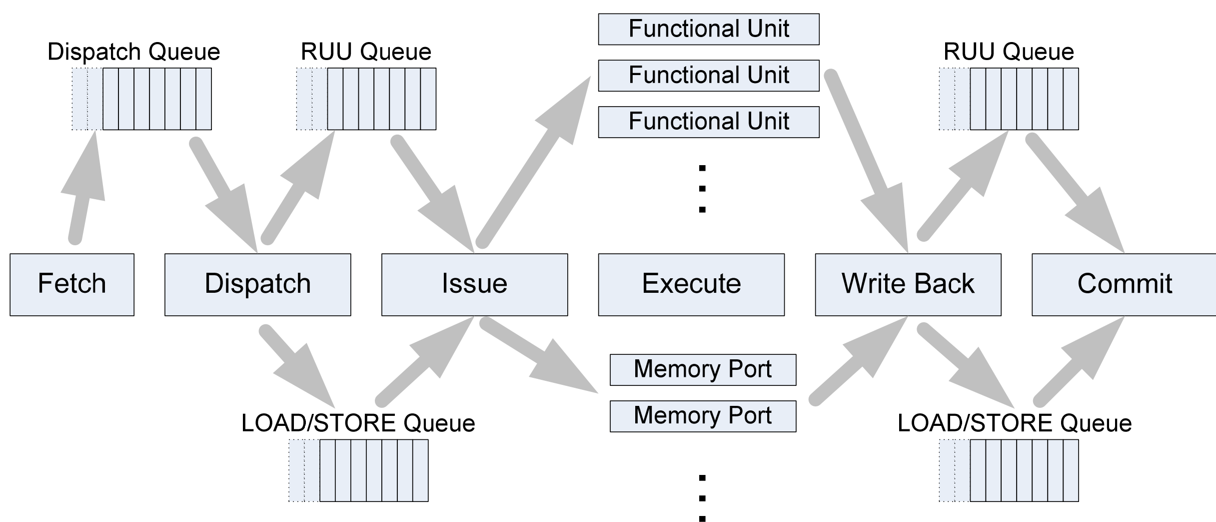


Fig. 10 Processor Pipeline Structure and Resources

In "Fetch" stage, instructions are fetched from the instruction cache to be filled into the "Dispatch Queue". Delay may be incurred if a cache-miss happens. Each cycle,

multiple instructions will be fetched until either: 1) Dispatch Queue is full; 2) fetch width is met; or 3) no instruction is available from the cache.

In “Dispatch” stage, instructions are retrieved from the “Dispatch Queue”, decoded, and assigned to a Register Update Unit (RUU) [33]. The RUU is a structure that serves as a Reserve Station. Instructions, together with the operands and results are temperately stored in this unit so as to resolve dependency and to ensure precise interrupt. RUUs are then en-queued to either an RUU queue to wait for the operands to be ready, or en-queued to the Load/Store queue for load store instructions. For out-of-order issuing, the dispatch operation will continue until either: 1) The RUU Queue or Load/Store Queue is full; 2) dispatch width is met; 3) the Dispatch Queue is empty. For in-order issuing, there is one extra condition that new instructions can be dispatched only when the previous instruction is ready to be issued. This makes sure the in-order nature of the issuing of instructions.

In “Issue” stage, the RUU queue is scanned and ready instructions with operands all already generated are issued to its corresponding Functional Unit, if available. A record of the issued instruction is still kept in the RUU queue to maintain the relative sequence of instructions. Issue width limits the number of instructions that can be issued each cycle. Issuing is also limited by the availability of the requested Functional Unit.

Instructions are actually executed in the Functional Unit. After execution, they enter the “Write Back” stage, where the result of the execution will be written back into the RUU and the dependency of subsequent instructions is resolved.

Finally, each cycle, instructions are committed in sequence in the “Commit” stage to maintain precise interrupt.

The SimpleScalar Simulation Toolset [34] is modified and used for simulating the above processor. It is based on exactly such a processor structure and is supporting a MIPS-like instruction set. Extra slower and power-frugal FU and extra instructions that are associated with these power-frugal FU are supported. The extra FU will have the same interface as their fast counter-parts. Extra instructions are added by some slight modification to the decoding part of the Dispatch Stage.

3.2 Power and Speed Trade-off for Functional Units

The performance of FU in microprocessors is usually pushed to extremes to provide the shortest latency. However, in reality, there are lots of situations where such fast execution is not necessary. In these cases, power can be saved by intentionally executing these carefully selected instructions on slower and more power-frugal FU. On the other hand, when best performance is needed, running instructions at full speed should also be possible. Thus, to achieve lower power dissipation without significantly harming the overall performance of a processor, we need a hardware knob with which we can choose whether to execute an instruction at higher speed and higher power consumption or to execute it at lower speed while saving power. Such a knob can be provided in various ways.

3.2.1 Circuit-level Tradeoff

One of such a design is based on the observation that faster FUs are typically schematically more complex and employ more transistors. Thus, when carrying out an

execution, the faster FU will usually consume more power.

Take an adder for example. According to [35], we list the circuit type, number of transistors employed, normalized mean dynamic power and worst-case delay in the following TABLE I.

TABLE I Normalized Power and Delay of 32-bit Adders

Type	**# of Transistors.**	**Mean Power**	**Worst Delay**
RCA	884	1.000	1.000
BCLA	2228	2.003	0.516
SDA-16	2184	2.047	0.311

The power-frugal Ripple Carry Adder is schematically much simpler than the other two faster ones by employing less than half of the transistors of BCLA and SDA-16. At the same time, the speed of RCA is also much slower than the other two. In this case, the difference in circuit structure incurred the varied speed and power. TABLE II lists power and speed of FUs of a same function but with varied performance as carried out by Mr. Ng Karsin in his masters' thesis [8].

TABLE II Per-execution Energy and Data Arrivals for Functional Units

Functions		Fast	Slow	Difference (percentage)
Addition	Energy (pJ)	56	23	33 (58.9%)
	Data Arrival(ns)	3.77	12.00	8.23
Subtraction	Energy (pJ)	57	24	33 (57.9%)
	Data Arrival(ns)	4.13	12.51	8.38
Multiplication	Energy (pJ)	703	394	309 (44.0%)
	Data Arrival(ns)	8.17	27.11	18.94
Division	Energy (pJ)	1218	1049	169 (13.9%)
	Data Arrival(ns)	30.26	54.68	24.42

We can expand such comparison of power/speed to many other Functional Units. These are simulation results generated in Synopsis under 0.35um technology. The trade-off between power and speed is clearly illustrated.

When building a processor, careful circuit design is needed to provide a spectrum of FU with varied power/speed nodes. The focus of this thesis is rather on the software technique to make use of these various FUs, so the detailed circuit design techniques to make such FU are not discussed.

3.2.2 An alternative: Voltage Scaling Driven Trade-off

Even though we can depend on circuit structures to provide the various FUs of different power and execution speed, it requires extra effort in designing. As mentioned in Chapter 2, supply voltage scaling can lower power dissipation at the price of slower execution speed. Thus, we can apply lowered supply voltage to a duplicated FU so as to make it a slower and power-frugal one.

The structure of a supply-voltage-scaled FU is illustrated in Fig.11.

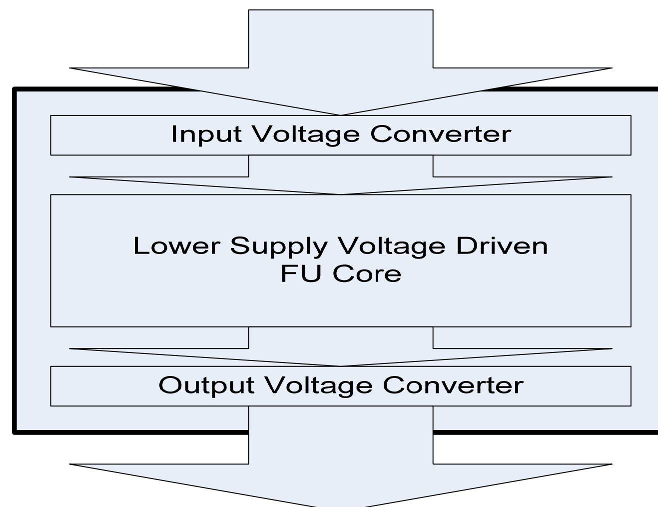


Fig. 11. Functional Unit with Scaled Supply Voltage

Such a method has the advantage of wide applicability. Most CMOS circuits can be readily incorporated in such a scheme and provide varied power and performance. No extra circuit designing needed. However, these voltage-scaling driven power-frugal FU will need voltage converters as their interface with the other parts of the processor. These converters also consume power at execution. As a result, the application of this method is limited to more complex FU only, where the power consumption by the extra voltage converter does not offset the power saved with scaled supply voltage.

3.3 Chapter Conclusion

In this chapter, we have described the architecture of the processor we target our research at and also proposed two ways of providing extra FUs with lower power and execution speed. These power-frugal FUs will be associated with extra instructions and be used when it is feasible as decided by the scheduling algorithm described in Chapter 6.

Chapter 4 Technique for In-order Issue Processors

With the extra power-frugal FU introduced into the processor architecture, the remaining issue is on where and when to use them. As the complexity of the issuing logic of in-order and out-of-order processors differs a lot, we separate the discussion to target at each separately. The approach for in-order issue processors are described here while the next chapter is dedicated to out-of-order issue processors.

4.1 Overview

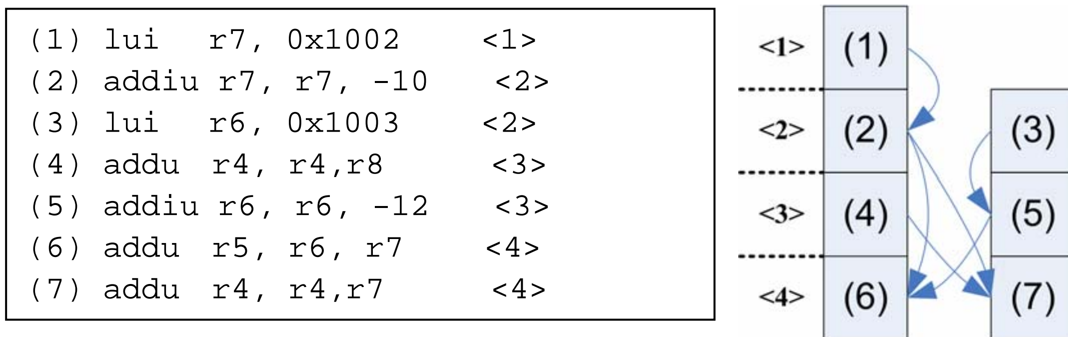
The in-order issue super-scalar processor we target at is the one that employs multiple FUs and issues multiple instructions in the source-code order. As the issue width may be larger than one and the latency of different FUs may vary, several instructions may be issued on-the-fly at any point in time. The benefit of in-order issue lies in that it significantly reduces the complexity of issue logic.

As the issuing of instructions is in-order, the behavior of the processor within a basic block is highly deterministic. Here, a “basic block” refers to a sequence of instructions with a single entry point, single exit point, and no internal branches. Within a basic block, by a single-pass code scan, the relative issue time and completion time of each instruction can be estimated under simplified assumptions. Such information can then be analyzed and used to determine which instruction should be executed at full speed while which could be executed at a relatively slower speed.

As performance is the most important factor for processors, in our approach, we assume the best performance execution. That is, we aim at not lowering the execution

speed of programs while reducing power consumption.

In conventional processor designs, FU are always designed to provide best performance. That is, any adder, multiplier or divider is only aimed at providing fastest execution time. All the instructions are executed at the fastest speed possible in the FU. However, though simulation, it is found that there always exist instructions whose results are not immediately utilized by subsequent instructions. Take the following segments of code in Fig.12 as an example.



```
(1) lui   r7, 0x1002     <1>
(2) addiu r7, r7, -10     <2>
(3) lui   r6, 0x1003     <2>
(4) addu  r4, r4,r8       <3>
(5) addiu r6, r6, -12     <3>
(6) addu  r5, r6, r7     <4>
(7) addu  r4, r4,r7       <4>
```


Fig. 12. Sample PISA[34] Code & Visualization

Let us assume an issue width of 2-instruction per cycle and single-cycle integer adder latency for all these instructions. A visualized illustration is given next to the code box, where instructions are vertically aligned to issue cycles. The arrows illustrate the data dependence within the code. It can be seen that the result of instruction (2) is generated before cycle <3>, but is not needed until cycle <4>. We can actually issue instruction (2) to a power-frugal adder, which takes 2 cycles to finish. Thus, (2) will be executed in parallel with (4) and (5) without blocking the issuing of any instruction. Such substitution does no harm to the overall performance, but saves power by utilizing a structurally simpler adder. Situations like instruction (2) are ubiquitous in every application, as shown by the simulation results in the coming

section. On the contrary, instructions like (1), (3), (4) and (5) should not be issued to power-frugal adders, as their results are immediately referenced by instructions to be issued in the next cycle.

Thus, if we can filter out eligible instructions and issue them to slower and more power-frugal FU, dynamic energy dissipation can clearly be reduced. To achieve this, we take a static method by analyzing the object code of programs to be run on the processor, filter out eligible instructions and then modify their op-code so as to associate them to power-frugal FU. Essentially, the FU choice is made statically and hence no fancy hardware is needed in the processor core. From the processor's view, only some new instructions and their corresponding FU are added.

In this chapter, we first describe a code-scanning algorithm that will filter out instructions whose results are not immediately utilized by later instructions. Power saving estimations based on simulation results is presented.

4.2 Static Instruction Filtering Algorithm

Our algorithm works on the object code of any program compiled for a PISA microprocessor [34]. The structure of the code conforms to the standard MIPS ECOFF structure. According to the header structure information, it is easy to browse to the text segment where the instructions are stored. Several steps are involved in the analysis of the instructions.

4.2.1 Basic Block Division

A first step is to divide the whole text segment into basic blocks for further

analysis. Based on the definition of a basic block, the following rules are used to derive basic blocks.

A) Any branch instruction marks the end of a basic block.

B) The instruction before the destination of any branching instruction marks the start of a basic block.

C) The instruction after the end of a basic block is a start of a next basic block.

In a single-pass code scan, start points of basic blocks are marked out according to the above Rule (A) and (B). These addresses are stored in a sorted link list. Then, according to Rule (C), end of basic blocks are generated. Any duplicated separation points are removed.

With this algorithm, basic blocks can be successfully isolated.

4.2.2 Instruction Filtering.

Before describing the algorithm for filtering instructions, definition of several terms should be given.

Definition: Latency. Latency is the number of cycles a certain type of FU (FU_{inst}) takes to finish its execution, written as $Lat(FU_{inst})$.

Definition: Performance Index. Performance Index (PI) represents the possible number of cycles between the cycle an instruction *(inst)* is issued and the cycle its result is first referred to, written as PI(inst). In this thesis, PI is often referred to as the *gap* between the issuing of an instruction and the first reference to its results.

An instruction with PI(inst)=m can thus be issued to a corresponding FU_{inst}^{k} with $Lat(FU_{inst}^{k})=n$ without degrading the overall performance if m>=n and FU_{inst}^{k} is one of the FU that corresponds to instruction *inst*. It can be seen that PI represents the execution laxity for any instruction and is the guiding light for our filtering algorithm. The aim of the filtering algorithm is then to estimate the PI for each instruction *inst* and then assign *inst* to a FU_{inst}^{k} with a longest latency that is smaller than PI: $Lat(FU_{inst}^{k})<PI(inst)$.

We impose the following assumptions when analyzing each of the basic blocks:

1) Values of all registers are readily generated immediately before the basic block;

2) The results of all arithmetic and logic instructions in the basic block will be referenced immediately after the current basic block;

3) The first instruction in a basic block is never issued together with instructions from the preceding basic block.

These assumptions are introduced because of the lack of context information, which is a native restriction of basic block based static code analysis. With these assumptions, we might occasionally misjudge the allowed latency for some instructions. Run-time events like cache miss and branching penalty might also slightly compromise the accuracy of our estimation. However, the inaccuracy in our analysis does not threaten the correctness of the program, but only results in run-time performance penalty which is later shown by our simulation to be trivial.

Based on the above assumptions, we can now emulate the behavior of the issue logic of an in-order issue processor by trying to issue as many instructions per cycle as the issue width allows until one of the operands of an instruction is not ready. To make sure the program runs at its best performance, the latency of each instruction is modeled as the smallest possible FU latency: $\min\{Lat(FU_{inst}^{k}),k=1...k_{max}\}$. Thus we can estimate the number of cycles between the time an instruction is issued and the time its result is first referenced by a following instruction. We use it as the estimation of the PI for that instruction. The data structures we employed are listed in TABLE III; while the algorithm is shown in Fig. 13.

TABLE III Data Structures Used in In-order Scheduling

Name	**Description**
Cycle	Cycle number starting from 0 for each basic block.
IP	Index of instruction within the basic block
Prod[*Rn*]	Index of the instruction which writes to register Rn.
Ready[*Rn*]	Number of cycles needed before value in Rn is ready.
Issue[*n*]	Cycle when instruction *n* is issued.
PI[n]	Performance Index of instruction n.

```
Cycle=0; Issue[0]=-1; IP=1;
While (IP<NUM_OF_INST_IN_Basic_Block)
{For (i=0;i<issue_width;i++) //Multiple Issue
 {  Inst=Get_Next_Inst(IP);
  If (!All_Operand_Ready(Inst)) break;
  For (each source operand inp of Inst)
    If (Prod[inp]!=0)
     PI[Prod[inp]]=MIN(PI[Prod[n]],Cycle-
                     Issue[Prod[inp]]);
  For (each destination operand dest of Inst)
  { Prod[dest]=IP; Ready[dest]=Inst.lat;}
  Issue[IP]=Cycle; // Record the issue time.
  IP++; // an instruction issued
 }
 Cycle++;
 For (each register Rn) // Time elapse
  If (Ready[Rn]>0) Ready[Rn]--;
}
For (Each Instruction n)
 PI[n]=MIN(PI[n],Cycle-Issue[n]);
```

Fig. 13 Algorithm for Performance Index Estimation

With the algorithm shown, we can estimate the PI for each FU related instruction. Now, depending on the available types of FU in a target processor, we can choose a most power-frugal FU for each instruction as long as the latency of the FU is smaller than the PI of the instruction. For example, if we have two FU with Lat(FU1)=1 and Lat(FU2)=4, we can assign instructions with PI=2 or 3 to FU1; while we choose FU2 for all those with an estimated PI above 4.

This is a one-pass code scanning algorithm with time complexity of O(n), where n is the number of instructions in the basic block. Obviously, it is highly efficient for implementation. After the FU choice for all instructions has been made, modified instruction codes are written back into the object code for later execution.

4.2.3 Simulation Results

For the simplicity of illustration, we only analyze integer ALU related instructions. Instructions using any other types of FUs can all be analyzed exactly in the same fashion. We carry out our experiments using the SimpleScalar Toolset [34] for PISA architecture, which implements the pipeline architecture described in Chapter 3. On the code analysis side, we composed the instruction filtering program within the toolset environment. Based on the algorithm described in the previous sub section, the PIs of integer ALU instructions in an object file are estimated and put as annotations in the 64-bit instructions. This represents modifying the op-code of the instructions. After that, the text segment with annotated instructions is saved back into the object file. Thus, we can generate new object files within which the PIs of ALU instructions are in the annotation field.

On the processor support side, we modified the *sim-outorder* simulator. As described in Chapter 2, we added extra power-frugal integer ALUs and modified the issue logic to identify the PIs in the annotation fields when issuing integer ALU instructions. Probes are also added to extract the actual percentage of ALU instructions that get issued to power-frugal integer ALUs when the benchmark programs are executed on large enough inputs. The processor configuration is listed in

TABLE IV.

TABLE IV Processor Configuration Used in In-order Scheduling

Functional Units	2 integer ALU		Latency = 1 cycle
	2 slow integer ALU		Latency = 2 cycles
	1 integer Multiplier		1 integer Divider
Decode Width	2 per cycle	Fetch Speed	2 per cycle
Issue Width	2 per cycle	Issue Mode	In-order issue
Commit Width	2 per cycle	Branch Predicor	Bimod

We analyzed eight pre-compiled integer benchmark programs from both SPEC95 and SPEC2000. The statistics generated in the static code analysis are listed in TABLE V. Static Instruction Filtering Percentage (SIFP) is the percentage of instructions filtered out to be issued to power-frugal ALUs.

TABLE V Code Analysis Results for In-order Processors

Benchmark	Total # of Inst.	% of ALU Inst.	SIFP
go (SPEC95)	70760	41.0%	14.6%
ijpeg (SPEC95)	45698	40.2%	14.0%
gcc (SPEC00)	310672	36.5%	12.6%
bzip (SPEC00)	18348	39.4%	15.5%
gzip (SPEC00)	25108	38.7%	14.5%
mcf (SPEC00)	14260	38.4%	14.8%
parser (SPEC00)	34800	37.5%	13.8%
vpr (SPEC00)	44828	39.7%	14.6%

From the above table, we can see integer ALU instructions are predominant in every program we analyzed, accounting for around 40% of all instructions. With the proposed time-efficient filtering algorithm, we succeeded in picking out around 15% of these ALU instructions as eligible for power-frugal ALUs with doubled latency

(PI=2). Note that all these benchmark programs were analyzed as given, without carrying out any code scheduling; hence higher percentage can be expected if optimization procedures are included.

To accurately estimate the efficiency of our approach in saving power, we should rely on the percentage of ALU instructions actually issued to power-frugal ALUs *at execution time*, that is, the Runtime Power-frugal ALU Issue Percentage (RPAIP), as shown in Fig. 14.

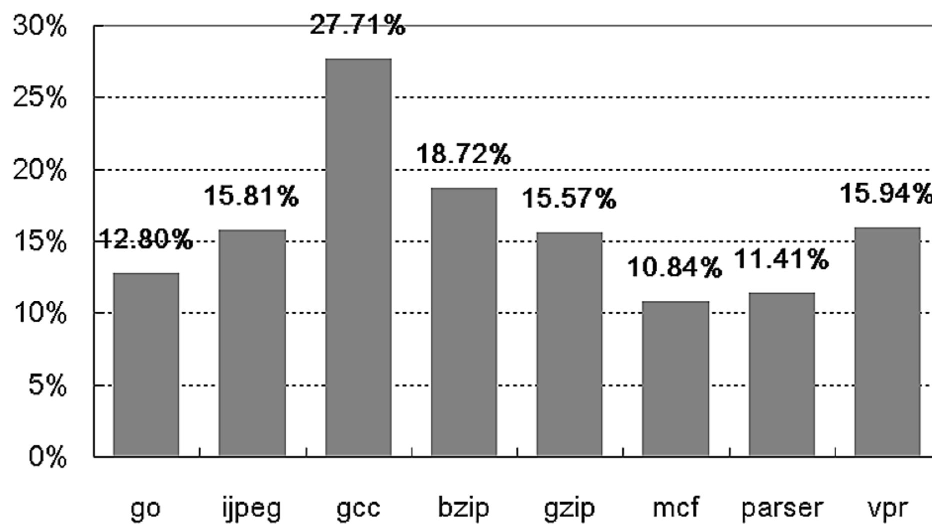
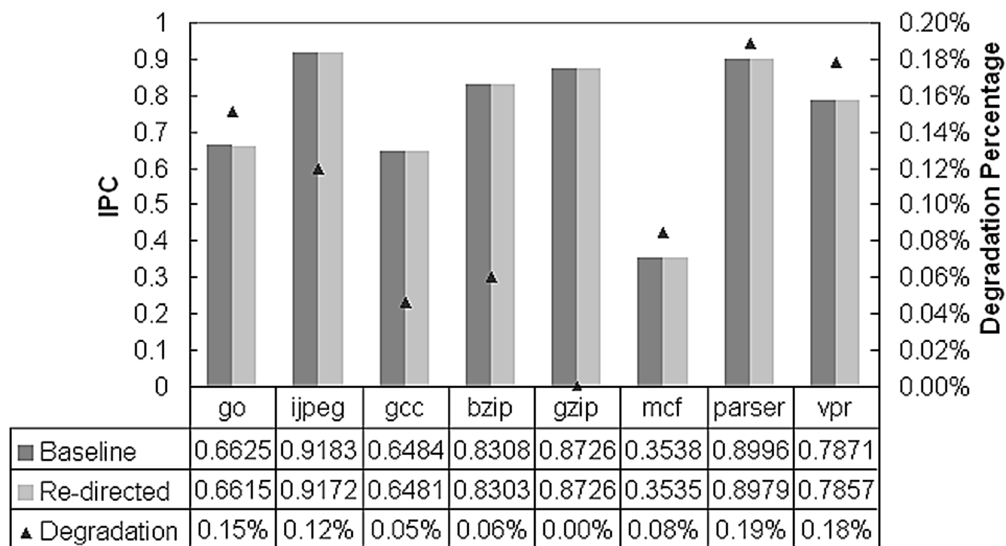


Fig. 14 Runtime Power-frugal ALU Issue Percentage (RPAIP)

Based on the above figure, it can be seen that with most programs, the proposed filtering algorithm resulted in re-directing 10.84% to 27.71% of all ALU instructions to power-frugal ALUs at run time. This RPAIP is different from SIFP basically because instructions are executed for the same number of times. Thus, RPAIP is a better measurement for actual power reduction. Assuming that a power-frugal ALU consumes 50% of the per-execution energy of a faster counterpart (as shown in Table II), we arrive at the estimation of 5% to 13% power-saving in the ALUs, based on

statistics in Fig. 14. This seems moderate, but as this technique can be combined with other lower power techniques, whatever amount saved here is an extra gain.

Even though the power-saving is achieved by lowering the performance of selected instructions, the overall performance of the programs is not supposed to be noticeably harmed. Use "Instruction per Cycle" (IPC) as the metrics to measure the overall performance of the programs. Fig. 15 shows IPC comparison of benchmarks before and after re-directing selected instructions to slow ALUs.



	go	ijpeg	gcc	bzip	gzip	mcf	parser	vpr
Baseline	0.6625	0.9183	0.6484	0.8308	0.8726	0.3538	0.8996	0.7871
Re-directed	0.6615	0.9172	0.6481	0.8303	0.8726	0.3535	0.8979	0.7857
Degradation	0.15%	0.12%	0.05%	0.06%	0.00%	0.08%	0.19%	0.18%

Fig. 15 IPC of Original and Modified Programs

As shown in Fig. 15, the performance degradation for utilizing slow and power frugal ALUs is negligible (less than 0.2%). Note that for this in-order issue processor, the number of fast ALUs is equal to the issue width. Thus there will not be resource conflict that hampers the issuing of instructions and the extra slow ALUs do not compensate the IPC of re-directed benchmarks. The performance degradation is mainly due to the inaccuracy that lies in the assumptions made when analyzing the

basic blocks. This shows the high accuracy of the proposed instruction filtering algorithm.

4.3 A step forward: Static Instruction Scheduling

The algorithm presented previously is one that does not re-order the original object code but only analyze it and make an off-line FU selection for instructions. Clearly, the amount of power reduction is limited by the sequence of instructions in the object code. Scheduling of the code may help exposing more instructions to be eligible for power-frugal FU. We have also developed an algorithm for scheduling the instructions. Interestingly, as the algorithm involves picking up suitable instructions to be filled in a next cycle slot, the algorithm can work for both in-order and out-of-order processors. The algorithm will be described in Chapter 6.

4.4 Chapter Conclusion

In this chapter, the code analysis technique for in-order processors is described. By simulating the issue logic of in-order issue processors, the issue time of instructions is estimated and the gap between interdependent instructions is used as the criteria for FU selection for instructions.

Chapter 5 Technique for Out-of-order Issue Processors

5.1 Overview

Out-of-order issue processors employ a more complex algorithm in issuing instructions. When an instruction has been decoded, the availability of its operands is checked. If some of its operands are not yet ready, the instruction will be issued into a Register Update Unit (RUU) [33] to wait for the generation of the operands, and instructions below can still be issued. Fake data dependence like WAW and WAR can also be resolved by hardware using Tomasulo's Algorithm and its extensions.

The complexity in the issue logic makes the estimation of the issue time and the time that an instruction is first referenced much more difficult than in-order processors.

As the instruction issuing behavior is more complex for out-of-order issue processors even within a basic block, it is not as simple as for in-order processors to filter out the candidate instructions to be executed in power-frugal FU. Instead of code analysis, we propose a different method for instruction filtering, which is based on pipeline status profiling. We also present the simulation results on power reduction.

5.2 Implementation

The guideline for FU choice is based on the actual gap (PI) between the issue of an instruction and the reference of its result. Thus, if we can record this PI for all instructions at run time, and through analyzing the profiled PI statistics for each instruction, the FU choice can be made. The op-code of these instructions can thus be

modified in the object file. This modified object file can then be executed on our proposed processor with extra power-frugal FU and dynamic power consumption can be reduced.

However, as each instruction will usually be executed for multiple times, we may get multiple PI values. This is because the issuing of instructions depends on many factors including the issuing of previous instructions that generate operands and the availability of FUs. In different iterations of the execution of a same instruction, these factors may vary and thus we may have varied PI values. In this case, the decision will be a statistical one.

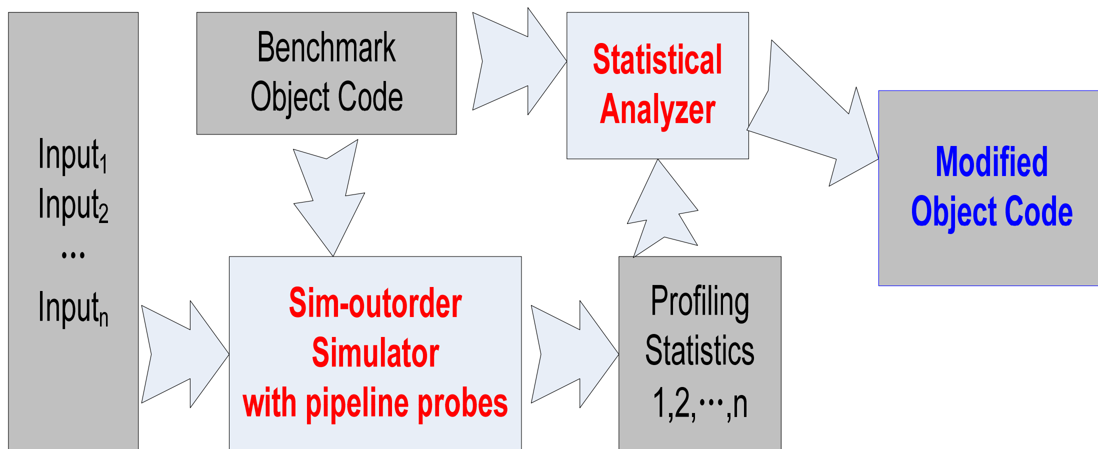


Fig. 16. Profiling Based Instruction Filtering System Structure

The pipeline profiling based instruction filtering is implemented within the SimpleScalar Toolset environment. The components are described in Fig. 16. With the sim-outorder simulator simulating the behavior of a generic out-of-order issue processor in detail, probes are added into various pipeline stages to record the gap time (PI) of each instruction in each iteration at run time. These PI values are saved into a hash table which is finally saved into a profiling statistics file. Varied inputs may be

used to generate multiple profiling statistics files. Subsequently, a statistical analyzer is developed to analyze these generated data files and in turn modify the correspondent object file. The detailed implementation is described here.

5.2.1 Recording PI values by Pipeline Profiling

In the first instance, object files together with proper input files are fed to the sim-outorder simulator for simulation. The behavior of each pipeline stage is simulated in detail. We put in probes in the **issue stage** to record the issue time of each instruction in the pipeline. This issue time is temporally saved in the instruction structure in the simulator.

Each instruction has two roles, either as a producer or a consumer.

Definition: Producer. Instruction *inst* is said to be the producer of register Rn, if Rn is a target register of *inst*.

Definition: Consumer. Instruction *inst* is said to be a consumer of register Rn, if Rn is a source register of *inst*.

We maintain data structures as described in the following table. These data structures are used in the commit stage. At **commit stage**, instructions are committed in original order, and each instruction will be processed both as a consumer and a producer of registers. The detailed processing is as given in Table VI.

TABLE VI Data Structures for Profiling Out-of-order Processors

Data Structures	**Description**

struct producer_rec_t
{
enum md_opcode op;
md_inst_t inst;
int valid;
md_addr_t PC;
tick_t issue_time;
long min_gap;
};
	Producer record item.
	It keeps the producer's op code (op) , instruction code (inst), memory address (PC), issue time (issue_time), current minimum gap (min_gap) and a valid bit (valid).

struct producer_rec_t Prod[Rn]	A list of producers of all registers.

struct PC_rec_t
{
struct PC_rec_t * next;
md_addr_t PC;
md_inst_t inst;
unsigned int num;
int cnt_less;
int max_gap;
int min_gap;
int FU_lat;
};
	The HASH table item for each instruction.
	It keeps a link to the next hash table item of the same hash value (next), the instruction address (PC), instruction code (inst), the number of times the instruction has been exeduted (num), the number of times the instruction has been executed with a gap less than the latency of its power-frugal FU(cnt_less), the maximum gap (max_gap) and the minimum gap (min_gap).

struct PC_rec_t HASH[]	The HASH table indexed by instruction address.

A) As a consumer. The issue time of the producers of all the source registers of *inst* are retrieved. If the gap between the issue time of any producer and the issue of *inst* is smaller than the current minimum gap time for the producer, the minimum gap time for the producer instruction is updated in the list of producers of all registers.

B) As a producer. *Inst* will be recorded as the current producer of all its target registers. However, before this is done, the previous producer record of all the target registers of *inst* should be saved. As the target registers of *inst* are to be updated by *inst*,

all the instructions afterwards are no longer referring to the results of the previous producers of these registers, thus the current minimum gap for the previous producers of these target registers is the final PI for them in this iteration. Hence, we update the instruction's record in the hash table. The hash table has been chosen as the data structure for recording the statistics of instructions because it can easily cover instructions over a wide range of memory locations and the efficiency of access is satisfactory.

With these two probes in **issue stage** and **commit stage**, the gap times (PI) of each instruction is recorded in the hash table. Essentially, the maximum gap, minimum gap, the number of times an instruction has been executed and the number of times the instruction has the minimum gap are recorded. When the simulation of the execution finishes, the whole hash table is saved in a data file for further processing by the statistical analyzer.

5.2.2 Statistical Analyzer

A statistical analyzer is developed to analyze the profiling statistics generated by the probes in the processor simulator. The analyzer will carry out the following processing.

1) Read the object code of a program and multiple profiling statistics files.
2) Match the profiling statistics to instructions in the object code.
3) Make a FU selection for each FU related instruction based on the profiling statistics.

4) Show various statistics including the number of FU related instructions found and the number of power-frugal FU chosen.

5) Modify the object code. Save the modified object code back to a new file.

The analyzer also shows the statistics and instructions in a readable way for easier debugging. The screen-shot of the analyzer is as below:

SS Analyzer

Text File: in\simplesim-3.0\go.dat.text [Load...]

Data File:

Data File Names	IPC
C:\cygwin\simple...	0.9565

[Add...] [Clear] [SaveBB]

Analysis Done!

[Analyze] Auto WriteBack

Src. SS File: C:\cygwin\simplesim-3.0\go.s [Load] Tag Threshold: 0.1

Output File: C:\cygwin\simplesim-3.0\go.s [Load] [Write Back]

Text Segment

Address	Instruction		# Min	Max	Min	FU	# Exe.
0x00448150	(addu	r21,r3,r23	57	5	1	1	291
0x00448158	(sll	r2,r12,2	0	9	2	1	291
0x00448160	(lui	r23,0x1007	291	1	1	1	291
0x00448168	(addiu	r23,r23,6192	291	1	1	1	291
0x00448170	(addu	r13,r2,r23	0	10000	245	1	291
0x00448178	(lw	r2,0(r21)	0	0	0	0	1047
0x00448180	(lw	r23,72(r29)	0	0	0	0	1047
0x00448188	(addu	r24,r23,r2	1047	1	1	1	1047
0x00448190	(sll	r2,r24,2	1047	1	1	1	1047
0x00448198	(addu	r2,r2,r17	811	4	1	1	1047
0x004481a0	(lw	r8,0(r2)	0	0	0	0	1047
0x004481a8	(sll	r5,r8,2	1047	1	1	1	1047
0x004481b0	(addu	r2,r5,r20	1045	2	1	1	1047
0x004481b8	(lw	r3,0(r2)	0	0	0	0	1047
0x004481c0	(lw	r2,0(r6)	0	0	0	0	1047
0x004481c8	(bne	r3,r2,0x4487b0	0	0	0	0	1047
0x004481d0	(lw	r23,0(r29)	0	0	0	0	361
0x004481d8	(beq	r8,r23,0x4487b0	0	0	0	0	361
0x004481e0	(addu	r2,r5,r7	164	1	1	1	164
0x004481e8	(lw	r2,0(r2)	0	0	0	0	164
0x004481f0	(slti	r2,r2,2	164	1	1	1	164
0x004481f8	(beq	r2,r0,0x4487b0	0	0	0	0	164
0x00448200	(lui	r23,0x1007	55	1	1	1	55
0x00448208	(addiu	r23,r23,-22592	55	1	1	1	55
0x00448210	(addu	r2,r5,r23	55	1	1	1	55
0x00448218	(lw	r5,0(r2)	0	0	0	0	55

Fig. 17. Statistical Analyzer Screen Shot

The core of this analyzer is the FU selection. As multiple PI values may be recorded at run time, a strategy is needed to make the FU selection. Considering the goal of keeping the execution performance uncompromised, a most straight forward way is to choose a FU with latency smaller than the minimum PI observed when profiling. This is a very rigid criterion and may result in too few power-frugal FU being chosen. For example, if an instruction has PIs of 2 for 999 times and has a PI of 1 only once due to a accidental structural hazard, a fast FU will be assign even though for 99.9% of the times its result is not immediately referred to in the next session. Obviously, this is not a good way of making the FU selection.

Alternatively, we may refer to the number of times an instruction has a gap less than the latency of its power frugal FU (num_less). If this number is small enough compared to the total number of times this instruction is executed (num), we may still choose a power frugal FU for it. Here, a threshold is employed. The FU choice is then made by: **if the ratio** $\frac{num_less}{num} \leq threshold$ **, power-frugal FU is chosen; otherwise, the high-power fast FU is chosen.**

The threshold represents a tradeoff between power and performance. A smaller threshold means less power-frugal FU will be used and the performance is better; while a larger threshold will save more power while might compromise the performance. Through thorough simulations, the optimal threshold may be decided according to specific power and performance consideration.

The generated modified object code is executable and carries extra op-codes that will be recognized by our processor with power-frugal FU. Power dissipation and execution performance can hence be measured and compared.

5.3 Pros and Cons of profiling based instruction filtering algorithm.

The pipeline profiling based instruction filtering algorithm is highly accurate in estimating the urgency of generating results for each instruction. The gap time measure is exactly what is happening within a processor core. Different from any code-analysis based approach, here no assumptions are taken. What is more, if supported by the simulator, cache miss and other factors can also be taken into account as they may also delay the need for fast generation of results.

However, this method also has its limitations. Firstly, the profiling statistics must be somehow generated. As hardware support for such profiling is forbiddingly expensive considering the data bandwidth and data amount, detailed processor architecture simulator is needed. This is somehow not a problem as modern processor developers all have such simulators for verification purpose. Slight modification to the simulators with the method described in this chapter can generate the statistics needed. But the accuracy of the simulator itself is still questionable and the simulation speed is also slow.

Secondly, as it is profiling based, the choice of input vector is very important. If the input vector is not sufficiently representative, parts of the object code may not be covered and thus the FU choice will not be made for that part of the program. Actually,

profiling test vector can hardly cover the full range of the object code, always leaving parts that are not analyzed. However, the impact is not as serious as it looks, as those not-covered parts are also less likely to be executed in reality if the test vectors are carefully chosen.

Thirdly, the profiling is a time consuming process for large input vectors on detailed processor structural simulators. The performance of the modified object code has to be tested on a second pass simulation. For deciding the optimal threshold for FU choice, multiple passes of simulation will be needed and the overall time could be very long. But as the each object code needs to be modified offline only once and can be used to save power dissipation in every run afterwards, it is well worth the time it takes in the instruction filtering stage.

5.4 Simulation Results

Here we present simulation results generated by applying the pipeline profiling based instruction filtering algorithm. Benchmark programs together with big input vectors are fed to the simulator with added probes to generate pipeline statistics. Hence, the object code and profiling statistics files are analyzed by the Statistical Analyzer to generate the modified object code. Essentially, the modified code carries the PI of instructions in the annotation filed of the instructions. The modified object code is in turn executed on the simulator with extra power-frugal ALUs again for measuring performance metrics.

5.4.1 System Configuration

The simulation is also done within the SimpleScalar Simulation Toolset [34].

The target processor architecture has the configuration as listed in TABLE VII Again, to simplify the processing; only integer ALU related instructions are processed. These instructions are predominant in the benchmark programs we selected and are always an important group of FU-related instructions in any program. Any other type of FU related instructions can be processed in exactly the same way.

TABLE VII Out-of-order Processor Configuration

Functional Units	4 integer ALU		Latency = 1 cycle
	2 slow integer ALU		Latency = 2 cycles
	1 integer Multiplier		1 integer Divider
Decode Width	4 per cycle	Fetch Speed	4 per cycle
Issue Width	4 per cycle	Issue Mode	Out-of-order issue
Commit Width	4 per cycle	Branch Predicor	Bimod

5.4.2 General Performance

By applying the proposed pipeline profiling based instruction filtering algorithm, we have generated the following simulation results in filtering out ALU instruction candidates to be issued to power-frugal integer ALUs. With each benchmark program, varied large inputs are used to fully cover the whole code range. In the statistical analyzer, a fixed threshold ratio of 0.1 is used. This means all the integer ALU related instruction who has a gap of larger than 2 in 90% of times are deemed to be suitable for issuing to power-frugal ALUs. In the following table, Code Coverage and Static Instruction Filtering Percentage (SIFP) are shown.

TABLE VIII Out-of-order Instruction Filtering Statistics

Benchmark	Code Coverage	ALU inst Percentage	SIFP
go (SPEC95)	78.7%	37.9%	42.8%
ijpeg (SPEC95)	72.4%	36.2%	43.7%
gcc (SPEC00)	69.7%	31.5%	56.0%
bzip (SPEC00)	79.5%	39.2%	39.3%
gzip (SPEC00)	68.1%	35.8%	49.5%
mcf (SPEC00)	76.2%	30.3%	54.8%
parser (SPEC00)	75.2%	32.2%	50.8%
vpr (SPEC00)	67.7%	33.1%	53.9%

As our approach is test-vector driven, the better analysis of the code depends on the coverage of the full object code. In this simulation, it is found that the coverage of the code has been quite good, with code coverage rates mostly around 70%. It can be seen, this instruction filtering algorithm is efficient in filtering out the power-frugal ALU instruction candidates. With a moderate threshold, statically around half of ALU instructions are filtered out for power-frugal execution. To accurately estimate the amount of energy saved, the modified object codes are re-simulated on the processor with power-frugal integer ALUs. The percentages of ALU-related instructions that are actually issued to power frugal integer ALUs (RPAIP) are as illustrated in the following table, together with the comparison of IPC for with and without power-frugal ALU execution.

TABLE IX Execution Simulation Metrics for Modified Codes

Benchmark	Baseline IPC	RPAIP	IPC with Power-frugal ALU
go (SPEC95)	0.9516	0.2633	0.9497
ijpeg (SPEC95)	1.3701	0.2620	1.3744
gcc (SPEC00)	1.0825	0.2509	1.0811
bzip (SPEC00)	1.4200	0.3198	1.4149
gzip (SPEC00)	1.4762	0.3455	1.4738
mcf (SPEC00)	0.6721	0.2873	0.6729
parser (SPEC00)	0.9892	0.2813	0.9887
vpr (SPEC00)	1.1703	0.2572	1.1623

The following figure illustrates the percentage of power-frugal ALU executions at run time. This can be directly used for estimation of power reduction.

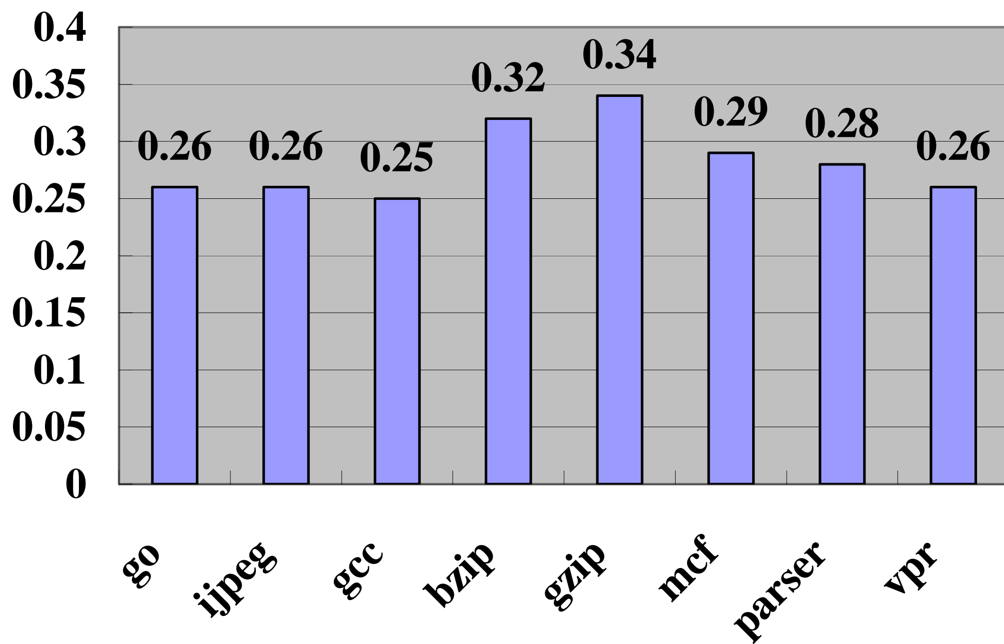


Fig. 18. Runtime Power-frugal ALU Issue Percentage

It can be seen, even though in most of the object code, about half instructions are filtered out to be executed in power-frugal ALUs, actually only less than 40% of all integer ALU instructions are actually issued to power-frugal ALUs. This is due to two reasons:

1) It is likely that those instructions that are filtered out to be executed in power-frugal ALUs have been executed for less number of times compared with those that were deemed not suitable for slower execution.

2) As only two power-frugal ALUs are included, structural hazard may force some of those filtered-out power-frugal instructions to be actually issued to fast ALUs.

Thus, it is clear that we have to base our power reduction estimation on run-time statistics RPAIP rather than the SIFP.

Again, IPC is chosen for comparison of the execution performance of these benchmarks. The same inputs were used for both cases and the IPCs are illustrated in the figure below.

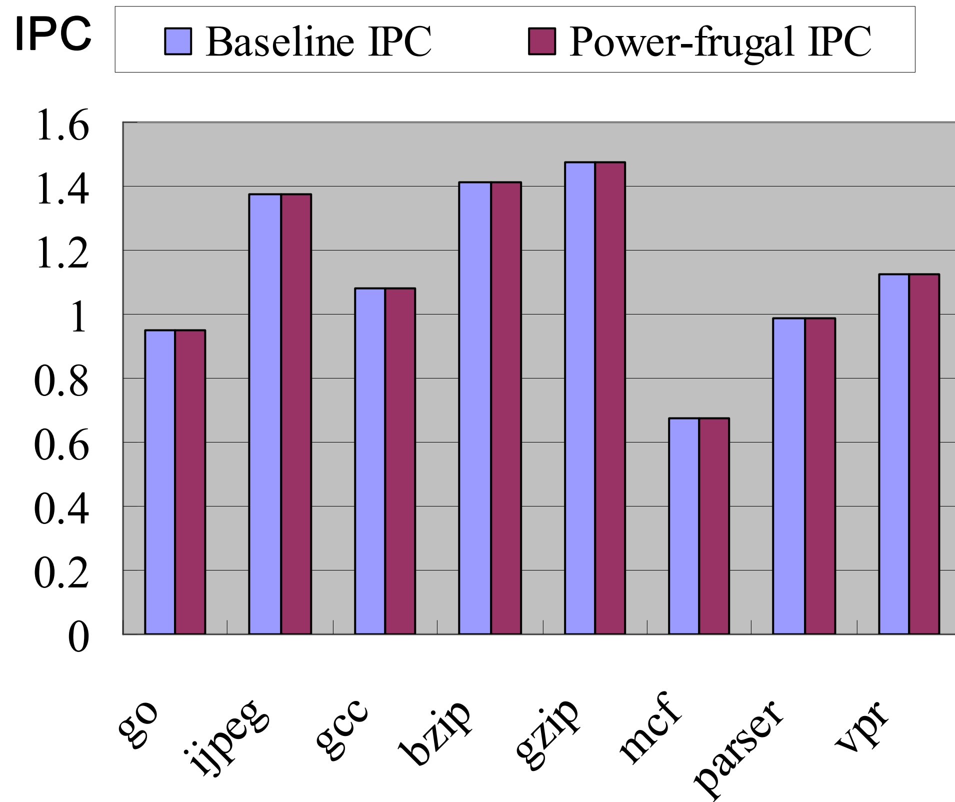


Fig. 19. Execution Performance Comparison (IPC)

As expected, such power reduction should not be accompanied by significant execution efficiency degradation. Fig. 19 shows the comparison of the IPC for original and modified object codes running. The difference of IPC is not noticeable. The difference of IPC is then shown in the figure below:

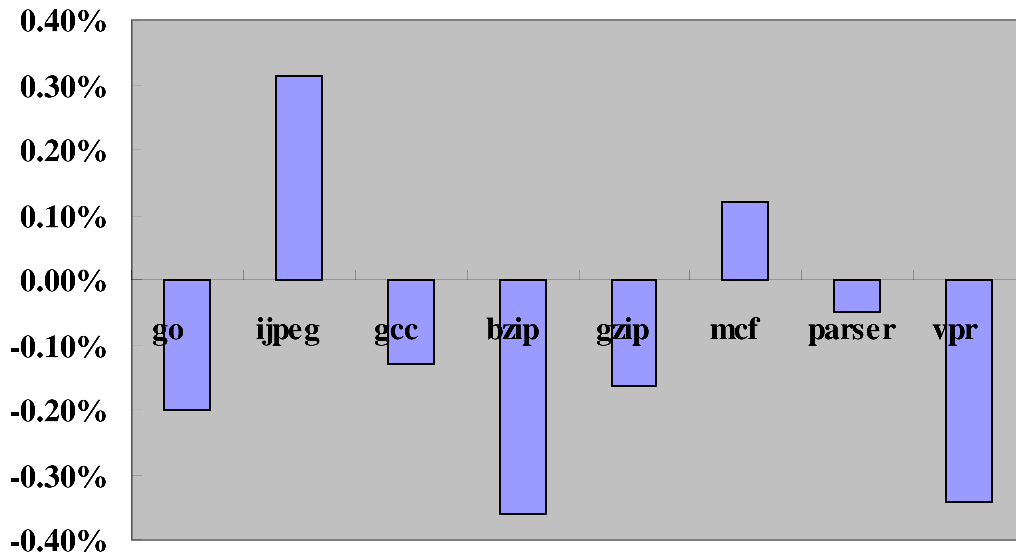


Fig. 20. Execution Performance Comparison (IPC)

Different from the results we have obtained with in-order issue processors, here some IPC of the modified object codes (*ijpeg* and *mcf*) are even better (faster) than the original counterpart. This is reasonable. In in-order processors, the number of integer ALUs is the same as the issue width and there will be at most the number of issue width of instructions set to execute each cycle. Thus, the inclusion of extra power-frugal ALUs will not benefit the execution efficiency. However, in the out-of-order issue architecture, there may be instructions awoke by the generation of register values which arrive together with the newly issued instructions. Thus the number of instructions to be executed at a single cycle may sometimes be more than the issue width. Hence, the inclusion of power-frugal ALUs will slightly benefit the execution efficiency and that explains the improvement of IPC. For ijpeg and mcf, the IPC benefit is not offset by the degradation incurred by the in-accuracy of FU selection.

Clearly, by including some extra power-frugal ALUs, a significant percentage of ALU instructions got executed at lower power cost while the overall execution efficiency is not compromised.

5.4.3 Impact of Threshold Ratio

In our method, there are several factors that should be carefully decided. For example, the threshold ratio in the statistical analyzer represents a knob that adjusts the balance of performance and power dissipation. The impact of the choice of this threshold ratio needs careful study.

We tried various threshold values with two selected benchmark programs (GO.SS and BZIP00.SS) and compare their power reduction and execution efficiency. The same simulation environment as the previous subsection is used here. Now the threshold ratio takes values ranging from 0.01 to 0.9. The simulation statistics are given in the following table. All the Static Instruction Filtering Percentage (SIFP), Runtime Power-frugal ALU Issue Percentage (RPAIP) and IPCs are averaged value over a spectrum of inputs of different quality.

TABLE X Impact of Threshold Ratio

$Ratio_{th}$	go (SPEC95)			bzip (SPEC00)		
	SIFP	RPAIP	IPC	SIFP	RPAIP	IPC
Baseline	0	0	0.9516	0	0	1.4200
0.01	39.1%	22.5%	0.9500	39.3%	29.3%	1.4150
0.1	42.8%	26.3%	0.9497	40.8%	31.2%	1.4149
0.2	45.3%	28.5%	0.9496	42.7%	32.5%	1.4146
0.3	47.5%	30.1%	0.9493	44.0%	33.3%	1.4142
0.4	49.6%	32.0%	0.9484	45.3%	34.2%	1.4141
0.6	53.9%	36.2%	0.9462	48.2%	37.3%	1.4115
0.8	58.3%	40.9%	0.9418	52.4%	39.7%	1.4088
0.9	60.9%	45.8%	0.9367	54.9%	41.7%	1.4013

A first observation is that the threshold ratio always has a linear relationship with the static number of instructions filtered out. This can be better illustrated by the figures below.

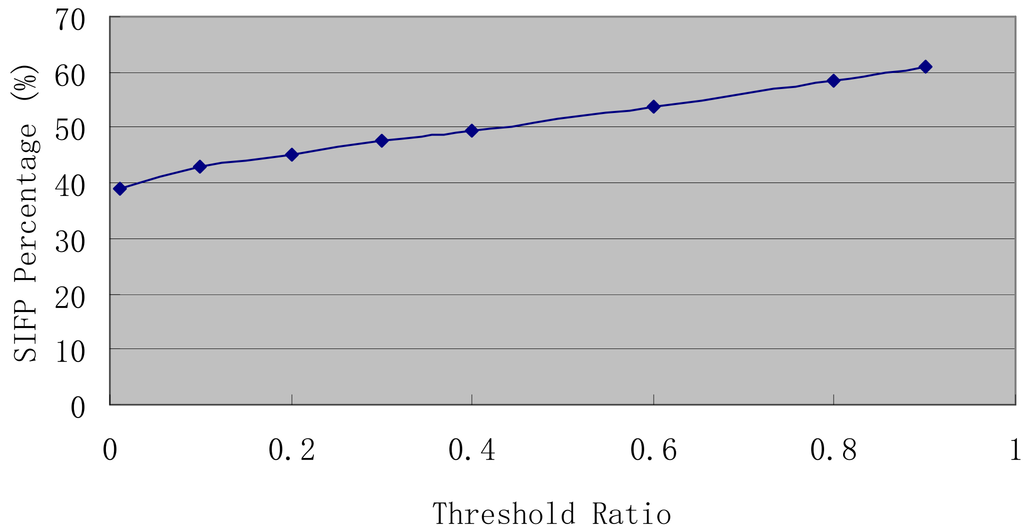


Fig. 21. SIFP for GO.SS with varied Threshold Ratio

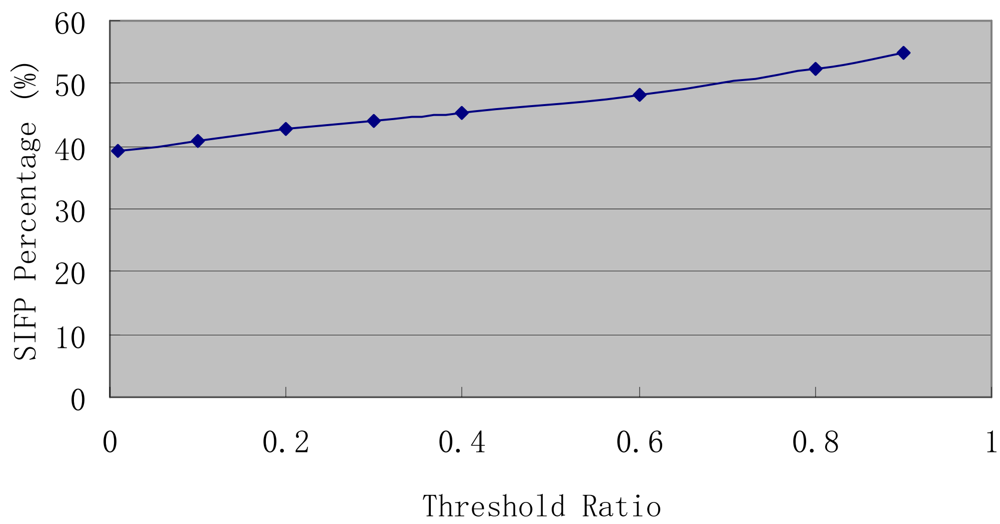


Fig. 22. SIFP for BZIP00.SS with varied Threshold Ratio

However, this linear dependency should not be used for reference when choosing a proper threshold ratio. Instead, RPAIP and the resulted IPC should be carefully studied. The following figures clearly illustrate the RPAIP and IPC dependence on the threshold ratio.

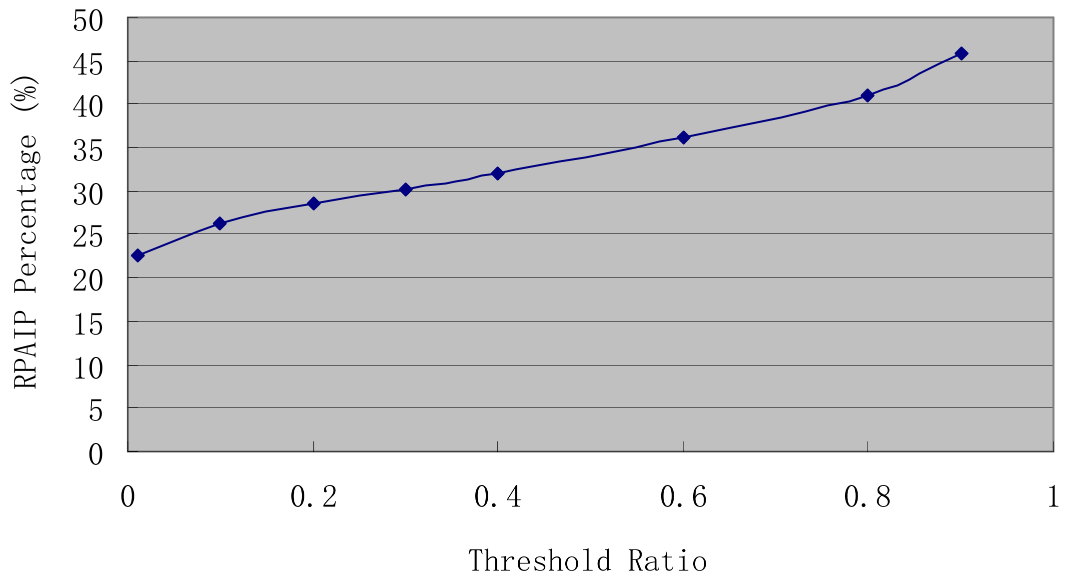


Fig. 23. RPAIP for modified GO.SS with varied Threshold Ratio

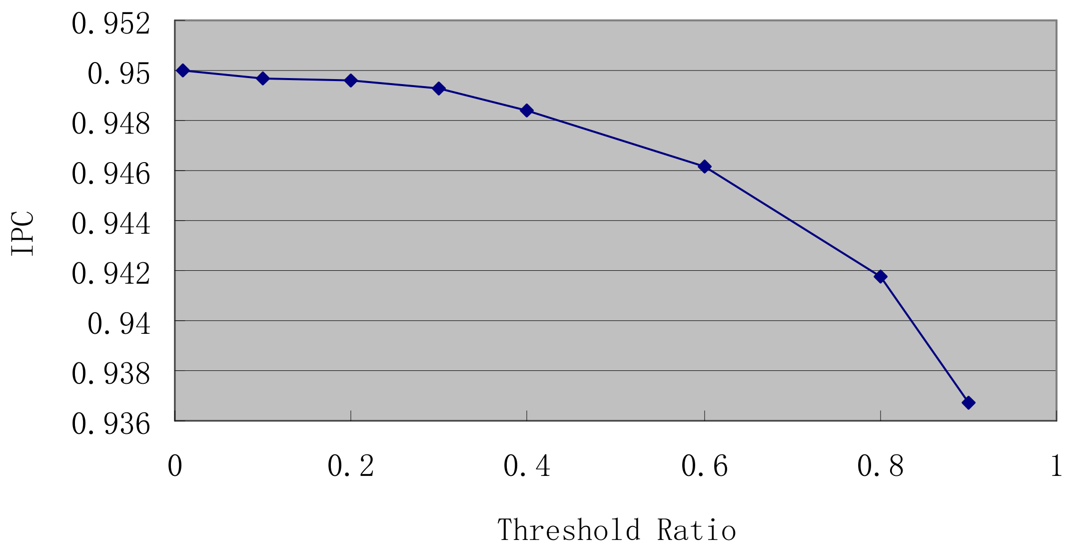


Fig. 24. IPC for modified GO.SS with varied Threshold Ratio

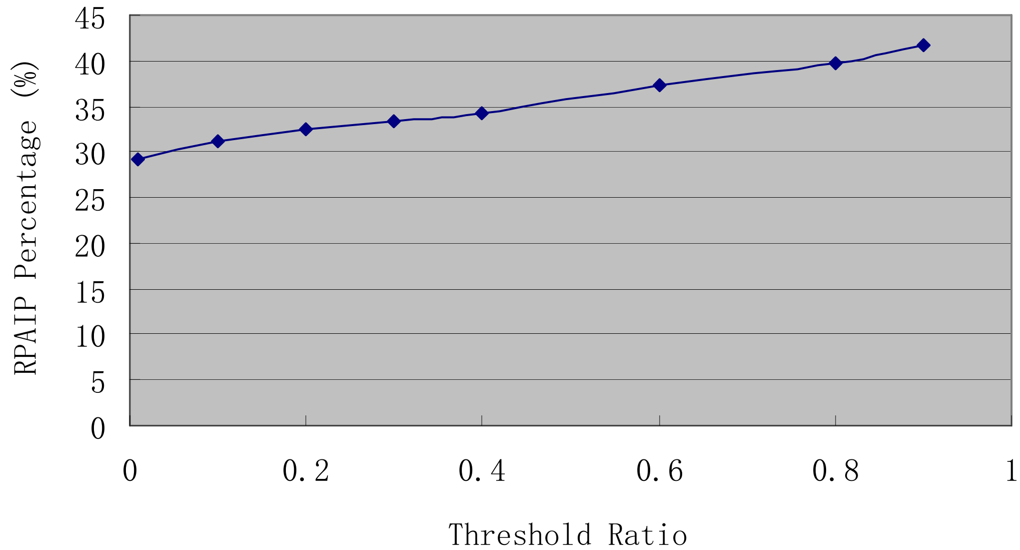


Fig. 25. RPAIP for modified BZIP00.SS with varied Threshold Ratio

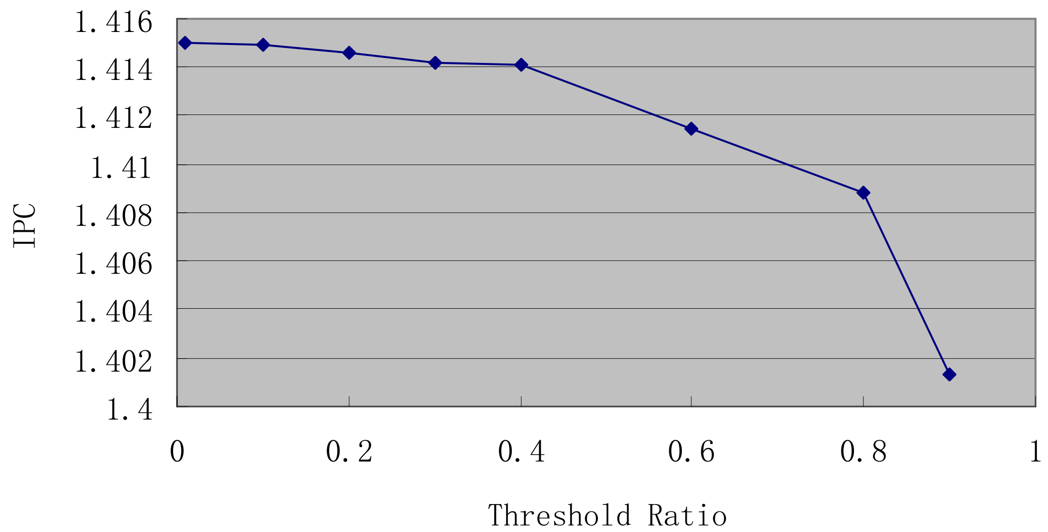


Fig. 26. IPC for modified BZIP00.SS with varied Threshold Ratio

From the above figures, it can be seen that with increased threshold ratio, the RPAIP first increases slowly, but when the threshold ratio exceeds a certain point, the RPAIP starts to increase exponentially. The IPC simply exponentially decreases with increased threshold ratio. Thus, to better trade off between power reduction (represented by RPAIP) and execution performance (represented by IPC), the threshold

value should be selected carefully according to the need in a specific application. As the performance is always a first choice, threshold ratio of less than 0.2 should usually be preferable.

5.4.4 Impact of the Number of Power-frugal FU

Another factor that should be taken into account is the number of Power-frugal FUs included into the modified processor architecture. Basically, in the modified processor architecture with extra power-frugal FUs, if an instruction carries an op-code for power-frugal FUs and in its issue cycle there is no available power-frugal FUs but faster compatible FUs are available, the issue logic will issue the instruction to a faster FUs so as to guarantee the performance as the first choice. Thus, adding more power-frugal FUs can reduce the possibility of filtered out power-frugal instructions being executed at higher power cost. However, on the other hand, the extra power-frugal FUs also incur more area cost. Thus, it is important to estimate the impact of the number of Power-frugal FUs and hence make a reasonable choice for the processor hardware architecture.

As the instruction filtering algorithm only finds out the instructions that can be issued to power-frugal FUs and does not deal with the actual power-frugal FU resource availability, a same modified object code can used for processors with varied number of power-frugal FUs. With the two selected benchmark programs (GO.SS and BZIP00.SS), 1 to 6 frugal ALUs are included for simulation and comparison. The issue width is fixed at 4 instructions per cycle. Simulation results are shown in the following table.

TABLE XI Impact of the Number of Power Frugal ALUs

# of Power-Frugal ALUs	go (SPEC95)		bzip (SPEC00)	
	RPAIP	IPC	RPAIP	IPC
Baseline	0	0.9516	0	1.42
1	0.2310	0.9498	0.2736	1.4150
2	0.2633	0.9497	0.3124	1.4149
3	0.2679	0.9497	0.3153	1.4149
4	0.2686	0.9497	0.3153	1.4149
5	0.2686	0.9497	0.3153	1.4149
6	0.2686	0.9497	0.3153	1.4149

These statistics are visualized in the figures below.

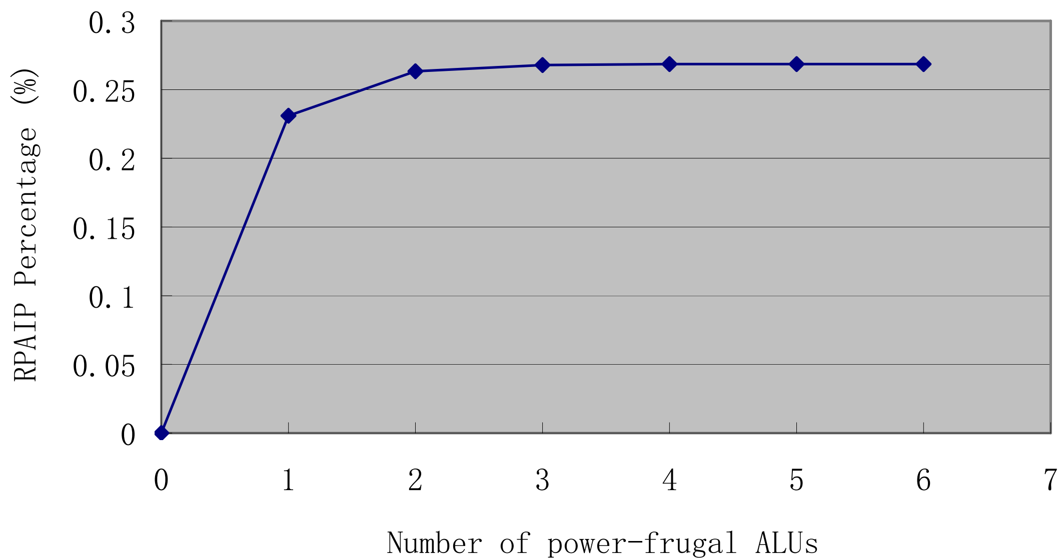


Fig. 27. RPAIP for modified GO.SS with varied Threshold Ratio

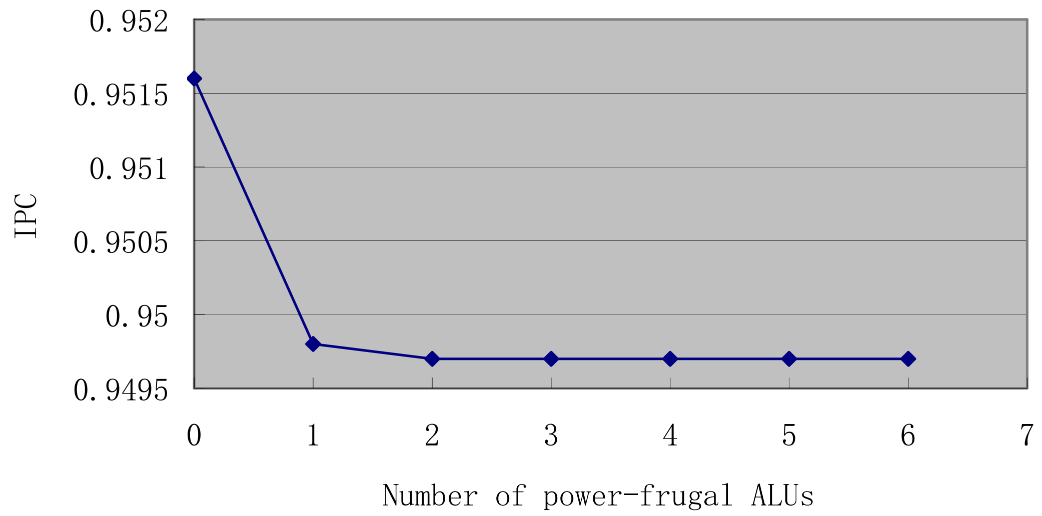


Fig. 28. IPC for modified GO.SS with varied Threshold Ratio

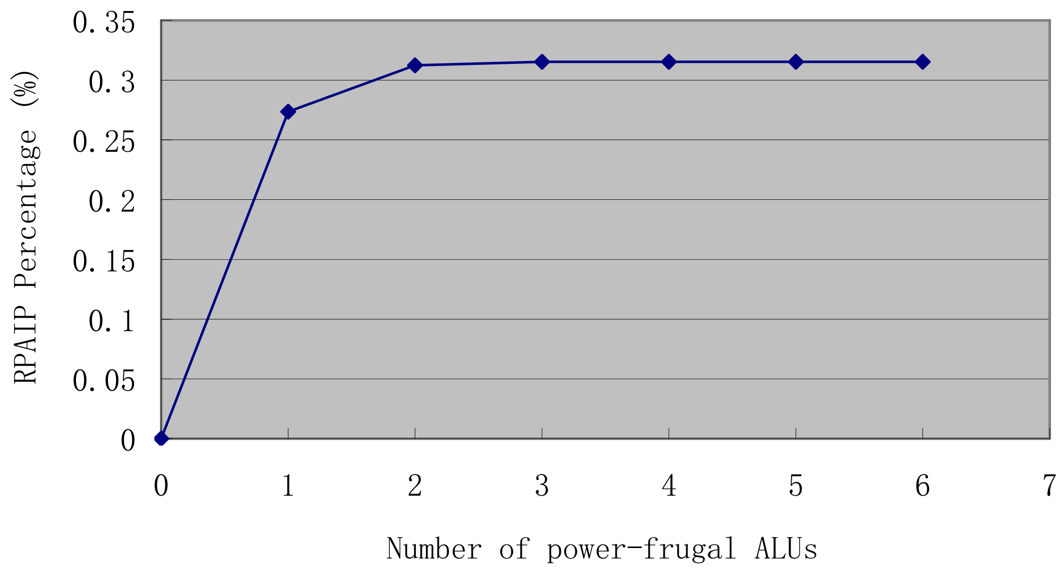


Fig. 29. RPAIP for modified BZIP00.SS with varied Threshold Ratio

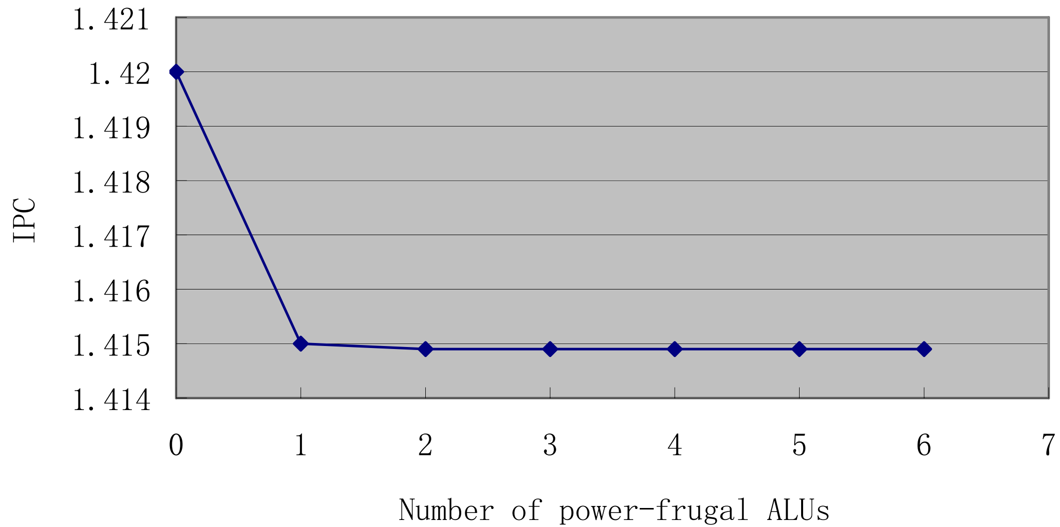


Fig. 30. IPC for modified BZIP00.SS with varied Threshold Ratio

Clearly, for both benchmarks, the impact of additional power-frugal ALUs is the same. The inclusion of more power-frugal ALU will allow more instructions to be issued to low power execution, with RPAIP increases and converges to a maximum value. With additional power-frugal ALUs added, IPC will be slightly degraded. This is because the filtering of instruction is not perfectly accurate and some instructions that should not have been issued to slow ALUs actually got issued to power-frugal ALUs. However, it is obvious that both RPAIP and IPC are not sensitive to the number of power-frugal ALUs include and more than 3 power frugal ALUs will have not further impact. This can be understood by that the need for power-frugal ALUs is totally met, as the converged RPAIP is less than 50%.

Thus we can conclude that with a processor of issue-width n, the number of power-frugal FU should be around $n*RPAIP_{converged}$.

5.5 Chapter Conclusion

In this chapter, we proposed a pipeline-profiling-based instruction filtering algorithm for out-of-order issue processors. This algorithm can be integrated into a detailed processor architecture simulator to generate accurate instruction execution statistics. Later, a statistical analyzer is developed to make the FU choice for each FU related instructions based on these statistics and generate modified object codes. Simulation results have shown a high percentage of integer-ALU related instructions (around 30%) can be issued to power-frugal ALUs. The impact of varied parameters in approach is also studied.

Chapter 6　Optimization: Static Instruction Scheduling

In the previous two chapters, two instruction filtering methods have been proposed to filter out instruction candidates for execution in power-frugal FU for in-order and out-of-order issue processors respectively. However, both methods only tag instructions and indicate in the modified object code which FU each instruction should be using. The original instruction order is preserved. Thus, the available number of instructions for issuing to power-frugal FU is limited, and the rate of power reduction is not up to the potential that actually exists. This can be illustrated in the example in Fig. 31.

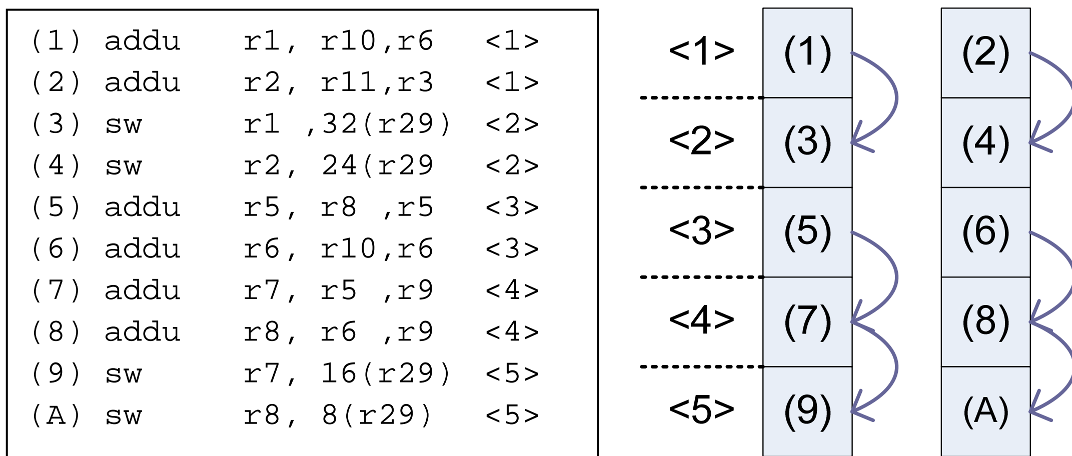


Fig. 31. Example: Original Code Sequence

With the above section of code, supposing an issue width of two, each cycle is fully used for issuing instructions and this section of code can finish within 5 cycles, as illustrated on the right. Clearly in this order of instructions, no ALU instruction can be issued to slower ALU as the results of instructions (1)(2)(5)(6)(7)(8) are all immediately needed by the next instruction. This is independent of the issuing scheme

of the processor. For both in-order and out-of-order issue processors, the issuing situation will be the same as each cycle the issue width are all met. Both instruction filtering algorithms could not improve the power efficiency. However, it can be seen that if we re-order these instructions and interleave the result generating and consuming instructions, we may considerably increase the number of instructions that can be issued to power-frugal ALUs. This is shown in Fig. 32.

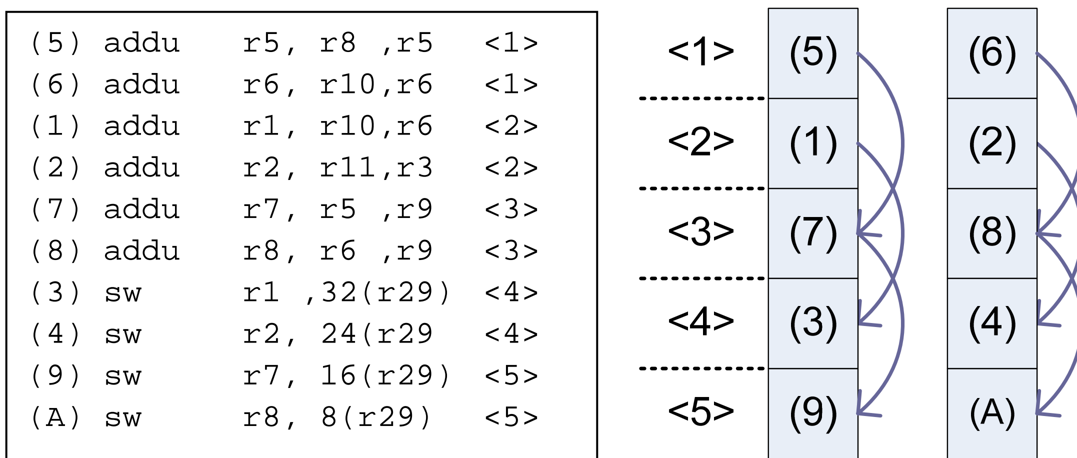


Fig. 32. Example: Re-ordered Code Sequence

In this case, the re-ordered section of code can still finish in 5 cycles but the results of instructions (1)(2)(5)(6)(7)(8) are not referenced by instructions issued in the next cycle and hence can all be executed in power frugal FU. This example shows the prospect of further power-saving by scheduling the basic blocks. Simulation in the coming subsections shows we can significantly improve the power saving efficiency by re-ordering the code and the scheduling algorithm is described here.

6.1 Scheduling Objective

The objective of the re-ordering of the instructions is to apply as many

power-frugal FU for instructions as possible while still keeping the number of cycles needed to execute a basic block of code. There are two tasks involved in this scheduling: 1) instruction re-ordering; 2) FU choice. To formally state the scheduling problem, the following definitions are needed.

Definition: Execution Duration. The execution duration of a basic block is the number of cycles needed to finish executing the basic block under the assumption that the basic block enters an empty pipeline.

Definition: Power Cost (PC). The power cost of a basic block is the sum of the single-execution energy consumption of each of the FU utilized in executing the basic block. It can be represented by:

$$PC = \sum_i E(FU_i) \tag{12}$$

Definition: Validity of Reordering. A re-ordering of the instructions in a basic block is said to be valid if all the register and memory modifications incurred in executing the re-ordered basic block is exactly the same as those incurred with the original basic block.

LEMMA: The validity of re-ordering is preserved if the Data Dependence Graph of the original basic block is preserved in the re-ordered one. The proof of this lemma is simple and thus omitted here.

Now the scheduling problem can be formally stated as:

Given a basic block of instructions, try to re-order the instructions and at the same time choose a proper FU for all the FU related instructions so as to:

1) Minimize the execution duration of the basic block.

2) With same minimum execution duration, minimize the corresponding power cost of executing the basic block.

With the limitation that the re-ordered basic block must be a valid one.

6.2 Scheduling Algorithm

This scheduling problem is an NP complete problem as the basic-block based instruction re-ordering problem, which is NP complete, is only a special case of our scheduling problem. To get the optimal solution, we have to explore all the possible solutions, which is very time-consuming. We take a compromised approach where we try to explore the solution tree, ignoring unlikely solution nodes as early as possible and limit the expansion of the solution tree when it is likely to grow out-of-control. The scheduling is done in several steps, as described in the sub-sections below.

6.2.1 Inter-dependence Table Generation

Firstly, given the object code of a program, we have to chop it up into basic blocks. Here the same algorithm of basic block division in Section 4.2.1 can be used. Then, with each of the basic block, the instruction inter-dependence table (IDT) is to be generated. This IDT represents the interdependence between instructions within a basic block and will be used as the guideline for valid instruction orders.

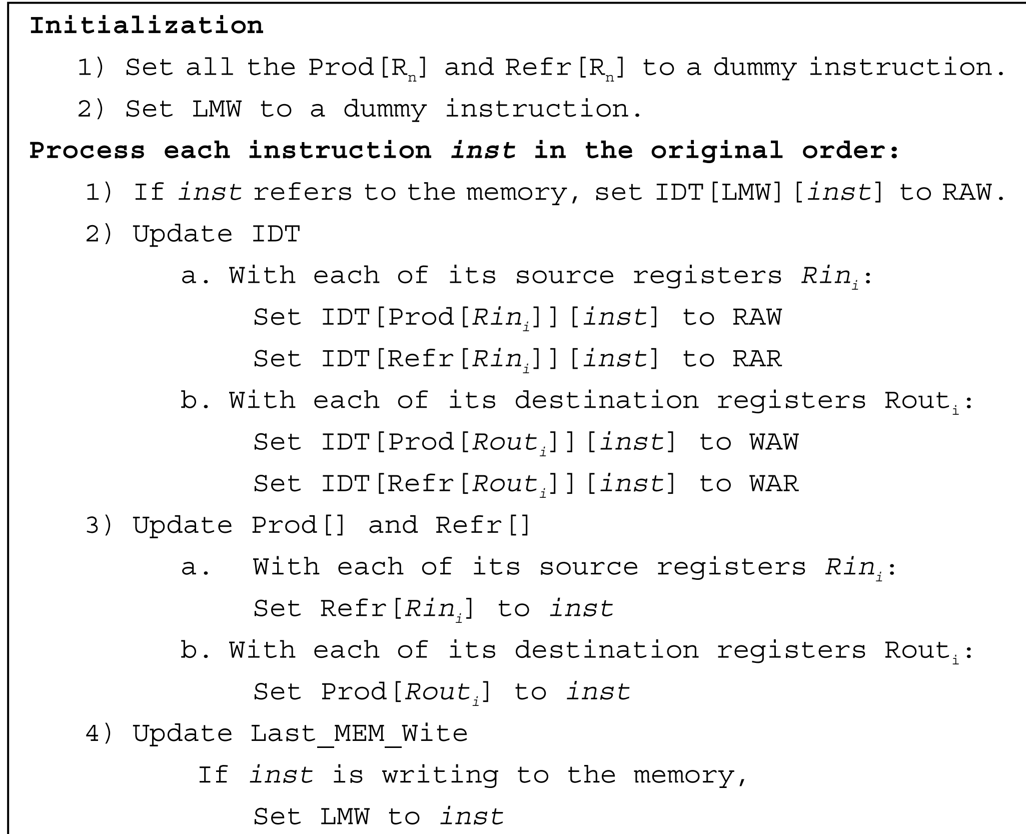
The IDT is a two-dimensional table where IDT[i][j] represents the dependence relationship between the i-th instruction and j-th instruction in the basic block. The possible dependence relationships are listed in the following table.

TABLE XII Interdependence Relationships

Relationship	Description
NONE	The j-th instruction is not dependent on the i-th instruction.
RAW	Read After Write. The j-th instruction refers to the result of the i-th instruction.
WAW	Write After Write. The j-th instruction writes to a same destination register as the i-th instruction.
WAR	Write After Read. The j-th instruction writes to a register that is referred to by the i-th instruction.
RAR	Read After Read. The j-th instruction reads a register that is referred to by the i-th instruction.

Obviously, the IDT[i][j] is always NONE if j=<i, as an instruction should not be dependent on an instruction that is after it in the original code order.

To build the IDT, the original basic block is scanned and the current producer and consumer of each register are kept in two lists Prod[R_n] and Refr[R_n]. The IDT building algorithm is listed below.



```
Initialization
   1) Set all the Prod[Rn] and Refr[Rn] to a dummy instruction.
   2) Set LMW to a dummy instruction.
Process each instruction inst in the original order:
   1) If inst refers to the memory, set IDT[LMW][inst] to RAW.
   2) Update IDT
        a. With each of its source registers Rini:
             Set IDT[Prod[Rini]][inst] to RAW
             Set IDT[Refr[Rini]][inst] to RAR
        b. With each of its destination registers Routi:
             Set IDT[Prod[Routi]][inst] to WAW
             Set IDT[Refr[Routi]][inst] to WAR
   3) Update Prod[] and Refr[]
        a.   With each of its source registers Rini:
             Set Refr[Rini] to inst
        b. With each of its destination registers Routi:
             Set Prod[Routi] to inst
   4) Update Last_MEM_Wite
          If inst is writing to the memory,
             Set LMW to inst
```

Fig. 33 Algorithm for IDT Generation

For the safety of memory referring instructions, all the memory access instructions are set as RAW dependent on the previous memory writing instruction (LMW). This guarantees that all the memory referencing instructions are kept in the original order when the basic block is re-ordered. With this algorithm, we can generate the IDT for a basic block in a single-pass scanning. IDT carries the interdependence between any two instructions in the basic block.

6.2.2 Equivalence Check

As the solution tree will be traversed, which is a very time consuming procedure, certain unnecessary branches on the solution tree should be trimmed. For example, if two instructions A and B are both using the same source registers and are both

generating a result that is not referenced within the basic block, then the relative sequence between A and B does not make any difference for the scheduling.

As a result, we need to detect all the equivalent instructions and record the equivalence for use in the scheduling algorithm.

Definition: Instruction Equivalence & Group. Two instructions i and j are said to be equivalent if the following conditions are met at the same time:

1) For any other instruction k, IDT[k][i] and IDT[k][j] are either both RAW or both not RAW.

2) For any other instruction k, IDT[i][k] and IDT[j][k] are either both RAW or both not RAW.

3) The FU related to instruction i and j must be the same.

With this definition, equivalence of instructions within a basic block is analyzed and recorded for use in the scheduling algorithm. Equivalent instructions are said to be of a same **group**.

6.2.3 Scheduling Algorithm

With IDT and Equivalence of instructions in a basic block generated, now we are ready to explore the solution tree to search for the optimal or sub-optimal program order and assumed FU choice. We need the following definitions for describing the algorithm.

Definition: Dominance. Instruction i is said to dominate instruction j if IDT[i][j] is RAW. This dominance essentially means instruction i must appear before instruction

j in the scheduled basic block.

Definition: Semi-Dominance. Instruction i is said to semi-dominate instruction j if IDT[i][j] is WAW or WAR.

Definition: Scheduled (Unscheduled) Instruction. An instruction inst is said to be a scheduled instruction for cycle n if *inst* is already filled in cycle m where (m<n). An instruction *inst* is said to be an unscheduled instruction for cycle n if inst is not a scheduled instruction for cycle n.

Definition: Live Instruction. An instruction *inst* is said to be a live instruction for cycle n, if *inst* is filled in cycle m (m<n) and (m + the assumed FU latency for inst >n). A live instruction must be a scheduled instruction. In the scheduling algorithm, for each cycle, all the live instructions are kept in a list LIVE[].

Definition: Quasi-Ready Instruction. An instruction *inst* is said to be a quasi-ready instruction for cycle n if *inst* is not dominated by any un-scheduled or live instruction for cycle n.

Definition: Ready Instruction. An instruction *inst* is said to be a ready instruction for cycle n if *inst* is a quasi-ready instruction and is not semi-dominated by any quasi-ready instruction for cycle n.

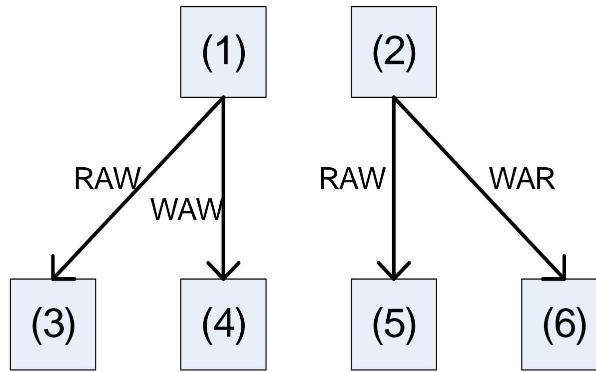


Fig. 34 Example for Ready and Quasi-Ready Instructions

The difference of ready and quasi-ready instructions can be illustrated by the above example. Suppose an issue width of 2 and 6 instructions in a basic block, where the interdependence is illustrated in the figure. Then for the first cycle slot, instruction (1), (2), (4) and (6) are ***quasi-ready*** instructions as they are not dominated (RAW) by any instruction. However, only instructions (1) and (2) are ***ready*** instructions as they are not even semi-dominated (WAW or WAR) by any other instruction.

The scheduling algorithm works in several steps as illustrated in Fig. 35. The rectangle boxes represent processing steps and the ellipsis's represent the input and output data of each processing step.

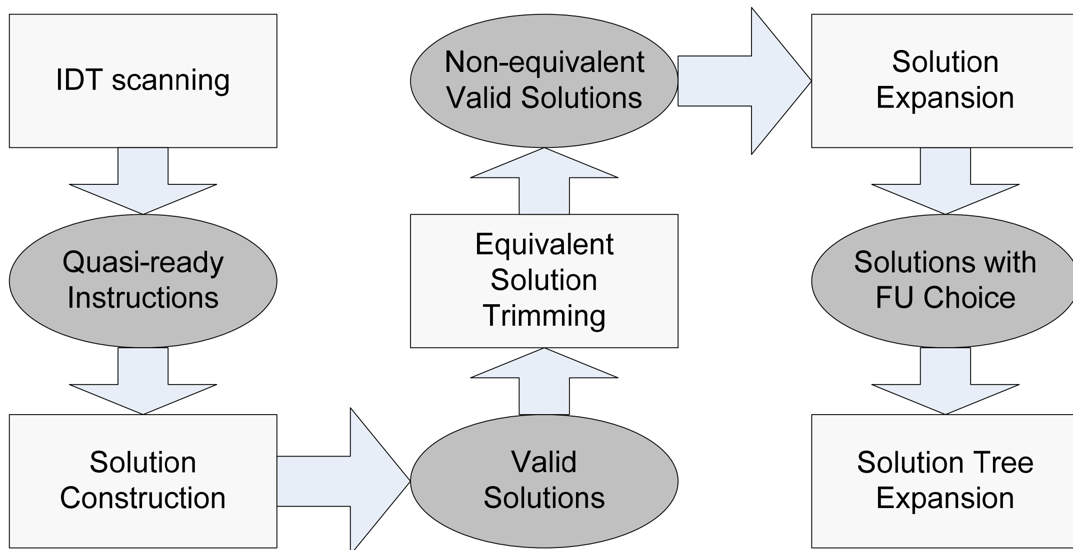


Fig. 35 Processing Steps for Basic Block Scheduling

It is a cycle based instruction picking process. For each assumed cycle, IDT is first scanned to build a pool of all the quasi-ready instructions. This is a simple scanning and filtering process.

Then, all the possible valid combinations of instructions to be filled in the current cycle slot are generated. These combinations of instructions are called solutions for a specific cycle slot.

A **valid** solution is one that meets either of the following two conditions:

1) consists of all ready instructions
2) consists of some quasi-ready instructions and all the un-scheduled instructions that semi-dominate these quasi-ready instructions.

Instructions in a solution must preserve their original relative order as in the original basic block.

In this step, if the number of quasi-ready instructions is less than the issue width,

one and only one valid solution can be generated. If the number of quasi-ready instructions is larger than the issue width, the number of solutions could be several.

With the generated valid solutions, a trimming step is taken to remove all the equivalent solutions to improve the efficiency of this scheduling algorithm. Two solutions are said to be equivalent if they have an equal number of equivalent instructions in each group.

After removing equivalent solutions, the remaining solutions are expanded with regard to FU selection. That is, if a solution has n FU related instructions and each of them has m possible versions of FU of different latency, the solution will be expanded to m^n solutions, each of which represents a different FU selection combination for the FU related instructions.

Now all the possible instructions to be issued in the current cycle have been generated. Each of these solutions will be tried for the current cycle and the live instruction list and scheduled instruction list will be updated accordingly before the next cycle is triggered. The solution tree is illustrated in Fig. 36.

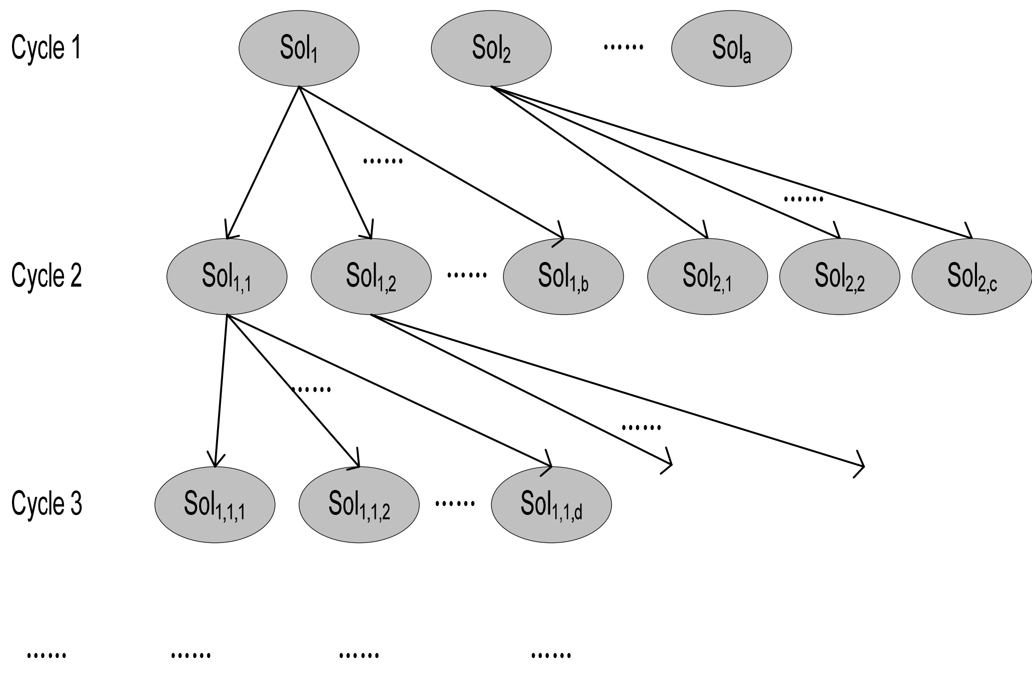


Fig. 36 Sample Solution Tree Aligned to Cycle Numbers

When all the instructions in the basic block has been scheduled, the trace from the cycle 1 level solution node to the last cycle level solution node represents an unique instruction order and FU selection assumption. The total number of cycles represents the estimated execution duration and the FU selection assumption can be used to estimate the overall power cost (PC). Hence, various instruction orders with FU selections can be compared according to our scheduling objective and an optimal solution can be found.

6.3 Discussions

6.3.1 Issue Scheme: In-order or Out-of-order?

The scheduling algorithm above does not deal with the issue scheme of the processor. Only the issue width is a parameter for the algorithm. This is because the out-of-order issue processors are only to make the instruction re-ordering at run time

within a limited range. As we statically re-order the instructions and try to fill as many instructions to our cycle slot as possible, the re-ordered code should be optimized for both in-order and out-of-order processors. In short, the scheduling algorithm is issue-scheme independent.

6.3.2 FU Selection

With the algorithm presented in the previous section, an optimal path on the solution tree corresponds to an instruction order and the FU selection assumptions. Now we have two choices: 1) use the FU selection assumptions directly as the FU selection for the instructions and generate an object file accordingly; 2) only save the instruction order in the object file and rely on the instruction filtering algorithms developed in the previous chapters to generate the FU choices.

For in-order issue processors, it is obvious that we should directly apply the FU selection assumptions as FU selection for the instructions as the instruction filtering algorithm is also a static estimation of the instruction issue cycles.

However, for out-of-order issue processors, the pipeline profiling based instruction profiling algorithm is a more accurate estimation of the issuing situation for all the instructions and we have an additional knob for the trading-off between power dissipation and execution efficiency. It might be desirable to re-filter the instructions with the statistical analyzer. However, simulation results in the coming section show that the improvement by employing a separate pipeline profiling based instruction filtering step is not significant and it also has the limitation of being test-vector based and time consuming. The final decision of whether to employ it or not depends on the

specific design requirement for a specific project.

6.4 Simulation Results

Here the simulation results pertaining to the scheduling algorithm are presented. For simplicity, we only analyze integer ALU related instructions and provide two types of integer ALUs for use with instructions.

6.4.1 In-order issue processors

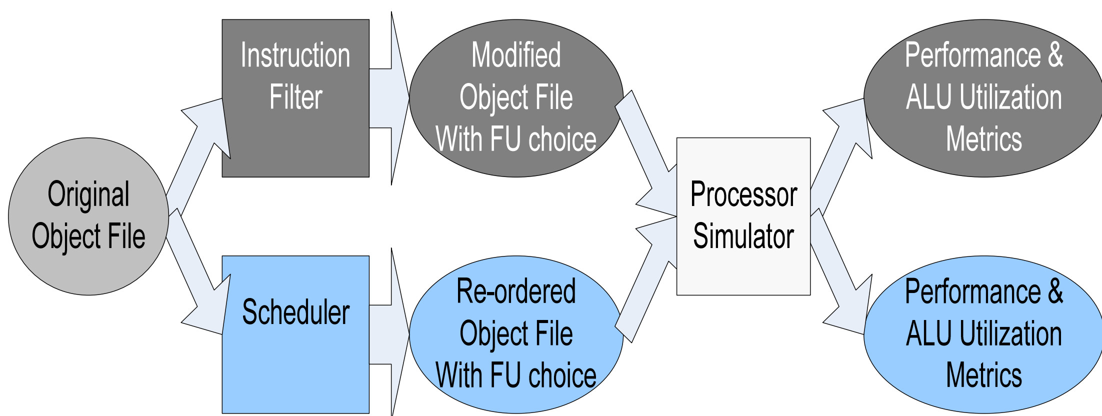


Fig. 37 Simulation Scheme for In-order Issue Processors

For in-order issue processors, we directly use the FU selection assumptions in the scheduling algorithm as the FU choice for instructions. Thus, the scheduler will take an object file, schedule it and generate an optimized object file. The optimized object file is then executed on the simulator to generate performance and integer ALU utilization metrics. These results are then compared with those we generated in Chapter 4.

The statistics of the scheduled object codes generated by the scheduling algorithm are listed in the following table, together with the runtime metrics measured in simulation. Here the same processor architecture in Chapter 4 is used. The detailed

configuration is listed in

TABLE IV.

TABLE XIII Statistics Of Scheduled Codes

Benchmark	SIFP	RPAIP	IPC with Power-frugal ALU
go (SPEC95)	0.192826	15.11%	0.6577
ijpeg (SPEC95)	0.158193	20.64%	0.9135
gcc (SPEC00)	0.126516	27.9%	0.6479
bzip (SPEC00)	0.165367	20.67%	0.8238
gzip (SPEC00)	0.193232	19.69%	0.8692
mcf (SPEC00)	0.190045	15.82%	0.3499
parser (SPEC00)	0.16332	14.16%	0.8915
vpr (SPEC00)	0.176809	20.09%	0.7812

we compare the RPAIP and IPC with those of object codes processed by the instruction filtering algorithm only, as illustrated in Fig. 38.

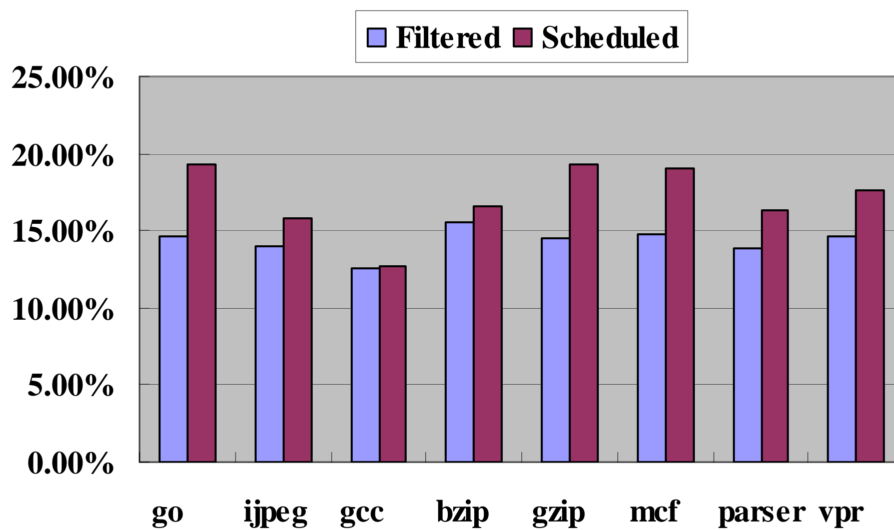


Fig. 38 SIFP Improvement of Scheduled code (compared with Filtered code)

As illustrated in Fig.38, by re-ordering the codes, more instructions can be exposed to have a larger PI and hence statically filtered out to be issued to power-frugal ALUs. This in-turn resulted in the improvement of the RPAIP, which directly corresponds to run-time power savings, as shown below.

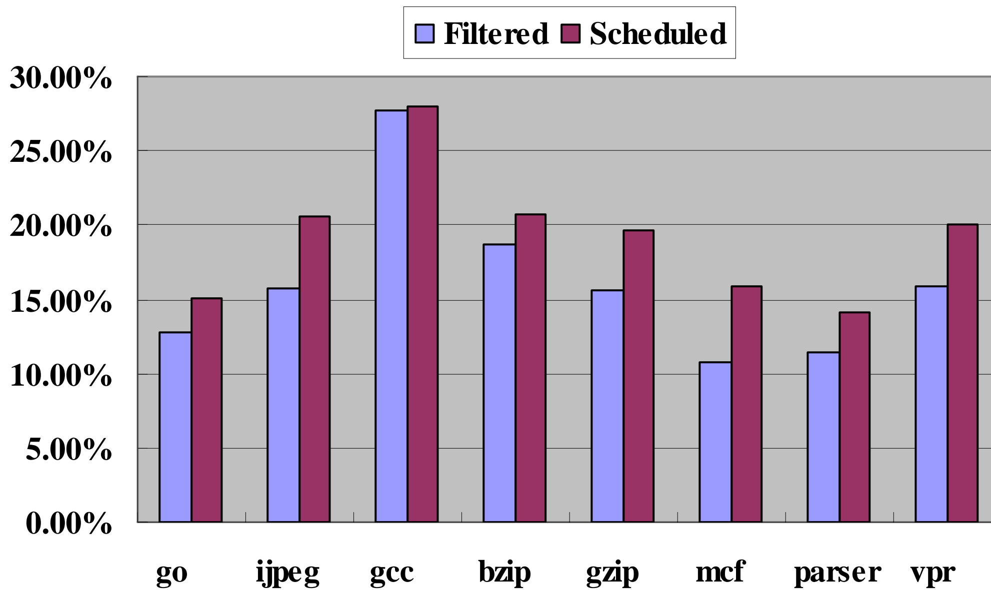


Fig. 39 RPAIP Improvement of Scheduled code (compared with Filtered code)

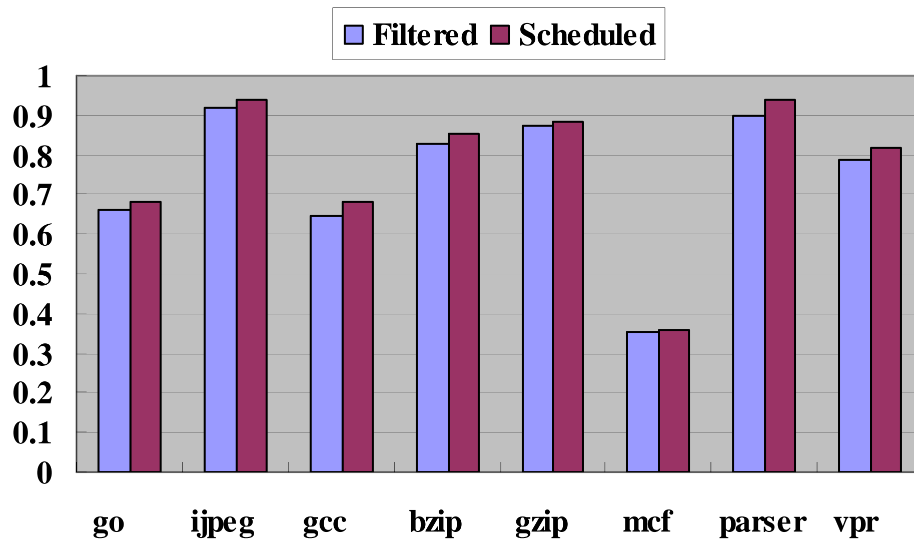


Fig. 40 IPC of Scheduled code (compared with Filtered code)

Fig. 40 shows that the IPC of the scheduled code is usually better than the filtered code. It has to be noted that the improvement of IPC comes together with more dynamic power savings. This is because our scheduling algorithm takes performance as first objective in scheduling. Obviously, the scheduler improved power saving together with the instruction execution efficiency and is a better way than the one we proposed in Chapter 4.

6.4.2 Out-of-order issue processors

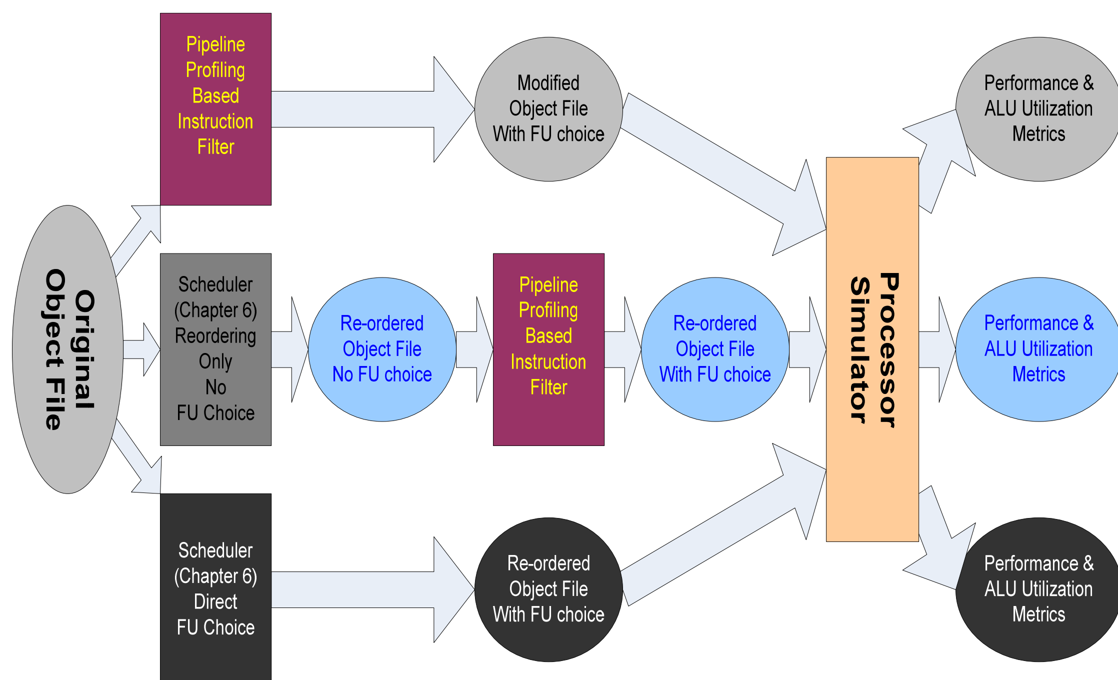


Fig. 41 Simulation Scheme for Out-of-order Issue Processors

For out-of-order issue processors, here we generate 2 groups of results as illustrated in Fig. 4. 1) Apply the scheduling algorithm which only generates re-ordered object file, followed by the pipeline profiling based instruction filtering algorithm which makes the FU choices; 2) Apply the scheduling algorithm and directly make the FU choice based on the FU selection assumption. Performance and ALU

utilization metrics are then compared with those generated by only applying the pipeline profiling based instruction filtering algorithm.

This simulation is carried only on two selected benchmark programs (GO.SS from SPEC95 and BZIP00.SS from SPEC2000). Table XIV lists the statistics generated with the proposed two methods.

TABLE XIV Impact of the Number of Power Frugal ALUs

Type of Optimization	go (SPEC95)			bzip (SPEC00)		
	SIFP	RPAIP	IPC	SIFP	RPAIP	IPC
No Optimization	0	0	0.9516	0	0	1.4200
Filtered Only (Threshold 0.1)	42.8%	26.3%	0.9497	39.3%	32.0%	1.4149
Re-order & Filtered	37.6%	29.8%	0.9564	31.4%	38.2%	1.4311
Re-order & Direct FU Selection	38.2%	27.9%	0.9537	30.2%	37.1%	1.4246

From the above figure, it has been shown that if the FU selection is made by another pass of profiling based filtering, the IPC is better, and more instructions will be issued to power-frugal ALUs. If the FU assumption is used directly as FU choice, the IPC is slightly worse and a bit less than those generated by another pass of profiling, but it is not input vector dependent and the effort of profiling is saved. Anyway, these scheduled codes both show better IPC and RPAIP than the filtering only method.

It can be concluded that the FU selection made with re-ordered code is a better approach for reducing dynamic power consumption for out-of-order issue processors. At the same time, the performance is improved.

6.5 Chapter Conclusion

In this chapter, an instruction scheduling algorithm has been proposed to re-order the instructions in a basic block so as to increase the number of instructions that can be issued to power-frugal FU while still maintaining the best performance. The FU choice assumptions made in the scheduling algorithm can also be directly used as the FU choice for the instructions. The scheduling algorithm is compatible with both in-order and out-of-order issue processors. Simulation results show that this scheduling algorithm provides higher power reduction over the filtering only algorithms and results in performance improvement. The scheduling algorithm is the best approach we propose in this thesis.

Chapter 7 Conclusion

With scaled feature sizes provided by the continuously advancing semiconductor technology, the number of on-chip functions in modern day microprocessors has been steadily increasing. On the other hand, battery industry has been relatively slow in providing power support for the largely increased power dissipation need for modern day microprocessors. Battery-life has been a term that largely restricts the overall performance of a system. Hence, minimizing power dissipation in microprocessors has become an important design consideration.

In this work, we focused on the Functional Units, the parts which actually execute instructions in a processor. Based on the observation that slower FU are more power-frugal, we introduce extra slower FUs (with lower per-execution energy) of a same function to those fast counterparts into a processor. Two instruction filtering algorithms and one scheduling algorithm are proposed to make use of these additional FU to save dynamic power.

The first instruction filtering algorithm is targeting at in-order issue processors. As the issue logic is simple, by analyzing the object codes, the relative issue time of instructions in a basic block can be estimated and instructions whose results are not referred to for a predetermined period of time can be filtered out. These instructions can then be issued to power-frugal FU without harming the performance of the program.

For out-of-order issue processors, such issue time estimation is not easy due to

the complexity of the issue logic. Hence we proposed to use profiling techniques to monitor the actual issue times when a program is running. The issue times of multiple iterations for a single instruction is recorded statistically and further analyzed to find out those instructions whose results are least likely to be referred in a certain amount of time by the downstream instructions. These instructions are then picked out to be issued to power-frugal FU.

Improved power reduction can be achieved if the instruction order can be changed to expose more instructions for power-frugal execution. A scheduling program aiming at both efficient execution (first objective) and more power reduction is proposed. With such a scheduling algorithm, instructions within a basic block are re-ordered and hence better power reduction and execution efficiency are achieved.

Comparing the simulation statistics generated, the scheduling algorithm is the most efficient in both power reduction and execution performance. Simulations show around 40% of all ALU instructions are executed in power-frugal ALUs for the benchmark programs used, which implies a power reduction in ALUs of 20%.

Bibliography

[1] V. De and S. Borkar, “Low power and high performance design challenges in future technologies,” in GLSVLSI '00: *Proceedings of the 10th Great Lakes Symposium on VLSI*, 2000, pp. 1-6

[2] R. Jejurikar, C. Pereira, and R. Gupta, “Leakage aware dynamic voltage scaling for real-time embedded systems,” in DAC '04: *Proceedings of the 41st Annual Conference on Design Automation*, 2004, pp. 275-280

[3] 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

[4] Intel SpeedStep(R) Power Manager White Paper

[5] Amit Agarwal, Hai Li, Kaushik Roy, “DRG-Cache: A Data Retention Gated-Ground Cache for Deep Submicron”, *IEEE Journal of Solid-State Circuits*, Vol. 38, No. 2, Feb 2003.

[6] Chung-Hsing Hsu, Ulrich Kremer and Michael Hsiao, “Compiler-Directed Dynamic Voltage/Frequency Scheduling for Energy Reduction in Microprocessor”, *ISLPED’01, California USA, Aug 6-7 2001*, pp 275-278

[7] Woonseok Kim, Jihong Kim and Sang Lyul Min, A Dynamic Voltage Scaling Algorithm for Dynamic-Priority Hard Real-Time Systems Using Slack Time Analysis”, *Proceedings of the 2002 Design Automation and Test in Europe Conference and Exhibition*.

[8] Ng Kar Sin, “A Low Power Design for Arithmetic and Logic Design”, Master’s

Thesis, 2004

[9] S. Haga, et al, “Dynamic Functional Unit Assignment for Low Power”, *Proceedings of DATE’03*, 2003.

[10] *International Technology Roadmap for Semiconductors*. [Online]. Available at: http://public.itrs.net/

[11] Kaushik Roy, Saibal Mukhopadhyay and Hamid Mahmoodi-Meimand, “Leakage Current Mechanisms and Leakage Reduction Techniques in Deep-Submicrometer CMOS Circuits”, *Contributed Paper in Proceedings of The IEEE, Vol. 91, No.2*, Feb 2003.

[12] R. Pierret, *Semiconductor Device Fundamentals*. Reading, MA: Addison-Wesley, 1996, CH. 6, pp. 235-300.

[13] Y. Taur and T. H. Ning, *Fundamentals of Modern VLSI Devices*. New York: Cambridge Univ. Press, 1998, CH. 2, pp. 94-95.

[14] 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*, A. Chandrakasan, W. Bowhill, and F. Fox, Eds. Piscataway, NJ: IEEE, 2001, CH.3, pp 48-52.

[15] 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 ICCAD '02: *Proceedings of the 2002 IEEE/ACM International Conference on Computer-Aided Design*, 2002, pp. 721-725.

[16] Thomas D. Burd, Trevor A. Pering, Anthony J. Stratakos, and Robert W.

Brodersen, “A Dynamic Voltage Scaled Microprocessor System”, *IEEE Journal of Solid-State Circuits*, Vol. 35, No. 11, November 2000.

[17] 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.

[18] S. Thompson, P. Packan, and M. Bohr, “MOS scaling: Transistor challenges for the 21st century”, *Intel Technol. J., 3rd quarter 1998*.

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

[20] Y. Ye, S. Borkar, and V. De, “New technique for standby leakage reduction in high-performance circuits”, in *Dig. Tech. Papers Symp. VLSI Circuits*, 1998, pp. 40-41.

[21] M. C. Johnson, D. Somasekhar, and K. Roy, “Leakage control with efficient use of transistor stacks in single threshold CMOS”, in *Proc. ACM/IEEE Design Automation Conf.*, 1999, pp. 442-445.

[22] Michael Powell, Se-Hyun Yang, Babak Falsafi, Kaushik Roy, and T.T. Vijaykumar, “Gated-VDD: A Circuit Technique to Reduce Leakage in Deep-Submicron Cache Memories”, *ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED)*, 2000.

[23] N. Sirisantana, L. Wei, and K. Roy, “High-performance low-power CMOS circuits using multiple channel length and multiple oxide thickness”, in *Proc. Int. Conf. Computer Design*, 2000, pp. 227-232.

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

[25] 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 Trans. VLSI Syst.*, Vol. 8, pp. 235-251, June 2000.

[26] S. Tyagi et al., "A 130 nm generation logic technology featuring 70nm transistors, dual Vt transistors and 6 layers of Cu interconnects", in *Dig. Tech. Papers Int. Electron Devices Meeting*, 2000, pp. 567-570.

[27] F. Worm, P. Ienne, P. Thiran, and G. D. Micheli, "An adaptive low-power transmission scheme for on-chip networks," in ISSS '02: *Proceedings of the 15th International Symposium on System Synthesis*, 2002, pp. 92-100.

[28] H. Zhang, V. George, and J. Rabaey, "Low-swing on-chip signaling techniques: E ectiveness and robustness," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 8, pp. 264-272, June 2000.

[29] T. Sakurai, H. Kawaguchi, and T. Kuroda, "Low-power CMOS design through Vth control and low-swing circuits," in ISLPED '97: *Proceedings of the 1997 International Symposium on Low Power Electronics and Design*, 1997, pp. 1-6.

[30] W. Jeong, B. C. Paul, and K. Roy, "Adaptive supply voltage technique for low swing interconnects," in ASP-DAC '04: *Proceedings of the 2004 Conference on Asia South Pacific Design Automation*, 2004, pp. 284-287.

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

[32] A. Narasimhan, M. Kasotiya, and R. Sridhar, “A low-swing differential signaling scheme for on-chip global interconnects”, in VLSID '05: *Proceedings of the 18th International Conference on VLSI Design held jointly with 4th International Conference on Embedded Systems Design (VLSID'05)*, IEEE Computer Society, 2005, pp. 634-639.

[33] Gurindar S. Sohi, “Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers”, in *IEEE Transactions on Computers*, Vol. 39, No. 3, March 1990.

[34] D. Burger and T. Austin. The SimpleScalar Tool Set, Version 3.0. Technical report, Computer Sciences Department, University of Wisconsin-Madison, 1999.

[35] C. Nagendra, et al, “Power-Delay Characteristics of CMOS Adders”,*IEEE Trans. On VLSI Systems*, Vol2, No.3., Sept 1994.

[36] MIPS ECOFF File Format.