

# ABSTRACT

Title of Thesis:       LOW ENERGY WAKE-UP LOGIC

Himabindu Kakaraparthi, Master of Science, 2003

Thesis directed by:   Professor Manoj Franklin  
                          Department of Electrical Engineering

Wake-up logic is responsible for informing instructions in the Window that are waiting to execute, about the availability of their input operands. The conventional method of wake-up consumes a significant percentage of the Instruction Window energy. Reducing the wake-up energy also addresses the Instruction Window hot spot problem caused due to the high power density of the Instruction Window.

In this work, we investigate the energy and power savings of a low complexity scheme that stores the dependence relations between instructions in an array and uses this array to simplify the wake-up. We then present a new wake-up scheme that further reduces the wake-up energy by using a smaller table to store dependence relations and dynamically allocates dependence slots to only those instructions that have dependents in the Window. Our approach leads to savings of up to 50% in wake-up energy and 15% in the Instruction Window power with

a very slight decrease in IPC. Also, both the schemes are more scalable than the conventional wake-up scheme with increasing Instruction Window size and Issue Width.

# LOW ENERGY WAKE-UP LOGIC

by

Himabindu Kakaraparthi

Thesis submitted to the Faculty of the Graduate School of the  
University of Maryland, College Park in partial fulfillment  
of the requirements for the degree of  
Master of Science  
2003

Advisory Committee:

Professor Manoj Franklin, Chair  
Professor Bruce Jacob  
Professor Donald Yeung

© Copyright by

Himabindu Kakaraparthi

2003

## DEDICATION

To my parents - the best I know.

## ACKNOWLEDGMENTS

I want to express sincere gratitude to my advisor, Dr. Manoj Franklin, for all his time, patience and support. I learnt a lot from him. I also want to thank my family and friends for not only making this thesis possible but also enjoyable.

Himabindu Kakaraparthi

# TABLE OF CONTENTS

<b>List of Tables</b>	<b>vii</b>
<b>List of Figures</b>	<b>viii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	2
1.2 Contributions of this thesis . . . . .	3
1.3 Organization of the Thesis . . . . .	5
<b>2 Background</b>	<b>6</b>
2.1 Out-of-Order (OOO) Execution . . . . .	6
2.2 The Instruction/Execution pipeline . . . . .	9
2.3 The Instruction Window . . . . .	10
2.4 Related Work . . . . .	11
<b>3 DL-based Wake-up</b>	<b>15</b>
3.1 Motivation and Outline . . . . .	15
3.2 Theory and Implementation . . . . .	16
3.3 A Comparison with the N-use scheme . . . . .	19

3.4	Impact on Performance (IPC) . . . . .	20
3.5	Impact of Clock Cycle . . . . .	21
3.6	Impact on Area/Space . . . . .	22
3.7	Branch Misprediction/Instruction Squashes . . . . .	23
<b>4</b>	<b>Analysis of Energy Consumption</b>	<b>25</b>
4.1	Energy Consumed in DL-based Wake-up . . . . .	25
4.2	Energy Consumed in Conventional Wake-up . . . . .	29
4.2.1	Tag Drive Energy . . . . .	31
4.2.2	Tag Match Energy . . . . .	32
4.2.3	Match-OR Energy . . . . .	33
4.3	A Comparison of the Energy Consumed in the DL-based Scheme and the Conventional Scheme . . . . .	33
4.4	Effect of Dlist Length on Wake-up Energy . . . . .	35
4.5	Scalability of DL-based Scheme with Instruction Window Size and Issue Width . . . . .	35
<b>5</b>	<b>Need-Based DL (NBDL) Wake-up Scheme</b>	<b>37</b>
5.1	Motivation . . . . .	37
5.2	Parent Instructions . . . . .	39
5.3	Implementation . . . . .	42
5.4	Energy Analysis . . . . .	45
5.5	Effect of NBDL scheme on delay . . . . .	45



5.6	Impact of NBDL scheme on Processor Area . . . . .	47
<b>6</b>	<b>Experimental Analysis</b>	<b>48</b>
6.1	Experimental Setup . . . . .	48
6.1.1	Simulator . . . . .	48
6.1.2	Benchmarks . . . . .	49
6.1.3	Baseline . . . . .	49
6.1.4	Microarchitectural parameters . . . . .	49
6.1.5	Simulation Parameters . . . . .	51
6.2	Results . . . . .	51
6.2.1	Performance Results . . . . .	51
6.2.2	Energy Savings in Wake-up . . . . .	56
6.2.3	Instruction Window Power Savings . . . . .	59
6.2.4	Energy Delay Product . . . . .	61
6.2.5	Scalability with Increasing Window Size and Issue Width . .	64
6.2.6	Performance of DL-based and NBDL Schemes for a Split Window Organization . . . . .	67
6.2.7	Overall Processor Power Savings . . . . .	68
<b>7</b>	<b>Conclusions</b>	<b>69</b>
	<b>Bibliography</b>	<b>74</b>

# LIST OF TABLES

6.1	Microarchitectural parameters . . . . .	50
-----	-----------------------------------------	----

# LIST OF FIGURES

2.1	Instruction Pipeline of an out-of-order execution processor . . . . .	8
3.1	DL-based wake-up . . . . .	17
3.2	Impact of DL-based wake-up on delay . . . . .	22
4.1	Logical view of the Dlist array . . . . .	26
4.2	Conventional wake-up logic . . . . .	29
5.1	Percentage of Parent instructions . . . . .	41
5.2	Need Based DL (NBDL) wake-up . . . . .	43
5.3	Effect of NBDL scheme on the delay of dispatch stage . . . . .	46
6.1	Relative IPC of DL-based scheme. Window size 64, Issue Width 4 .	52
6.2	Performance comparison of DL based and NBDL wake-up scheme. Window size 64, Issue Width 4 . . . . .	54
6.3	Relative wake-up energy of DL-based scheme. Window size 64, Issue Width 4 . . . . .	56
6.4	Wake-up energy comparison of DL-based scheme and NBDL scheme. Window size 64, Issue Width 4 . . . . .	58

6.5	Relative Window power of NBDL scheme. Window size 256, Issue Width 8 . . . . .	60
6.6	Wake-up energy delay product as a function of increase in delay . .	63
6.7	Scalability of wake-up energy of NBDL scheme with Window size and Issue Width . . . . .	65

# Chapter 1

## Introduction

Throughput has always been the primary goal in processor design, especially for general-purpose processors. In order to achieve greater processing power, many techniques such as pipelining, caching, parallel execution (superscalar or VLIW), and out-of-order execution are usually employed. Although these techniques are quite successful in improving the throughput, they increase the complexity of the logic involved. This increase in complexity has two main effects - (1) increase in delay associated with the logic (2) increase in the energy/power consumed by the logic.

The increase in energy consumption is a serious problem. In fact, present day general-purpose processors such as the Alpha 21364 and PowerPC 704 dissipate about 86 and 100 Watt, respectively [1]. Thus, energy has become a significant factor in processor design. This is also true for embedded systems and laptops that run on battery power and hence are required to have low energy consumption in order to be viable. The secondary effect of increased energy

consumption is an increased expense for cooling the chip and for packaging. So, reducing the energy consumption of the processor is a prime concern. A related problem is the high power density of the Instruction Window, which is at the heart of current day out-of-order superscalar processors. The high power density causes a local hot spot. The problems associated with such a hot spot are difficulties in layout and packaging.

## 1.1 Motivation

Wake-up energy is a significant part of the Instruction Window energy consumption. The wake-up logic is responsible for informing waiting instructions in the Window about the availability of their input operands. Reducing the wake-up energy addresses the twin problems of high energy consumption and the Instruction Window hot spot.

An inspection of the conventional method of wake-up shows a remarkable potential for energy savings. This is because the conventional wake-up involves a fully associative search: the availability of an operand is fanned out to all instructions in the Instruction Window. These include instructions that are ready for issue and instructions waiting on other operands. Huang, et al state that, most often, each instruction has only 1 or 2 dependents [2]. This means

that all other instructions in the Instruction Window do not need this information and hence there is a lot of redundancy in the broadcast operation. By eliminating this redundancy, we can save a considerable amount of energy.

## 1.2 Contributions of this thesis

Recently, some alternate schemes have been proposed to reduce the complexity of wake-up [3, 4, 5, 6, 7]. All of these are dependency-based schemes. The schemes maintain tables of dependency relations and index into these tables to find the dependents of instructions and wake them up. Thus, they eliminate the wake-up of redundant instructions and wake up only those that are necessary. However, the energy/power savings for these schemes have not been quantified or analyzed.

In this work, we formulated a wake-up scheme called *DL-based* (Dependence List-based) wake-up scheme that is based on the low complexity dependency-based schemes and investigated the energy/power savings that can be achieved by the same. It falls into the general class of dependency based wake-up schemes and its energy savings can be considered representative of the class. The scheme associates a list of dependent instructions with each instruction in the Window and reads this list to wake up only the necessary instructions when the producer instruction completes execution. We found that significant

savings in energy could be gained with very little reduction in throughput.

Further, we noted that not all instructions in the Instruction Window have dependents and so we need not allot space to all instructions in the Window to store the dependency relations as we did in the DL-based scheme. In order to investigate the possibility of obtaining more energy savings by eliminating this redundancy, we conducted a study on the average number of *Parent* instructions in a program (Parent instructions are instructions in the Instruction Window that have at least one consumer instructions present in the Window).

Leveraging the results of this study, we propose a scheme that further reduces the wake-up energy/power consumption. This scheme stores the dependence relations of only those instructions that have dependents in the Window, i.e. the Parent instructions. This scheme is called the Need Based DL scheme (NBDL scheme), because the slots in the table used to store the dependence information are dynamically allocated based on need. Thus, the size of the table used to store the dependence information can be reduced from its original size in the DL-based scheme, giving rise to further energy savings. Our simulation results show that the NBDL scheme achieves almost as much throughput as the DL-based scheme while using a dependency table that is half the size of that in the DL-based scheme. Also, the NBDL scheme is more scalable than the conventional wake-up logic with increasing Window size and Issue Width.



## 1.3 Organization of the Thesis

The organization of the thesis is as follows. Chapter 2 defines the problems in detail and provides the necessary background. Chapter 3 introduces the dependency based wake-up scheme of low complexity, i.e. the DL-based scheme, and its implementation. Chapter 4 analyzes the wake-up energy of the new scheme and the conventional scheme. Chapter 5 motivates and describes a scheme for achieving more savings in wake-up energy, i.e. the NBDL scheme. Chapter 6 provides the experimental framework and presents the results of the simulations. Chapter 7 presents the conclusions.

## Chapter 2

### Background

This chapter provides the necessary background to understand the problems addressed in this work and the solutions we propose. We shall briefly describe the basic concepts of out-of-order execution, the execution pipeline and the Instruction Window in Sections 2.1 - 2.3. Section 2.4 discusses related work that has been done to reduce the complexity of wake-up.

#### **2.1 Out-of-Order (OOO) Execution**

To improve the processing throughput, pipelining and parallel execution are now used widely. However, the throughput in both cases is fundamentally limited by data dependencies. A data dependency exists between an instruction A and an instruction B if A's output is an input operand for B or vice versa. These dependencies are mainly of two types: 1. Artificial dependencies that arise due to reuse of registers and can hence be avoided; 2. Real dependencies that arise due

to actual data flow between instructions. Artificial dependencies are resolved using renaming. However, the real dependencies are unavoidable and prevent dependent instructions from executing until the producer instruction completes execution. This can cause stalls in the pipeline [8]. In order to maintain a stream of dynamic instructions feeding into the pipeline(s) and to get around this data dependency problem, out-of-order (OOO) execution is used. This primarily involves finding independent instructions further down in the dynamic instruction stream and executing them in a different order from that in the stream, while still maintaining correctness. The Tomasulo approach for OOO execution involves instructions executing out-of-order and writing their results to a Common Data Bus (CDB) from where the results are read by any waiting dependent instructions [9]. Thus, out-of-order execution is used to improve the throughput by avoiding stalls in the pipeline arising from some instructions whose operands are not yet ready.

The increase in throughput is achieved, however, at the cost of increase in complexity. When compared to an in-order processor, the out-of-order processor requires extra logic to check for dependencies between instructions and to resolve artificial dependencies. Also, a buffer is required to store the instructions that have been fetched but not issued for execution due to data dependencies. This buffer is called the Instruction Queue (IQ). Also, a mechanism is required to inform the waiting instructions in the IQ about the fulfillment of their

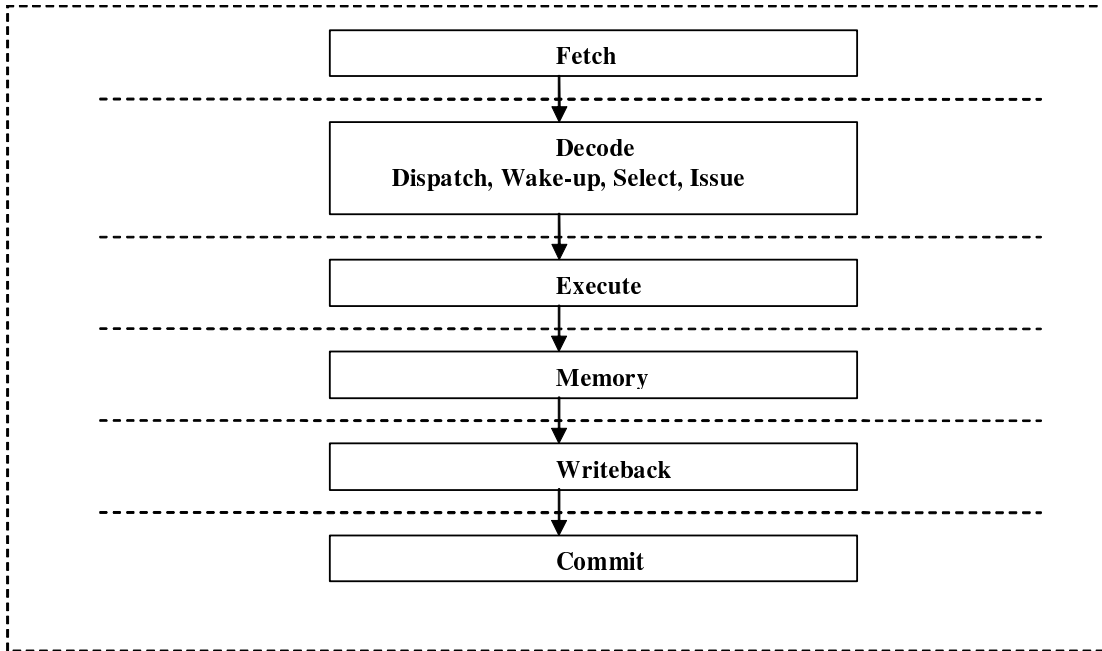


Figure 2.1: Instruction Pipeline of an out-of-order execution processor

dependencies (i.e. the availability of their input operands). This mechanism is called *wake-up*. Thus, on the whole, the complexity of the instruction pipeline increases due to out-of-order issue and execution. This leads to a corresponding increase in the energy/power consumption of the processor.

## 2.2 The Instruction/Execution pipeline

The Instruction pipeline of a typical out-of-order execution, general purpose processor can be broadly divided into 6 key stages. Figure 2.1 illustrates the pipeline. In the *Fetch* stage, instructions<sup>1</sup> from the dynamic instruction stream are fetched from the instruction cache to the fetch buffer. In the *Decode* stage, the instructions are decoded and checked for data dependencies, the dependencies are resolved, and instructions are sent to the IQ. This stage can be subdivided into *Dispatch* and *Issue*. Dispatch involves decoding the instructions and sending them to the IQ after resolving their data dependencies by renaming, whereas Issue selects a subset of the instructions in the IQ and sends them to their respective functional/execution units. This selection is subject to the availability of input operands of the instructions and the availability of execution units. Any instruction whose input operand is unavailable can be bypassed by instructions that are further down in the instruction stream. These waiting instructions are woken up when their input operands become available. Thus, instructions are fetched and dispatched to the IQ in the order of their appearance in the instruction stream but they can be issued to functional units out-of-order. In the *Execute* stage, instructions are executed in the functional units. In the next stage, i.e. the *Memory* stage, instructions that require memory operations, such

---

<sup>1</sup>In this document, we shall refer to *dynamic instructions* as *instructions* unless noted otherwise.

as loads/stores, access the memory hierarchy. In the *Writeback* stage, instructions that have completed their operations write back their results to the common data bus from where the results are collected by the dependents (in the Tomasulo scheme).

Finally, there is the *Commit* stage that is specific to out-of-order execution. This is the stage where the outputs of the instructions that have completed execution are committed. This has to be done in-order (dispatch order) to support precise interrupts in the face of out-of-order execution. “A pipeline is said to support precise interrupts if the saved process state (i.e. program counter, register file, etc) is consistent with the sequential architectural model” [10]. As instructions are executed out-of-order, additional help is required to maintain the original sequence of instructions for commit. A structure called the *Re-Order Buffer* (ROB) is used to maintain all instructions in dispatch order. The ROB is written to at dispatch time as instructions dispatch (in-order) and contains the descriptors or identifiers (IDs) of the instructions.

## 2.3 The Instruction Window

The ROB and the IQ can be combined to a single storage unit called the Register Update Unit (RUU) or Instruction Window in general [10, 11]. Instructions are

sent to the Window after the dispatch stage and stay there until the end of the commit stage. The Window is accessed to write instructions at dispatch time, to select the instructions for execution and to write operands to the respective functional units at issue time, to write results to the dependents in the Window and wake them up (broadcast operation) at writeback time. Thus, in a single cycle, the Window is read/written several times corresponding to instructions in different stages of the pipeline. This causes the power density of the Instruction Window to be very high leading to a Window *hot spot* problem. Ponomarev, et al estimated that more than 27% of the total power expended within a processor is dissipated in the Window [12]. A significant part of the Instruction Window energy is comprised of instruction wake-up energy and hence any reduction in the wake-up energy correspondingly decreases the power density of the window.

## 2.4 Related Work

Recently many dependency-based schemes have been proposed to reduce the complexity of wake-up [6, 7, 4, 3]. Dependency relations can be modeled as producer-consumer relations. Instructions that depend on other instructions can be considered as the consumers of a value created by the producers. In conventional wake-up the onus is on the consumers: they store the dependency relation and listen to the results being broadcasted each cycle to find out when

their input operands become available. Dependency based schemes move this responsibility onto the producers. In these schemes, tables of dependency relations are maintained and indexed into to find the consumers of each value. When an instruction completes execution, it finds out who the dependents are and explicitly wakes them up. This helps in reducing the complexity of wake-up as it reduces the number of instructions considered for wake-up. A few such low complexity, dependency-based schemes are discussed next.

Canal and Gonzalez have proposed a low complexity issue logic scheme that keeps track of dependent instructions using a table [6, 7]. Their scheme partitions the Instruction Queue into 2 parts: The N use queue and the Separate Ready Queue (SRQ). The N-use queue has an entry for every physical register in the architecture and each entry holds the first N dependents of the corresponding physical register. The SRQ holds instructions that have all of their input operands available and are ready to execute. Instructions are issued in program order from the SRQ. When an instruction completes execution, it indexes into the N-use table entry corresponding to its output register and wakes up the instructions in that entry. These woken up instructions are then moved to the SRQ from where they are issued in order.

Soner Onder and Rajiv Gupta present a scheme that also sets up lists of instructions to be woken up by producer instructions [4]. Each instruction is



allowed to wake up two dependent instructions and two siblings each for either of its input operands. The scheme proposed by Sato, et al stores dependency information in the form of relations between instructions in a table and is closest to our low complexity DL-based wake-up scheme [3].

All these studies only look at the performance effects of the reduced complexity of wake-up logic. They hint that reduced complexity leads to reduced energy and power too but have not quantified or analyzed it. The DL-based scheme that we formulated falls into the general class of dependency-based schemes and hence the energy/power savings obtained with our scheme can be considered to be representative of that achieved by other such schemes.

Some other schemes have been proposed for low power/energy wake-up logic and Instruction Window [2, 1, 13, 14, 15]. These schemes are orthogonal to our study and some of them can be combined with our approach to improve power/energy savings. Huang, et al propose indexing to selectively enable the comparator of only those instructions that are being woken up [2]. When multiple instructions are being woken up, the scheme reverts to conventional wake-up logic. Thus, it is a hybrid scheme. Folegnani and Gonzalez design an issue logic that saves energy by gating out empty Window entries and ready instructions for wake-up [1]. Also, they dynamically scale the Instruction Queue size to suit the program behavior thereby saving energy. Kucuk, et al propose a

scheme that dynamically resizes the ROB based on occupancy statistics [13].

Buyuktosunoglu, et al present an adaptive Issue Queue design that varies the Issue Queue size to match workload demands [14, 15]. Further, they combine this with fetch gating that gates the fetch mechanism whenever there is a flow mismatch between instructions fetch and instruction completion.

## Chapter 3

### DL-based Wake-up

This chapter presents and explains a low complexity, dependence based wake-up scheme called the Dependency List (DL) based wake-up scheme. Section 3.1 motivates the discussion and outlines the scheme. Section 3.2 describes the DL-based wake-up scheme in detail and Section 3.4 analyzes the performance of the DL-based scheme when compared to conventional, associative wake-up.

#### **3.1 Motivation and Outline**

As described in Chapter 1, the conventional wake-up involves considerable redundancy in the broadcast process. However, if we eliminate the broadcast to reduce the energy consumption, the dependency information has to be stored elsewhere. This can be done by associating a list of dependent instructions with each instruction in the Instruction Window. Once an instruction completes execution, it can read this list and wake up only those instructions that are in its

dependency list. The DL-based scheme that we formulate is based on this idea.

## 3.2 Theory and Implementation

The DL-scheme associates each instruction with a list of dependent instruction identifiers. These lists of instruction identifiers (IDs) are called Dependence lists (Dlists). A new set of buffers called the *Dlist array* is added to the microarchitecture. The instructions are stored in the Window and the corresponding Dlists are stored in the Dlist array. Thus, each instruction in the Window is statically associated with an entry in the Dlist array. We use the Window slot number of an instruction as its identifier. At decode/dispatch time, each instruction accesses the Register Availability Table (RAT) to find out if its input operands are ready. If they are, then the instruction is dispatched to the Instruction Window straight away as usual. If any input operand is not available at this time, the instruction is still dispatched, but additionally, the instruction is required to register with its producer instruction as a dependent. The RAT provides the details of this operand's producer, i.e., the Window slot number of the producer instruction. Using this index provided by the RAT, the dependent instructions index into the Dlist array and enter their ID in the producer's Dlist. If an instruction has two non-ready input operands, it writes its ID to the Dlists of both the producers. On completion of execution, the Dlist of the instruction is

read and the corresponding instructions are set to ready, i.e., they are woken up. The Dlist array look-up can be overlapped with the reading of operands from the Window to the functional units or the instruction execution itself.

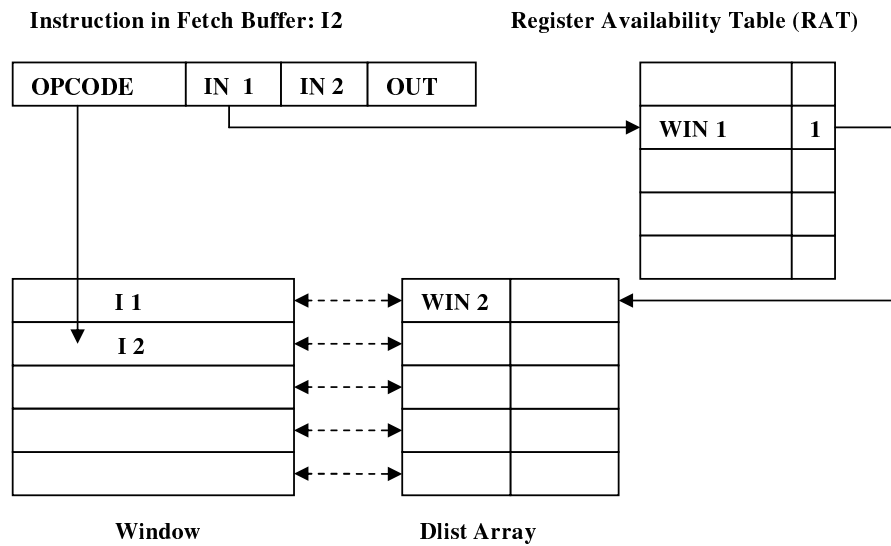


Figure 3.1: DL-based wake-up

Figure 3.1 illustrates the DL-based wake-up scheme. The figure shows how each entry in the Window is statically associated with an entry in the Dlist array. Also, note that each row in the Dlist array has two columns, i.e., each row can

hold the IDs of two dependent instructions. This parameter, i.e. the number of columns in the array, is called the *Dlist length*. In Figure 3.1, the instruction I1 is currently in the Window slot number 1, i.e. WIN 1. It produces an output IN1 that is used by the succeeding instruction I2. Thus, instruction I2 depends on instruction I1 through the operand IN1. At dispatch time, instruction I2 queries the RAT for its input operands. The instruction indexes into the RAT using IN1 and finds that IN1 is not available and that I1 is producing it. I2 is dispatched to the Window slot WIN 2. Simultaneously, it obtains the Window slot number of the producer instruction (WIN 1). I2 now uses this slot number to index into the Dlist array and writes its ID, i.e. WIN 2, to the Dlist array as depicted in the figure. Thus, the Window slot number of an instruction is used as its ID when making entries in the producer's Dlist. When I1 completes execution, it indexes into the associated slot in the Dlist array and wakes up the instructions waiting on it, i.e. I2.

The number of instructions within the Dlist of each instruction (Dlist length) determines the number of dependent instructions that can be woken up by a producer instruction. If the number of dependent instructions that arrive in the Window before the producer completes execution is greater than Dlist length, instruction dispatch is stalled. This is because dispatch has to be carried out in order and an instruction cannot be dispatched unless its input operands are already available or its corresponding dependence relation has been stored

somewhere. So, if the producer's Dlist is full, instruction dispatch can proceed only when the producer instruction completes execution and writes the result to the register file from where it is read by the dependent instructions.

### **3.3 A Comparison with the N-use scheme**

Canal and Gonzalez's scheme is similar to the DL-based scheme. However, their scheme is more complicated and there are significant changes to the Issue stage architecture. As described in Section 2.4, their scheme partitions the Instruction Queue into two parts and instructions are issued from only the SRQ. Thus, this scheme consumes a significant amount of energy in moving instructions from one queue to other. Also, the performance degrades because instructions are constrained to issue only from the SRQ and the SRQ might overflow in which case dispatch needs to be stalled. Also, by partitioning the IQ into two parts, the use of a Register Update Unit (RUU) in this architecture is disallowed. This is because instructions need to be maintained in dispatch order in the RUU and not moved after that.

### 3.4 Impact on Performance (IPC)

In this subsection, we look at how our DL-based wake-up scheme affects the performance (measured in Instructions Per Cycle or IPC). A significant reduction in the IPC would render a scheme undesirable. The loss of IPC can be attributed to the dispatch stalls that occur when there is no free space in the producers' Dlists. As we increase the size of the Dlists we can accommodate more instructions in the Dlist and hence move closer to the ideal/ conventional case, which has a Dlist length of infinity. However, increasing the Dlist length increases the energy consumed because of the increase in the size of the Dlist array. Thus, this parameter represents a trade-off between energy and performance.

An important observation in this regard is that as the Window size increases, we are able to accommodate more instructions in the Window and hence look deeper into the instruction stream to exploit more parallelism in the program. This implies that more dependent instructions may be accommodated in the Window in case of the conventional wake-up scheme. However, our DL-based scheme is unable to use this expanded Window as well because of the limited number of Dlist slots per instructions. Dispatch is stalled when there is no space in the producer instructions' Dlists even though there is space in the Window to accommodate these instructions. Hence the increase in IPC with an increased Window size is slower in case of DL-based wake-up when compared to the case of



conventional wake-up. This implies that the relative IPC of the DL-based scheme may decrease slightly with increasing Window size.

### 3.5 Impact of Clock Cycle

Since the DL-based scheme introduces a new piece of logic in the dispatch stage, it affects the delay of this stage. Any change in the delay of any stage of the pipeline impacts the clock cycle. Hence, it is important to study the effect of the scheme on the delay of the dispatch stage. As discussed in Section 3.2, the Dlist array update takes place after the RAT look-up in the dispatch stage, and before the issue stage. Thus, the update lies on the critical path of the pipeline.

Figure 3.2 shows how the Dlist of a producer can be updated in parallel with writing the consumer instruction to the Window. Since each entry of the Window is statically mapped to an entry in the Dlist, the Dlist update involves just indexing into the Dlist array and writing to it. The only excess delay of such an update in comparison to a Window write would be the delay involved in decoding an index in the array. Hence, we expect that the impact of the DL-based wake-up scheme on the delay of the dispatch stage is not significant.

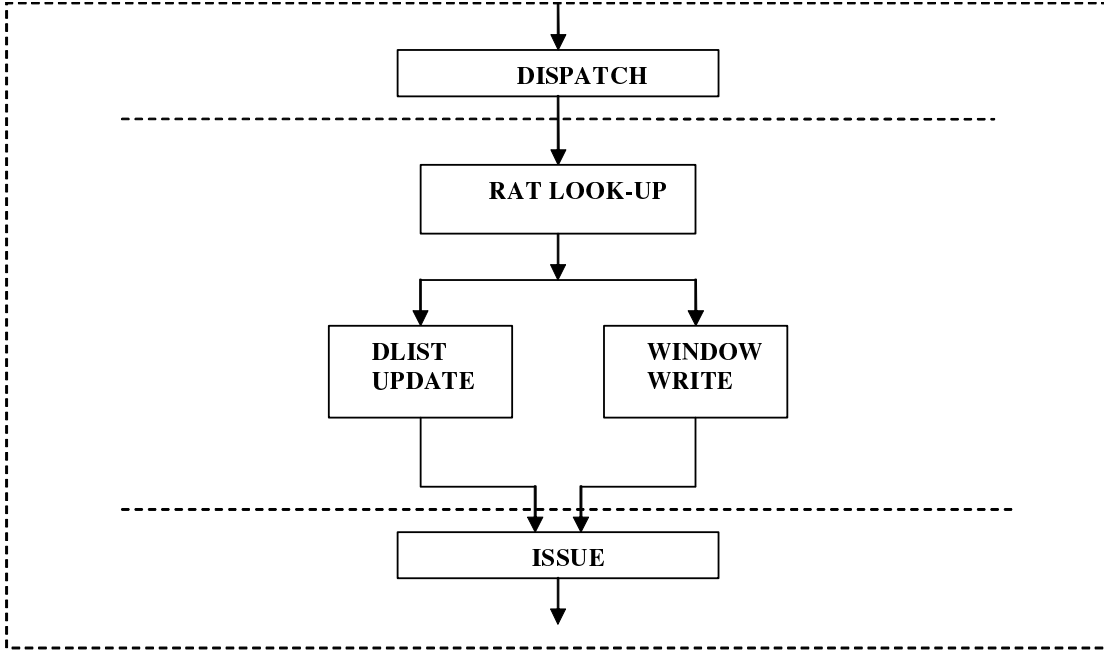


Figure 3.2: Impact of DL-based wake-up on delay

### 3.6 Impact on Area/Space

The DL-based wake-up scheme introduces a new array (Dlist array) into the microarchitecture. However, we think that this will not increase the processor area. This is because the DL-based scheme merely moves the storage of the dependency relations from the consumers to the producers. Thus, the tags which were stored in the Window in the conventional scheme are now stored in the Dlist array in the DL-based scheme. The decoder logic of the Dlist array is the only additional space consumer. On the other hand, a significant amount of area is saved since the Window is no longer a CAM array and all the comparators are eliminated.

### 3.7 Branch Misprediction/Instruction Squashes

So far, we have discussed the DL based wake-up assuming that there are no instruction squashes. In case of instruction squashes due to branch mispredictions or interrupts, additional mechanisms are required to maintain the accuracy of wake-up while using the DL-based wake-up. This is because, if a consumer instruction is squashed and its Window entry is filled with another independent instruction, the producer of the squashed instruction might wake-up an instruction incorrectly. To prevent this scenario, the Dlist of the squashed instruction's producer needs to be cleared to reflect the instruction squash. This can be implemented in a few different ways depending on whether space or delay is the primary constraint.

One way would be to use the RAT to find the producers of the squashed instruction and then clear their Dlists. The RAT is typically backed up to the last consistent copy in case of an instruction squash. Using this RAT, the producers of the squashed instruction can be found out and their Dlists can be cleared. However, this will lead to a significant increase in delay of dispatch stage due to the serial process. A better alternative would be to maintain the producers' tags along with each instruction in the Window and use these to directly index into the corresponding Dlists to clear the entries. This maintains the delay characteristics and slightly increases the space required.

Another implementation that trades off space for delay involves making a copy of

the Dlist array whenever a branch instruction is handled. Whenever, a misprediction occurs, the Dlist array is simply backed up to the last consistent copy. This will eliminate incorrect wake-ups. This implementation is similar to that of the RAT. Because branch misprediction now requires only a change in the base pointer to the Dlist array, the delay involved is insignificant. However, there is an increase in the space consumption, as multiple copies of the Dlist array are now required to be stored in the microarchitecture. An important point to note is that instruction squashes do not impact the wake-up energy savings as such because the process of wake-up itself remains the same. The overall Window energy would include the energy involved in bringing the Dlist array to a consistent state. However, in this thesis we have not quantified this additional energy consumed for instruction squashes.

## Chapter 4

# Analysis of Energy Consumption

In this chapter, we present a detailed analysis of the individual components of the energy consumed in the case of the DL-based wake-up and the conventional wake-up in Section 4.1 and 4.2. A comparison between the energy consumption of these two schemes is presented in Section 4.3. Section 4.4 deals with the effect of the parameter Dlist length on the energy consumption of the DL-based scheme and Section 4.5 studies the scalability of the schemes with Instruction Window size and Issue Width.

### **4.1 Energy Consumed in DL-based Wake-up**

A logical view of the Dlist array can be given as follows: The array consists of word lines and bit lines running across the array. The individual bits in a word line are stored in transistors. The bit lines run perpendicular to the word lines. Figure 4.1 gives the logical view of the Dlist array.

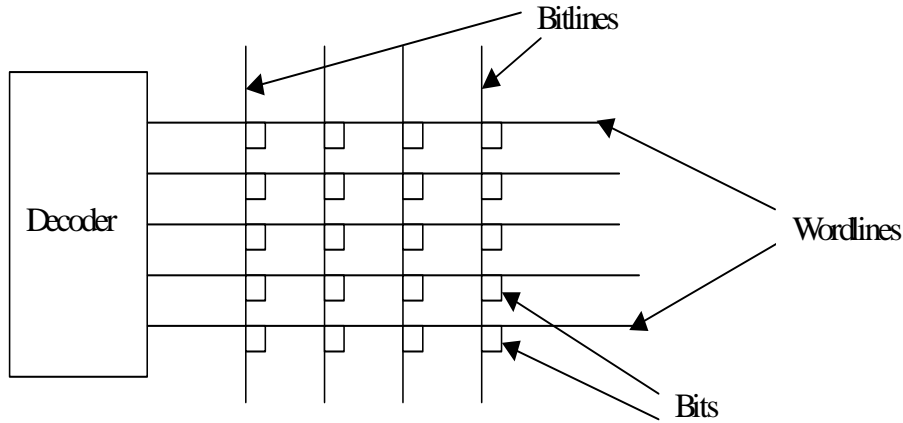


Figure 4.1: Logical view of the Dlist array

In the DL-based scheme, wake-up takes place as follows: the Window slot number of the completing instruction is used to index into the Dlist array and the corresponding entry is read. The ready bits of the instructions in the entry are then set to 1. Thus, this is a table look-up, i.e. a RAM based scheme. Each table access consists of the following steps.

- (1) The index in the array is supplied to the decoder that decodes it into the corresponding entry or word in the array.
- (2) The required word line is then raised.
- (3) All the bit lines are raised and the data is read from/written to the bit lines.

Correspondingly, the energy consumed in a Dlist array access can be divided into three main components: decoder energy, word line energy, and bit line

energy.

$$Energy_{DL} = Energy_{DLdecoder} + Energy_{DLwordline} + Energy_{DLbitline} \quad (4.1)$$

The decoder energy is proportional to the number of rows in the structure. However, a component of the decoder energy is also proportional to the number of bits required to decode an address in the array which is  $\log DLsize$ . Thus, for the Dlist array, the decoder energy can be approximated as follows.

$$Energy_{DLdecoder} \propto DLsize(1 + \log DLsize) \quad (4.2)$$

where  $DLsize$  is the number of entries in the Dlist array.

The word line energy is proportional to the word line length. The length of each word line is given as follows:

$$Wordlinelength = cols \times (CellWidth + ports \times BSpacing) \quad (4.3)$$

where  $cols$  is the number of columns in the array,  $CellWidth$  is the width of each cell in the array, and  $BSpacing$  is the spacing between the bit lines. The Dlist array has  $2 \cdot IW$  write ports and  $IW$  read ports, where  $IW$  is the Issue Width.

This is because each instruction may write to the Dlist array twice, once for each dependent operand, and read the Dlist array once. Therefore,

$$Wordlinelength_{DL} = Dlistlength \times \log WINSIZE (CellWidth + 3 \times IW \times BSpacing) \quad (4.4)$$

where  $WINSIZE$  is the number of rows in the Window. Putting the above

equations together,

$$Energy_{DLwordline} \propto Dlistlength \times \log WINSIZE \times IW \quad (4.5)$$

Similarly, the bit line energy is proportional to the bit line length.

$$Bitlinelength = rows \times (CellHeight + ports \times WSpacing) \quad (4.6)$$

where rows is the number of rows in the array, *CellHeight* is the height of each cell in the array and *WSpacing* is the spacing between the word lines. For the Dlist array, the bit line length is given by

$$Bitlinelength_{DL} = DLsize \times (CellHeight + 3 \times IW \times WSpacing) \quad (4.7)$$

The bit line energy is the energy required to raise *all* the bit lines in the array.

Therefore,

$$Energy_{DLbitline} \propto Dlistlength \times \log WINSIZE \times DLsize \times IW \quad (4.8)$$

As we see from Equation 4.8, the bit line energy is the largest contributor as it includes the energy to drive all the bit lines spanning the Window.

We make two observations from this analysis:

1. From Equation 4.2, we conclude that the decoder energy increases non-linearly with the Dlist array size.
2. From Equation 4.8, we infer that bit line energy increases linearly with the Dlist array size and non-linearly with the Window size.



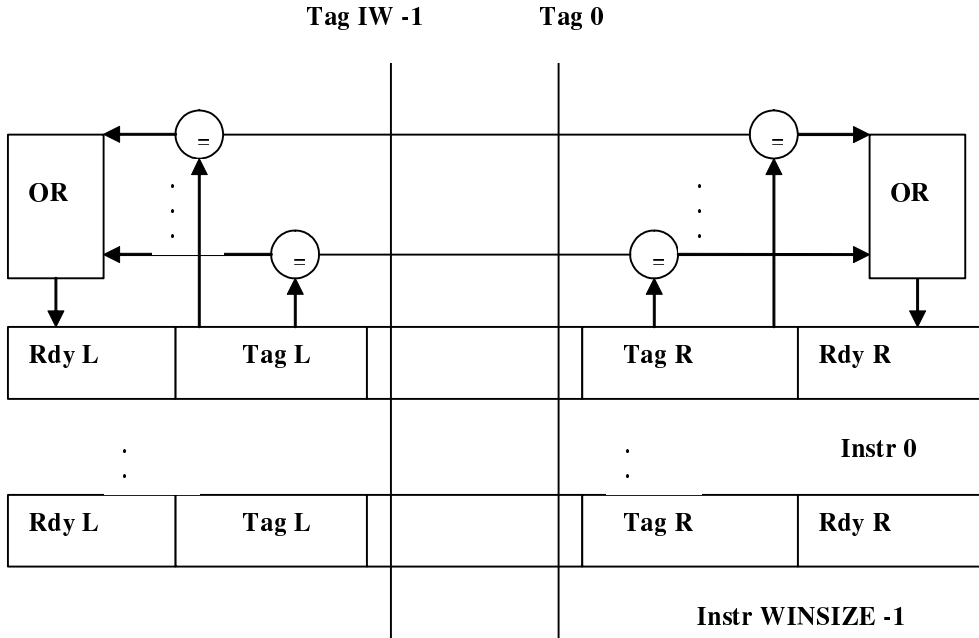


Figure 4.2: Conventional wake-up logic

## 4.2 Energy Consumed in Conventional Wake-up

In conventional superscalars, the Instruction Window is designed as a Content Addressible Memory (CAM). Each entry in the Window holds one instruction. The tags (identifiers) of both the input operands of the instruction are stored in the Window [5]. Also, each instruction has two ready bits - one for each of its operands (Rdy L, Rdy R). These bits indicate the availability of the corresponding input operand of the instruction. When both of them are set to ready, the instruction is ready to execute and is considered for issue by the select logic.

Figure 4.2 illustrates the conventional wake-up logic. Tag lines run across the length of Window. These lines are used to broadcast the tags of instructions completing execution. At any time, there can be at most Issue Width (IW) instructions completing execution where Issue Width is the maximum number of instructions that can be allocated execution units in any cycle. Each instruction completing execution broadcasts its tag to the Window. Hence there are IW tags and  $IW * \text{tag size}$  number of taglines. Along with these lines, there are match lines running across the width of the Window. There is one match line per broadcasted tag per instruction operand in the Window. When the tags are broadcast, each operand's tag is compared with all the broadcasted tags in parallel. If there is a mis-match, the corresponding match line is pulled low. All the match lines for each operand are then OR-ed together. Thus, if any of the broadcasted tag matches with a tag an operand is waiting on, the output of the OR block is 1. This is used to set the corresponding ready bit of the operand. When both the ready bits are set to 1, the instruction is ready to execute. This is how wake-up is achieved.

The conventional wake-up is performed in three steps:

1. The tag is broadcasted to the entire Window.
2. The tag is compared to the stored tags and the match line is pulled low if there is a mis-match.
3. All the match lines for each stored tag are OR-ed together to check for match.

Corresponding to the three steps, the energy consumed can be divided into three parts: tag drive energy, tag match energy and match-OR energy.

$$Energy_{Conv} = Energy_{tagdrive} + Energy_{tagmatch} + Energy_{matchOR} \quad (4.9)$$

### 4.2.1 Tag Drive Energy

The tag drive process can be further split into two steps:

- (1) drive the tag onto the tag lines
- (2) write the tag to each entry in the Window.

Thus, tag drive energy can be expressed as:

$$Energy_{tagdrive} = Energy_{drive} + Energy_{write} \quad (4.10)$$

The drive energy is proportional to the overall length of the tag lines. The length of each tag line is given by

$$taglinelength = rows \times (CellHeight + ports \times Mspacing) \quad (4.11)$$

where  $Mspacing$  is the spacing between the match lines. For the Window

$$taglinelength_{CAM} = WINSIZE \times (CellHeight + IW \times Mspacing) \quad (4.12)$$

There is one tag line for each bit in the tag. Therefore,

$$Energy_{drive} \propto WINSIZE \times IW \times \log WINSIZE \quad (4.13)$$

The tag write is similar to a write in an array as analyzed in Section 4.1.

However, in this case it is a write to *all* entries in the Window, whereas in the

case of the Dlist array access, it is a write to a *single entry*. Hence, all the word lines have to be raised in the case of a tag write. The bit line energy and the word line energy can be calculated using the formulae in Section 4.1. Therefore,

$$Energy_{write} = Energy_{CAMbitline} + Energy_{CAMwordline} \quad (4.14)$$

$$Bitlinelength_{CAM} = WINSIZE \times (CellHeight + IW \times Mspacing) \quad (4.15)$$

$$Wordlinelength_{CAM} = \log WINSIZE \times (CellWidth + IW \times Tspacing) \quad (4.16)$$

where  $Tspacing$  is the spacing between the tag lines. Using the above equations,

$$Energy_{CAMbitline} \propto WINSIZE \times \log WINSIZE \times IW \quad (4.17)$$

$$Energy_{CAMwordline} \propto WINSIZE \times \log WINSIZE \times IW \quad (4.18)$$

Thus, from Equations 4.10, 4.13, 4.14, 4.16 and 4.17,

$$Energy_{tagdrive} \propto WINSIZE \times \log WINSIZE \times IW \quad (4.19)$$

## 4.2.2 Tag Match Energy

Tag match energy is the energy consumed in driving the match lines. This is proportional to the length of the match line length and can be computed similar to wordline power above.

$$Matchlinelength_{CAM} = \log WINSIZE \times (CellWidth + IW \times Tspacing) \quad (4.20)$$

$$Energy_{tagmatch} \propto WINSIZE \times \log WINSIZE \times IW \quad (4.21)$$

### 4.2.3 Match-OR Energy

The match-OR energy is the energy to required to OR all the match lines to check if there was a match. This depends on the number of inputs to the OR block which is equal to the Issue Width. This energy, however, is quite less when compared to tag drive and tag match energy.

## 4.3 A Comparison of the Energy Consumed in the DL-based Scheme and the Conventional Scheme

In Section 4.2.1, we described how a tag write energy can be modeled to a table access where all the entries are being written. The Dlist array and the Window have the same number of rows. If the Dlistlength is set to 1, then both the structures have the same granularity of access. Thus, the tag write energy can be directly compared to the energy consumed in one Dlist access. Since tag write involves writing to all entries in the window, energy consumed for it is much higher than that consumed for a single Dlist access.

$$Energy_{write} > Energy_{DL} \quad (4.22)$$

The other component of tag drive energy is the drive energy that is consumed in writing the tag on to the tag lines in the window. This is comparable to the

bitline energy of a Dlist access, and hence to the total energy of a Dlist access (since bit line energy constitutes most of the access energy).

$$Energy_{drive} \approx Energy_{DL} \quad (4.23)$$

Thus,

$$Energy_{tagdrive} = Energy_{write} + Energy_{drive} > 2 \times Energy_{DL} \quad (4.24)$$

$$\Rightarrow Energy_{Conv} > 2 \times Energy_{DL} \quad (4.25)$$

As most instructions have at least one operand ready at the time of dispatch, they access the Dlist at most twice. Because the conventional wake-up consumes more power than two Dlist accesses, energy can be saved by using the DL-based scheme instead of the conventional scheme.

It must be noted that an increase in execution time degrades the throughput and also increases the energy consumption. This increase in energy consumption is because all the pipeline units now run for a longer time. The units can be selectively shut down when they are idle. This is called *clock gating*. However clock gating is not perfect and so the units consume a small amount of power even when they are idle. Therefore, in this work, we aim to reduce both the wake-up energy and power while retaining most of the performance.

## 4.4 Effect of Dlist Length on Wake-up Energy

The comparison we made above is based on a microarchitecture that has a Dlist length set to 1, i.e. the DL-1 scheme. Here, we analyze how the wake-up energy of the DL-based scheme varies with the Dlist length. As the Dlist length is increased, the number of entries in each row, i.e. the number of columns, of the Dlist array increases. This increases the number of bit lines in the Dlist array, with a proportional increase in the bit line energy. As the conventional wake-up scheme is independent of the Dlist length, the savings in energy with respect to the conventional scheme decrease with increasing Dlist length.

## 4.5 Scalability of DL-based Scheme with Instruction Window Size and Issue Width

During the Select phase, all the instructions in the Instruction Window that haven't completed execution are searched to find ready instructions that can be issued to the functional units to execute. So, increasing the Window size increases the depth of the dynamic instruction stream that is searched to find instructions that can be executed in parallel. This increases the chances of finding independent instructions that can be executed in parallel and hence Instruction Level Parallelism (ILP) increases. Also, throughput can be increased by increasing the number of instructions allowed to execute in parallel. Typically

the Issue Width that can be sustained is limited by the ILP that can be found in the program. Hence, the Window size and the Issue width are typically increased in tandem to achieve higher performance.

However, as the Window size and Issue Width increase, the tag line length, the match line length, and the size of tag increase. There is a non linear dependence of the tag drive energy and tag match energy on both the Window size and the Issue Width. Thus, conventional wake-up energy increases rapidly when these parameters are increased [16]. On the other hand, the decoder energy per access in the DL-based scheme is independent of the Issue Width. The dependence of bit line energy and word line energy on Issue Width is similar to the tag drive energy and tag match energy. So, both the conventional and the Dlist scheme scale almost equally with Issue Width. However, there is a difference in their scalability with Window size. As we described in Section 4.2,

$$Energy_{Conv} \propto WINSIZE \times \log WINSIZE \quad (4.26)$$

This is not true for the Dlist energy as the word line energy is proportional to  $\log WINSIZE$  and a component of the decoder energy is proportional to  $WINSIZE$  only. Because of this, overall wake-up energy in the DL-based scheme varies slower than that in the conventional scheme. Thus, the DL-based wake-up scheme is more scalable and the energy savings increase with the Window size and the Issue Width.



## Chapter 5

### Need-Based DL (NBDL) Wake-up Scheme

Having studied the DL-based scheme, we now present a scheme with potential improved energy savings in wake-up. This scheme is called Need Based DL (NBDL) wake-up scheme. The motivation for this scheme is presented in Section 5.1. Section 5.2 discusses a study to find the possibility of further wake-up energy savings that forms the basis for the NBDL scheme. Finally, Section 5.3 outlines the scheme and its implementation with the help of an illustration.

#### **5.1 Motivation**

In Chapter 3 we presented the DL-based scheme, which reduces the wake-up energy consumption while still keeping the overall IPC close to conventional. However, it is to be noted that a considerable amount of energy is still being consumed in the DL-based wake-up scheme for each Dlist array access. The main

reason is the size of the Dlist array. It has as many entries as the RUU and hence a lot of energy is consumed in decoding the address and in driving the bit lines. The long bit lines that run through the entire length of the array are undesirable from both energy and delay points of view [16].

Based on our observations of Section 4.1, we know that one way of reducing both the decoder energy and bitline energy is by reducing the number of rows in the array structure, i.e. the number of entries in the Dlist array. Currently this is set to the Window size. This is because the DL-based scheme implicitly assumes that an instruction can be dependent on any other instruction in the window, i.e., all the instructions in the Window could be producing values that are used by other instructions in the Window. So, one way of reducing the energy is reducing the Dlist array size. However, blindly reducing the Dlist array size could have a negative impact on the performance. One way would be to construct a Dlist array of only  $n$  entries, where  $n$  is size to which the Dlist array is to be scaled, and to allot all the available slots to the instructions in a First Come First Served (FCFS) order. Since an instruction cannot be dispatched unless its dependence relation has been stored, this would lead to dispatch stalls after the first  $n$  instructions. Therefore, though this scheme would be simple to implement, it would decrease the processing throughput. Essentially, it would have a similar effect as reducing the Window size. A better, and more efficient, scheme would be to allocate the slots in the Dlist array to only instructions that need them. To

evaluate the possible energy savings from such a scheme, we conduct the following study.

## 5.2 Parent Instructions

Parent instructions can be defined as un-issued producer instructions within the Instruction Window that have consumers, also within the Window. Consumer instructions arriving at the Window after their producers complete execution can read the outputs from the register file or could have the outputs dynamically forwarded to them. In the case of Parent instructions, both producer and consumer instructions co-exist in the Window and the producer instructions have not completed execution. An important observation is that only these Parent instructions need slots in the Dlist array to store dependency relations. Therefore, it is sufficient to assign slots in the Dlist array table to these instructions. This implies that the Dlist array size needs to be equal to the number of Parent instructions in the application, on an average. Hence, the required Dlist array size needs to be big enough to accommodate the average number of Parent instructions.

Another point to note here is that, as the Window size increases, the number of dependent instructions in the Instruction Window increases. This is because

we now look deeper into the dynamic instruction stream and hence the possibility that the instructions dependent on a particular instruction entering the Window while the producer instruction is waiting to execute or is executing is higher. Thus, the number of Parent instructions in a benchmark is a function of the Window size.

The savings obtained by restricting the Dlist array size to the number of Parent instructions in the Window on an average will be considerable only if there are a significant amount of non-Parent instructions. Examples of non-Parent instructions are instructions like stores, branches, dead instructions, and instructions that are separated from their consumers by a large distance.

In order to investigate the possible savings by eliminating excess Dlist slots, we first collected data on the average number of Parent instructions in the Instruction Window at any time for 8 benchmarks in the Spec2000 benchmark suite. The Window sizes were varied and accordingly the Issue Widths were also varied. The results are shown in Figure 5.1. On an average, we found that around 60% of the instructions in the Window at any time are producer instructions. This implies that 40% of the instructions in the Window are non-Parent instructions and hence do not need Dlist array slots. This implies that the Dlist array size can be made to be 60% of the Window size. These results were leveraged to design a new wake-up scheme to gain significantly higher energy savings over the DL-based scheme described in Chapter 3.

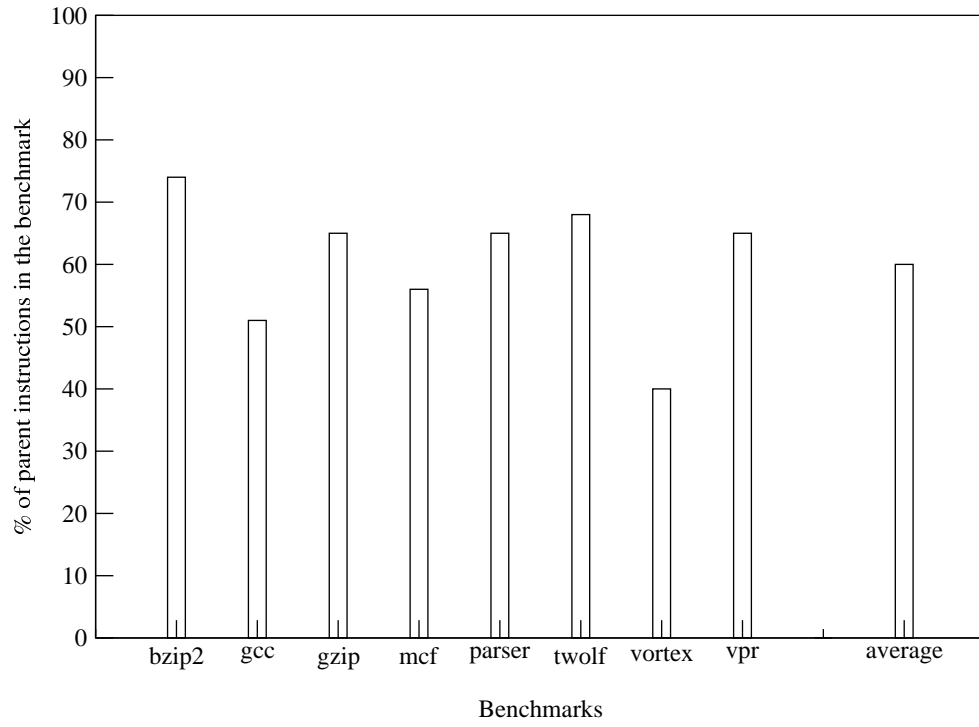


Figure 5.1: Percentage of Parent instructions

## 5.3 Implementation

The NBDL scheme is similar to the DL-based scheme described before since the dependence relations are again stored in an array (Dlist array). However there are some changes. The Dlist array is no longer as big as the Window. Rather, the average number of Parent instructions in the Window for that particular Window size is used to arrive at a good size for the Dlist array. (The average number of Parent instructions over a general benchmark suite can be computed by profiling.) Also, the Dlist array and the Window do not have a static relationship now. Instead, the Dlist array entries are allocated to the instructions in the Window dynamically, based on need.

The NBDL scheme can be described as follows: at dispatch time, the RAT is queried to see if the instruction's input operands are available. If they are, the instruction is dispatched to the Window. The RAT also contains one bit to indicate if the instruction producing that particular operand has a Dlist entry or not. If the operands of any instruction are not ready and the producer instruction already has a Dlist entry, then the dependent makes an entry in this particular Dlist slot. If the producer is not ready and has no Dlist slot, then the allocation logic assigns a Dlist slot to it subject to availability of slots, and the process of storing the ID of the dependent in this slot follows as before. Thus, Dlist slots are allotted dynamically based on need. When an instruction completes execution,

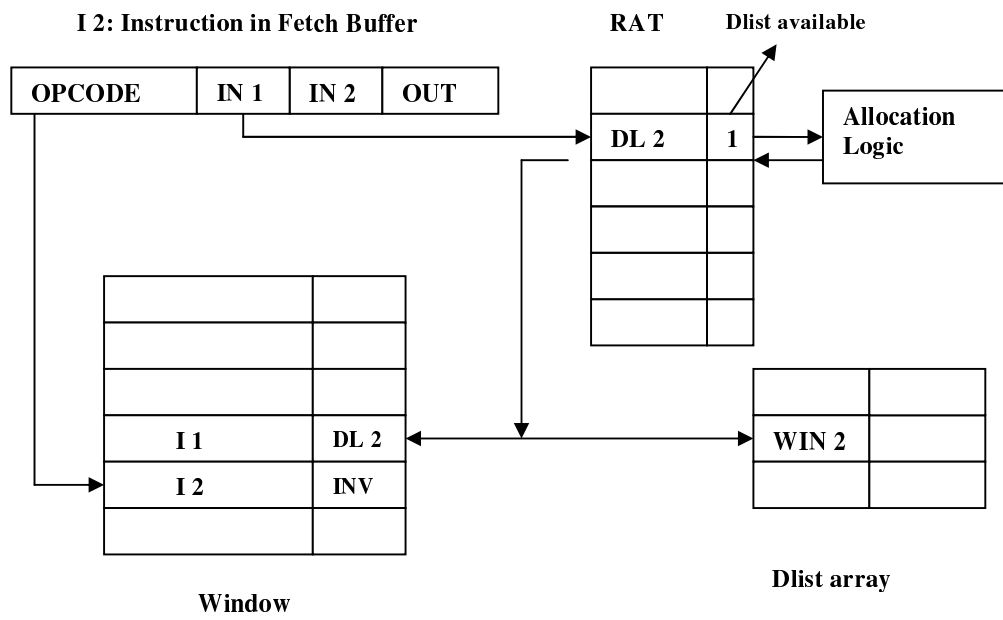


Figure 5.2: Need Based DL (NBDL) wake-up

the corresponding Dlist array slot is read and the dependent instructions are woken up. Subsequently, the producer instruction's Dlist slot is freed.

If, at any point, the producer instruction cannot be assigned a Dlist slot because there are no more free Dlist slots or if the Dlist of a producer is already full, then the instruction dispatch is stalled until further slots are available. As the Dlist array and Window no longer have a static, one-to-one mapping, the Window entries now need to store a tag corresponding to the index allotted to them in the Dlist array. But this is not a problem because this tag just takes the place of the tags of the input operands in the case of conventional wake-up and does not increase the space or energy consumption of the Window.

Figure 5.2 illustrates the NBDL wake-up scheme. In the figure, instruction I2 is dependent on instruction I1 for its input operand IN1. At dispatch time, the instruction indexes into the RAT using IN1. The Dlist available bit is read and if the bit is 0, a fresh Dlist entry is allocated to the producer and the bit is set to 1. The index to the allocated slot, DL 2, is given to the dependent instruction. The dependent instruction is dispatched to the next RUU slot RUU 2. I2 now enters its ID, RUU 2, in the producer's Dlist. The Dlist index, DL 2 is also written to the RUU slot of the producer so that it can be read at wake-up. Thus, only those instructions that require Dlists are identified and allotted space in the array. Wake-up takes place similar to the DL-based scheme - when an instruction



completes execution, the Dlist of the instruction is read using the stored index and the dependent instructions are woken up. Again, the Dlist array look-up can be carried out in parallel with the operand reading to the functional units and the execution of the instructions itself.

## 5.4 Energy Analysis

The energy consumption of the NBDL-based scheme can be analysed similar to DL-based scheme in Chapter 4. All the components of the energy are the same. The energy savings in NBDL scheme when compared to the DL-based scheme arise from the fact that the Dlist array is of a smaller size, i.e., it has fewer rows. This in turn reduces both the bit line energy and decoder energy proportionally. As all the energy components of the NBDL scheme are similar to the DL-based scheme, they vary similarly with Window size and Issue Width and this implies that the NBDL scheme scales better than the conventional scheme with Window Size and Issue Width.

## 5.5 Effect of NBDL scheme on delay

Figure 5.3 depicts the dispatch stage of an out-of-order superscalar processor implementing the NBDL scheme for wake-up. It can be seen that the logic in the dispatch stage is similar to that in the DL-based scheme. The additional

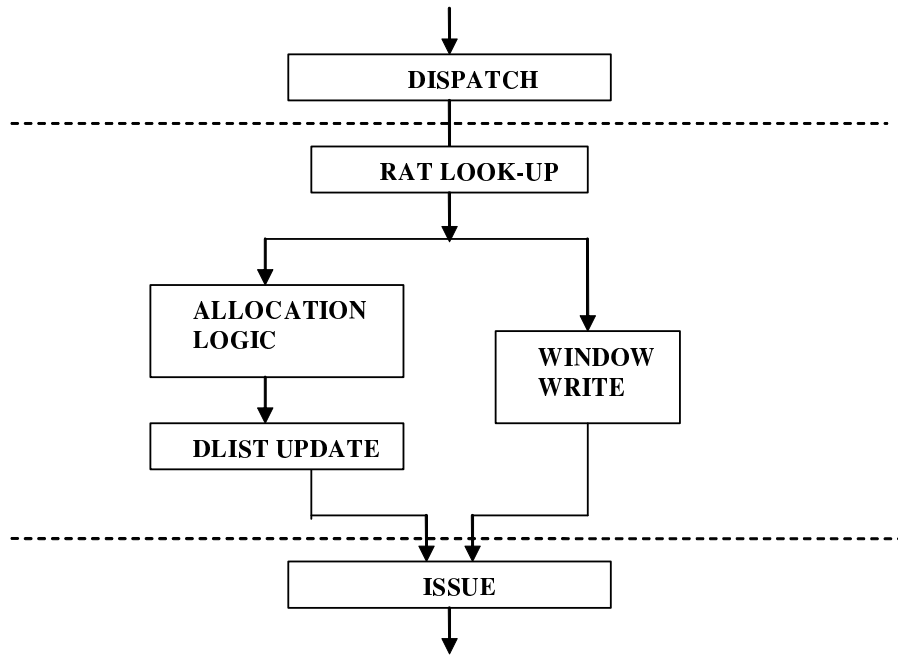


Figure 5.3: Effect of NBDL scheme on the delay of dispatch stage

component is the allocation logic that dynamically allocates Dlist array slots to instructions in the Window. A simple implementation of this allocation logic is implementing the Dlist array as a queue and allocating Dlist slots from the tail. The Dlist slots can be freed when the corresponding instructions commit. In this way, the allocation logic would only have to check if the head and tail pointer are equal to check if there are any more free slots in the Dlist array. Since it involves only a pointer comparison, the allocation logic has very little delay. Hence, we estimate that the NBDL scheme does not significantly increase the delay of the dispatch stage. This means that the clock cycle of the processor would not be considerably affected.

## 5.6 Impact of NBDL scheme on Processor Area

The microarchitecture of the NBDL scheme is similar to that of the DL-based scheme. The only difference is that, since the Dlist entries are dynamically allocated to the instructions in the Window, the instructions in the Window now need to store a pointer to their corresponding Dlist array entry. If an instruction does not have a Dlist entry, this pointer is set to invalid. However, the size of this pointer is  $\lg DLsize$  where  $DLsize$  is the number of rows in the Dlist array. We estimate that this additional storage is not significant. Again, we note that since the Window is no longer a CAM, it occupies much lesser space. Thus, we infer that the NBDL scheme would not increase the processor area significantly.

## Chapter 6

# Experimental Analysis

This chapter presents the experimental results and discusses their ramifications. Section 6.1 gives the details of the experimental setup. Section 6.2.1 presents the performance results of both the schemes in comparison with the conventional scheme. The wake-up energy savings obtained by both the schemes (DL-based and NBDL) are discussed in Section 6.2.2. The Instruction Window power savings are described in Section 6.2.3. Finally, section Section 6.2.5 provides the details on the scalability of our schemes.

## 6.1 Experimental Setup

### 6.1.1 Simulator

We used the sim-outorder of the SimpleScalar that simulates an out-of-order, superscalar processor as the baseline. It is a performance level simulator[17]. We extended this to simulate the DL-based scheme and the NBDL scheme. This

primarily involved adding a Dlist array and an allocation logic to the underlying architecture. We used Sim-wattch, which is an add-on tool for SimpleScalar, to obtain energy and power estimates[18].

### **6.1.2 Benchmarks**

We selected a subset of the SPEC2000 benchmark suite: `bzip2`, `gcc`, `gzip`, `mcf`, `parser`, `twolf`, `vortex` and `vpr` to run simulations. 500 million instructions of each benchmark were simulated after fast forwarding 200-500 million instructions depending on the length of each benchmark's initialization segment.

### **6.1.3 Baseline**

We chose the conventional associative wake-up as the baseline. We simulated three wake-up schemes: the conventional, the DL-based and the NBDL scheme. The savings are quantified as relative energy of the new schemes in comparison with the conventional wake-up energy.

### **6.1.4 Microarchitectural parameters**

The principal parameters of the microarchitecture used for the simulations are given in Figure 6.1

Fetch Width	4, 8
Issue Width	4, 8
Commit Width	4,8
RUU Size	32, 64, 128, 256
Load/Store Queue	16, 32, 64, 128
Functional Units	4, 8

### 6.1.5 Simulation Parameters

The Window size is varied from 32 entries to 256 entries and the Issue Width is increased correspondingly in order to study the scalability aspects of the new schemes. The Dlist length, i.e. the number of dependent instructions a producer instruction can keep track of, is varied between 1 and 4. In the NBDL scheme, the Dlist array size is made half the size of the Window.

## 6.2 Results

### 6.2.1 Performance Results

Figure 6.1 shows the relative IPC achieved by the DL-based wake-up scheme for Dlist lengths 1, 2, and 4 with respect to the baseline conventional wake-up scheme. The baseline IPC (1.0) is also shown. The results are shown for eight benchmarks in the SPEC2000 suite. The Window size is set at 64 and Issue Width at 4. The last set of columns shows the average over all the eight benchmarks. The DL-based wake-up schemes with Dlist lengths 1, 2 and 4 are named DL 1, DL 2, and DL 4 respectively. The baseline conventional wake-up scheme is called Conv.

Figure 6.1 shows that the relative IPC in most cases is less than 1. This implies that the throughput decreases when we move to the DL-based wake-up

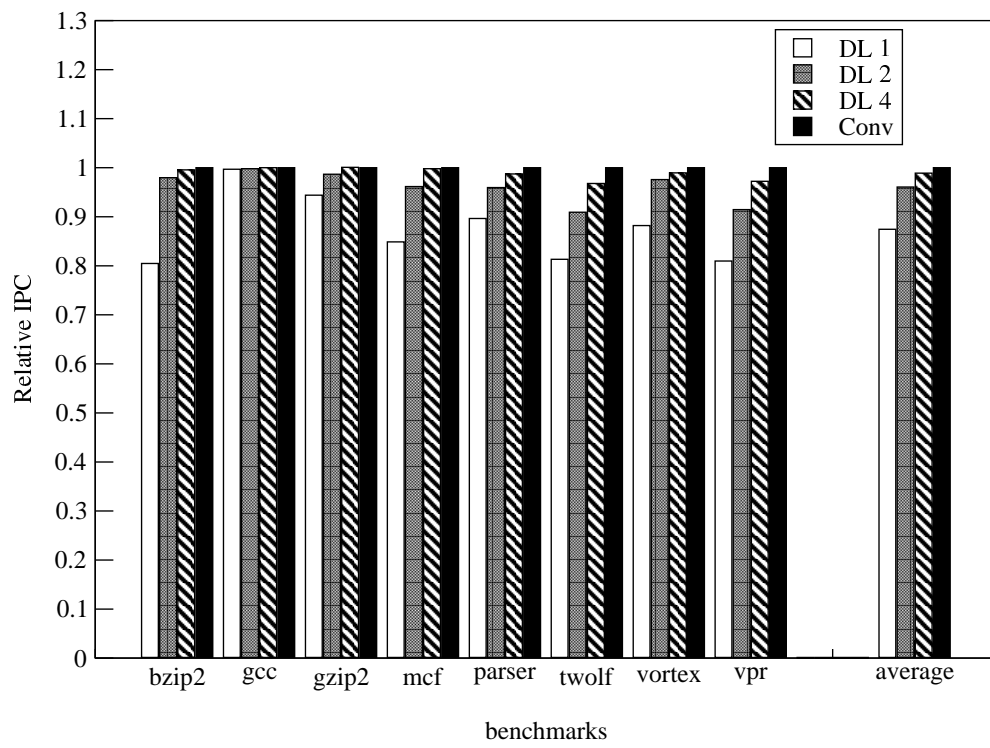


Figure 6.1: Relative IPC of DL-based scheme. Window size 64, Issue Width 4



from the conventional wake-up. This decrease is because of the dispatch stalls that arise from lack of space in the producer's Dlist. Because the number of instructions in a producer's dependence list is restricted, instructions that cannot be accommodated in the producer's Dlist cannot be dispatched until the producer instruction completes execution and writes the result to the register file. And because dispatch is in order, no other instructions can be dispatched until the stalled instruction is dispatched.

Another observation from the graph is that as the Dlist length increases, i.e. as we go from DL 1 scheme to DL 4 scheme, the average relative IPC increases. This can be explained as follows: In the case of conventional wake-up, we can accommodate any number of dependents of an instruction as long as there is space in the Window because the availability of operands is fanned out to the entire Instruction Window. Therefore, the effective Dlist length is infinity and hence the dispatch stalls are completely avoided. Thus, as Dlist length is increased, the performance of the DL-based wake-up scheme moves towards that of the conventional wake-up scheme. On an average, DL 1 achieves 86% of the performance of the conventional wake-up scheme and DL 2 and DL 4 achieve 95% and 98% respectively. DL 2 achieves most of the IPC of the conventional wake-up scheme. This confirms the fact that most instructions have at most 2 dependents [2]. In conclusion, all the DL-based schemes achieve most of the throughput of the baseline scheme, especially the DL 2 and DL4 schemes.

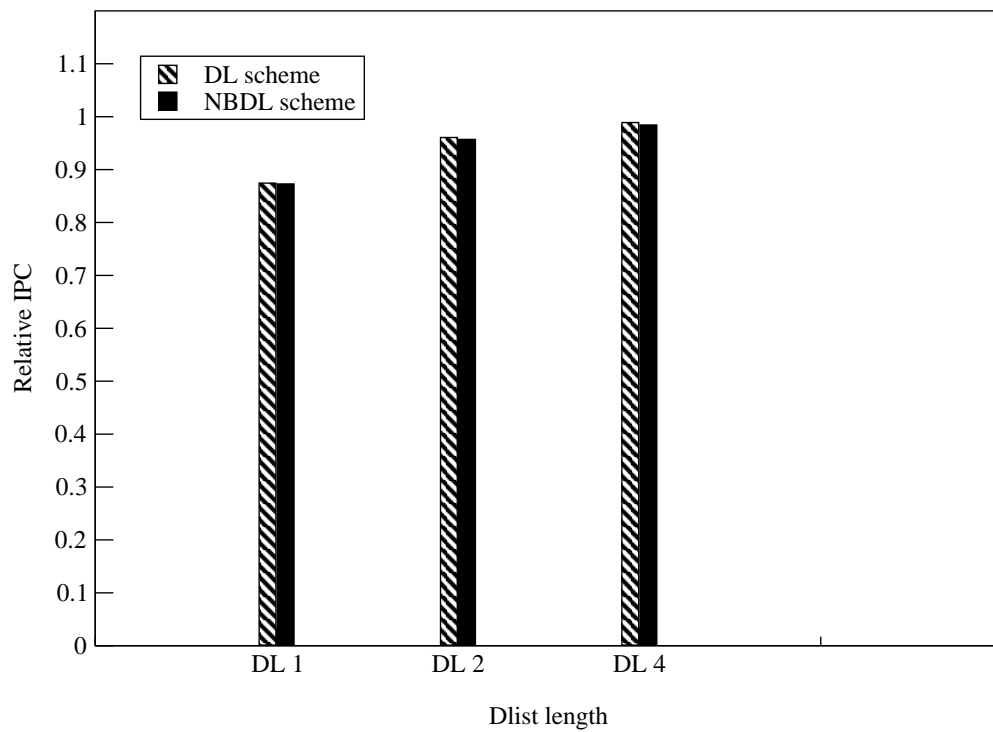


Figure 6.2: Performance comparison of DL based and NBDL wake-up scheme.

Window size 64, Issue Width 4

Figure 6.2 shows the relative IPC achieved by the DL based wake-up scheme and the NBDL scheme for three Dlist lengths. The IPC is averaged over the same 8 benchmarks as before. In this comparison, the size of the Dlist array is set to half the size of the Instruction Window, i.e. 50% of RUU size. As noted in the previous section, 60% of the instructions in the Window need Dlist slots so the required size of the Dlist array would be 60% of the Window size. We made the Dlist array slightly smaller, just to make the Dlist array size a power of two. The Window size is again set at 64 and the Issue Width is 4. Figure 6.2 shows that even with this smaller size of the Dlist array, the NBDL scheme achieves almost the same performance as the DL-based scheme. This implies that with the NBDL scheme, even with a Dlist array size only half the size of that in the DL-based scheme, the same IPC as in the DL-based scheme can be achieved. This is a key advantage as a fall in IPC would imply two things: (1) Performance degradation (2) Increase in overall energy consumption (as described in Section 3.3). The increase in energy consumption is because the units now run for a longer time and consume a small amount of energy even when they are idle (imperfect clock gating). On the whole, we see that the NBDL scheme performs as well as the DL-based scheme. On an average, it achieves 98% of the performance of the DL-based scheme

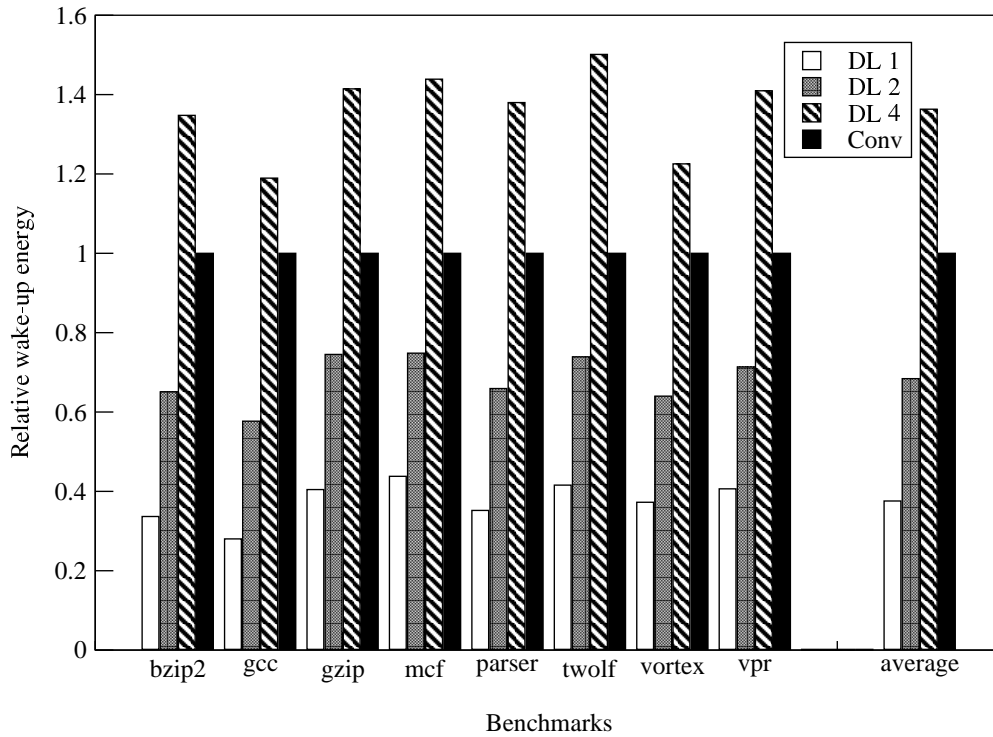


Figure 6.3: Relative wake-up energy of DL-based scheme. Window size 64, Issue Width 4

### 6.2.2 Energy Savings in Wake-up

Figure 6.3 shows the relative wake-up energy of the DL-based scheme for the three Dlist lengths compared to the conventional wake-up energy for a Window size of 64 and Issue Width of 4. The DL-based scheme saves significant amount of energy for the Dlist sizes 1 and 2, while it consumes more energy for Dlist size 4. This can be explained as follows: as Dlist length increases, the number of columns in the Dlist array increase and consequently the bit line energy

increases. This increases the wake-up energy of the DL-based scheme. On the other hand, the conventional wake-up scheme is independent of the Dlist length. So, this energy remains constant as we vary the Dlist length. So, on the whole the relative energy of DL- based schemes increase as the Dlist length increases. Correspondingly, the energy savings of the DL based schemes decrease as Dlist length increases. Figure 6.3 shows that, on an average, DL 1 consumes only 35% of the conventional wake-up energy and DL 2 consumes 67% of the conventional energy of wake-up while DL 4 consumes 138% of conventional. This implies that the DL 4 scheme actually consumes more energy than the conventional scheme. It can be concluded that the DL 2 scheme, on the whole, is the best choice as it achieves 95% of the conventional IPC while saving up to 33% of the energy in wake-up.

Figure 6.4 presents the relative wake-up energy of the DL- based scheme and the NBDL scheme for different Dlist lengths. The results shown are the wake-up energies of the DL and NBDL scheme averaged over the eight benchmarks that were selected. The baseline for comparison for both schemes is the conventional, fully associative wake-up scheme. The NBDL scheme uses a Dlist array of half the Window size. The Window size is 64 and Issue Width is 4. The wake-up energy of the conventional scheme is 1.00.

It can be seen that the NBDL scheme uses significantly less energy when compared to DL-based scheme. This is because of the size of the Dlist array in

### Relative wake-up energy of DL and NBDL schemes

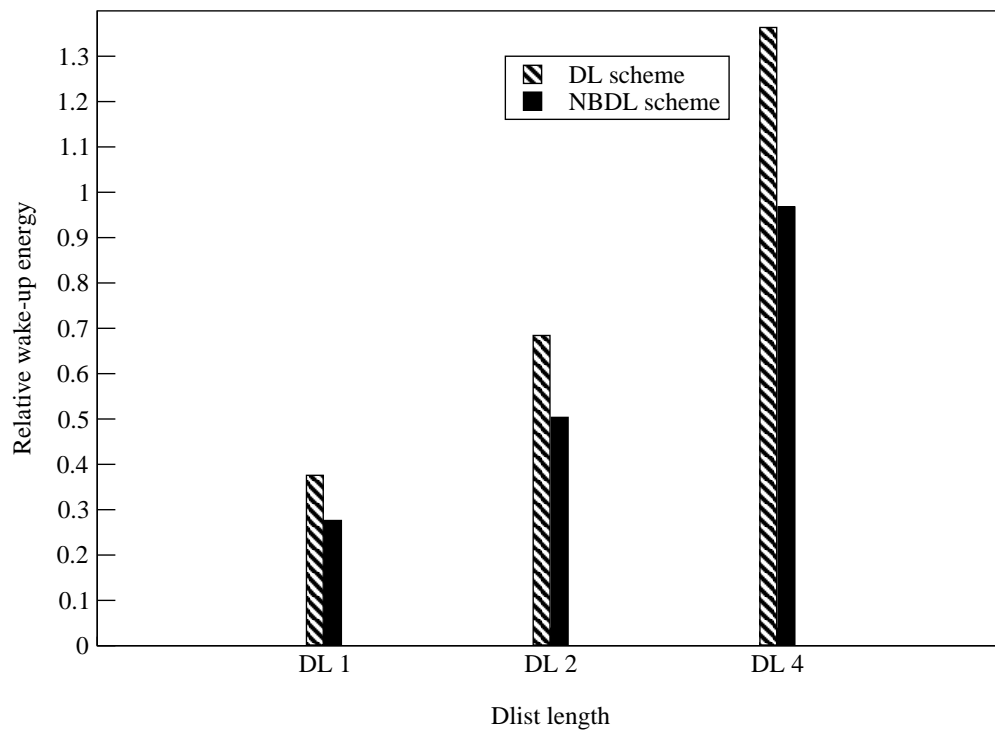


Figure 6.4: Wake-up energy comparison of DL-based scheme and NBDL scheme.

Window size 64, Issue Width 4

the NBDL scheme is half of that in the DL-based scheme. This implies a 50% reduction in the number of rows in the DL-based scheme. This in turn gives a significant reduction in the bit line energy and decoder energy. The NBDL scheme consumes 26%, 50% and 96% of the wake-up energy of the conventional scheme for Dlist lengths 1, 2 and 4.

The energy savings of NBDL scheme with respect to the corresponding DL-based scheme is 10%, 18%, and 38%. This implies that the energy savings of the NBDL scheme with respect to the DL-based scheme increase with the Dlist length. As Dlist length increases the number of columns in the Dlist array increase. This implies that the size of the Dlist array as a whole increases. Therefore, the energy saved by reducing the number of rows of the array also increases. Finally, it can be concluded that the NBDL scheme with a Dlist length size 2 performs the best as it achieves 94% of the performance of the conventional wake-up scheme while saving 50% of the wake-up energy.

### **6.2.3 Instruction Window Power Savings**

The savings in the Instruction Window power in the NBDL scheme are presented in Figure 6.5. The NBDL schemes with Dlist lengths 1, 2 and 4 are named NBDL 1, NBDL 2 and NBDL 4 respectively. Again, the size of the Dlist array is

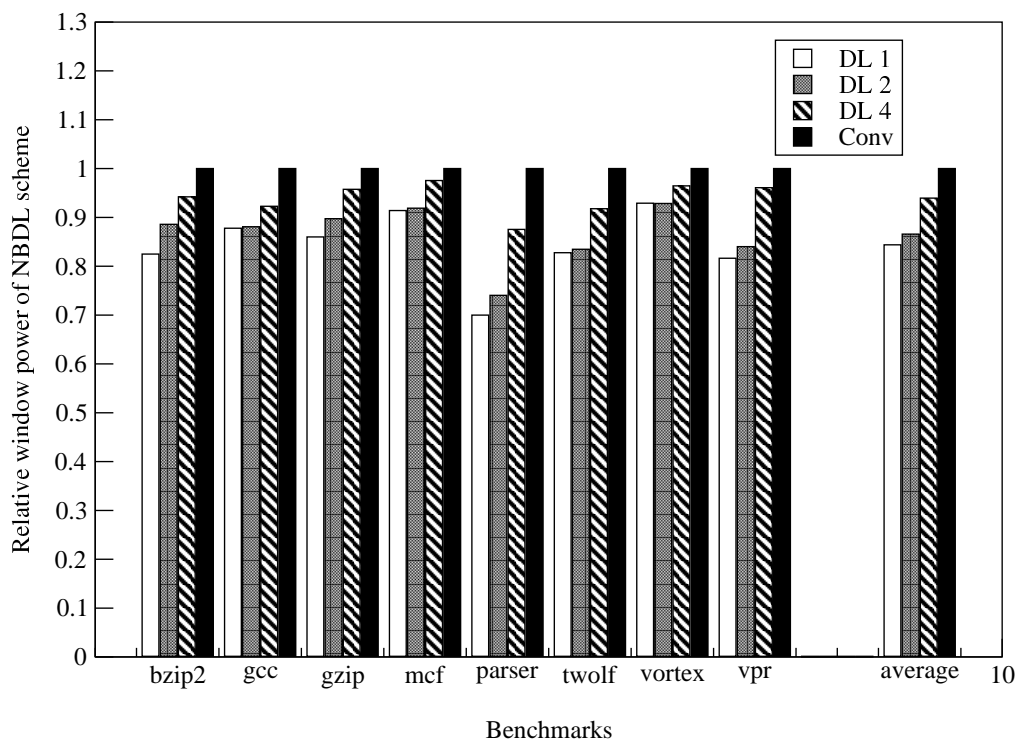


Figure 6.5: Relative Window power of NBDL scheme. Window size 256, Issue Width 8



half the Window size. The size of the Instruction Window is set to 256 entries. The Issue Width is 8. The savings in Window power range from 30% for NBDL 1 to 5% in NBDL 4. On an average, the savings in Window power are 18%, 15% and 7% for Dlist lengths 1, 2 and 4. This implies that even NBDL 4 scheme leads to a considerable reduction in Window power unlike the corresponding DL 4 scheme that actually burnt more power than the conventional scheme. This is impressive because the NBDL 4 scheme captures almost 99% of the IPC of the conventional scheme. This implies that the Instruction Window hot spot problem can be solved by using the NBDL 4 scheme with almost no decrease in performance. The overall best results are again for the Dlist length 2. NBDL 2 saves 15% of the Instruction Window power and 50% of wake-up energy (Figure 6.3) while maintaining good performance.

#### **6.2.4 Energy Delay Product**

We showed in Section 6.2.2 that the DL-based scheme and the NBDL scheme achieve significant savings in wake-up energy. However, they also suffer a slight throughput decrease. Also, the cycle time of the processor might be affected due to the extra delay of both these schemes. As all these factors are important, we evaluate the wake-up energy delay product of the DL-based scheme and the NBDL scheme relative to the conventional scheme. This measure contains all the

factors, and provides a good index to the overall performance of a scheme. The lower the Energy Delay Product, the better. The Energy Delay Product can be calculated as follows

$$EDP = Energy \times Delay \quad (6.1)$$

where  $EDP$  denotes the Energy Delay Product. Delay can be expressed as

$$Delay = CPI \times CycleTime = CycleTime/IPC \quad (6.2)$$

where CPI denotes the average number of Cycles Per Instruction, IPC denotes Instructions per Cycle and CycleTime denotes the cycle time of the processor.

Using the above equations, the wake-up energy delay product of the DL-based and NBDL schemes can be computed as follows:

$$EDP_{DLwkup} = Energy_{DLwkup} \times ClockCycle_{DLwkup}/IPC_{DLwkup} \quad (6.3)$$

$$EDP_{NBDLwkup} = Energy_{NBDLwkup} \times ClockCycle_{NBDLwkup}/IPC_{NBDLwkup} \quad (6.4)$$

where the subscripts  $DLwkup$  and  $NBDLwkup$  indicate the DL-based wake-up scheme and the NBDL wake-up scheme, respectively

Figure 6.6 shows the variation of the wake-up energy delay product with percentage increase in cycle time of the processor due to using the DL-based or the NBDL scheme. This graph gives us an estimate of the increase in the cycle time that can be tolerated by these schemes to perform better than the conventional scheme. The graph shows three curves, one for each of the wake-up schemes: the conventional, the DL 2, and the NBDL 2. The energy delay

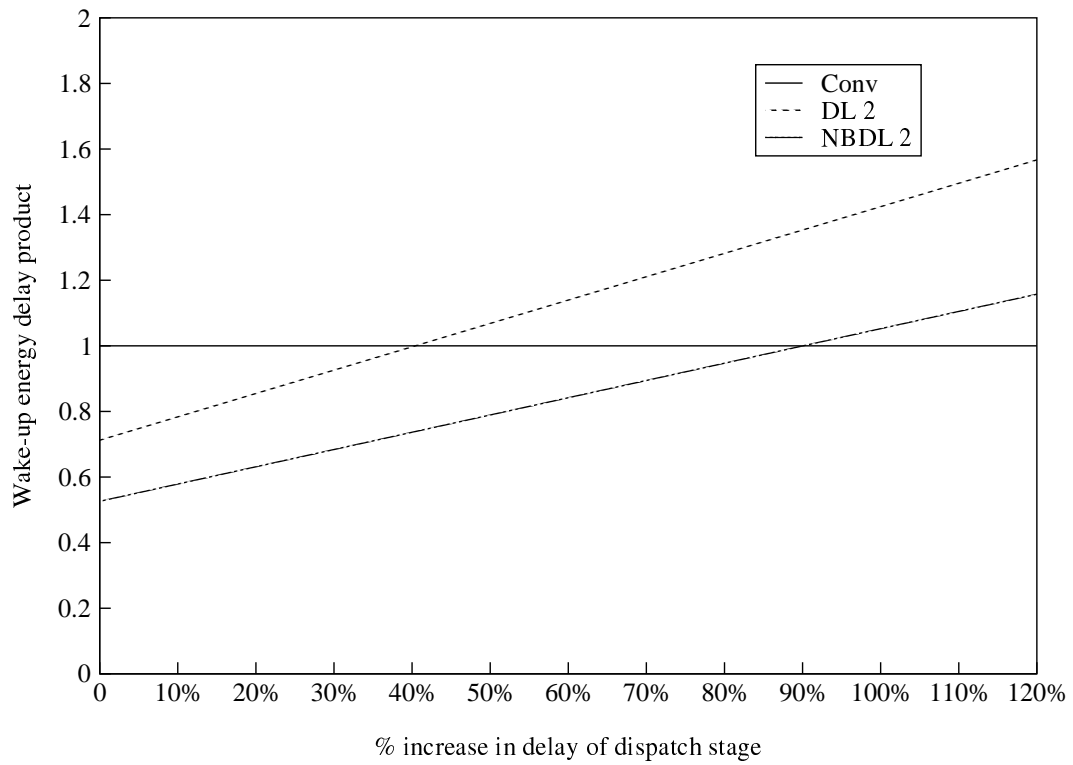


Figure 6.6: Wake-up energy delay product as a function of increase in delay

products are averaged over all the benchmarks and Window sizes. Because the wake-up energy delay products of the DL-based scheme and the NBDL scheme are given relative to the conventional scheme, the line corresponding to the conventional scheme remains flat at 1. As the cycle time increases due to the excess delay of wake-up, the wake-up energy delay product of the DL-based scheme and that of the NBDL scheme increases linearly. The point of intersection of these lines with the line corresponding to the conventional scheme gives the percentage increase in cycle time above which the conventional scheme performs better than the respective schemes. From Figure 6.6 we see that this happens at 40% for the DL-based scheme, and 90% for the NBDL scheme. As neither of these schemes can cause such a significant increase in delay, we can conclude that these schemes perform better than the conventional scheme.

### **6.2.5 Scalability with Increasing Window Size and Issue Width**

In order to achieve a higher throughput, more instructions should be executed in a given time period. One way to achieve this is to increase the number of instructions being executed in parallel per cycle. In order to do this, we need to find enough independent instructions in the dynamic instruction stream to keep all the parallel execution units occupied. This, in turn can be achieved by looking deeper into the dynamic instruction stream. This is implemented by increasing

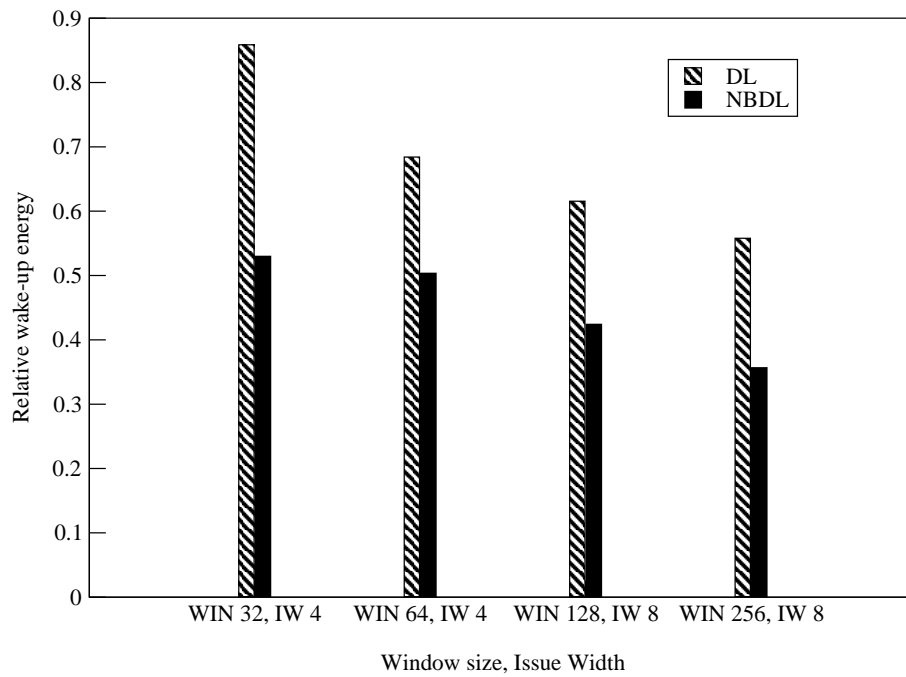


Figure 6.7: Scalability of wake-up energy of NBDL scheme with Window size and Issue Width

the Window size. Thus, in order to improve performance, the Issue Width and Window size will be increased. And so, it is important to study the scalability of our schemes with these parameters. Figure 6.7 shows the impact of the Window size and Issue Width on the relative wake-up energy in the DL and NBDL schemes. The Window size is varied from 32 to 256 entries and the Issue Width is correspondingly increased from 4 to 8. The results are averaged over the eight benchmarks. As these parameters are increased, the relative wake-up energy of both DL and NBDL schemes decreases. The decrease is especially sharp from Window size 64 to Window size 128. This is because the Issue Width also increases simultaneously from 4 to 8. The figure shows that as the Window size and Issue Width increases, the relative wake-up energy of both schemes with respect to the conventional decreases non-linearly. This implies that both our schemes are more scalable than the conventional scheme. This is a key advantage because this implies that energy savings will scale with Window size and Issue Width. In effect, this implies that the savings will scale with throughput which is wonderful.

## 6.2.6 Performance of DL-based and NBDL Schemes for a Split Window Organization

So far, we have only discussed the wake-up energy savings possible for an architecture that uses a unified Window, to store dynamic instructions throughout their lifetime. An alternate architecture uses a split Instruction Queue (IQ) and Re-Order Buffer (ROB) where the IQ holds all the un-issued instructions and the ROB contains all the instructions that have been dispatched and not yet committed. In this case, conventional wake-up would involve broadcasting the availability to only the IQ. As the number of entries in the IQ is typically about 30-40% of the ROB, the broadcast would be less expensive and as a result the savings in wake-up energy by using the DL-based scheme or the NBDL scheme would be somewhat lower. However, using the DL-based scheme is advantageous as the energy consumption of broadcast will still increase rapidly with increasing IQ size and Issue Width. The energy consumption of the DL-based scheme and the NBDL scheme scales better with these two parameters, and hence these schemes will perform better than the conventional scheme in the future, even for a split Window architecture.

### 6.2.7 Overall Processor Power Savings

The overall processor power savings for the DL-based scheme averaged over the four Window sizes (32, 64, 128 and 256) is 25% for DL, 1,6% for DL 2, and 2% for DL 4. DL 1 achieves very high overall power savings because there is a significant trade-off of IPC and also a considerable reduction in the wake-up energy. As DL 2 and DL 4 achieve most of the IPC, the overall power savings arise only out of the wake-up energy. These figures, however, are conservative estimates because they do not reflect the fact that the Window writes at dispatch are smaller for the DL-based scheme. The overall power savings of the NBDL scheme are about 2% higher than the DL-based scheme.

In summary, we see that all of the DL-based schemes and NBDL schemes achieve most of the throughput of the baseline scheme, especially with Dlist lengths 2 and 4. Simultaneously, both these schemes achieve considerable savings in the wake-up energy and Window power when compared to the conventional scheme. It can be concluded from the results that a Dlist length of 2 performs the best for both the schemes. For this Dlist length, the DL-based scheme saves 33% of the wake-up energy on the average, while trading off only 2% of the throughput while the NBDL scheme achieves wake-up energy savings of 50% for the same small loss of throughput. Window power is also correspondingly reduced (15% savings for the NBDL scheme).



## Chapter 7

### Conclusions

Energy consumption of a processor is a very important factor in the design of the processor. The Instruction Window of an out-of-order processor forms the core of the processor as it houses the dynamic instructions for most of their lifetime.

Consequently, it is accessed several times in the pipeline. This leads to its high energy consumption. Also, due to the large power consumption just within the Window region, the power density of the Instruction Window is very high causing a local hot spot. This causes the cost of cooling and packaging of the chip to increase. It also causes problems in ensuring chip reliability. In the current day superscalars, wake-up logic consumes a significant portion of the Instruction Window energy consumption and hence reducing the wake-up energy solves the twin problems of high Window energy consumption and the Instruction Window hot spot.

In this work, we evaluated an alternate wake-up scheme based on maintaining dependence lists of instructions, effectively moving to a lookup-based wake-up from conventional associative wake-up. In order to explain the energy savings

obtained, we analyzed the energy consumption of the conventional wake-up scheme and our scheme. We compared the wake-up energy to that of a conventional superscalar over 4 different Window sizes and corresponding Issue Widths. Our results show that significant energy savings can be achieved from this scheme with very little trade-off in performance. Also, the energy savings increase with increasing Instruction Window size and Issue Width implying that this scheme is more scalable than the conventional. This is a really important advantage as both these parameters are going to be increased in the quest for higher performance.

Our observations from the energy analysis of the DL based wake-up scheme led us to investigate a possibility of further energy savings. For this purpose, we did a study on the number of instructions in the window that actually feed other instructions (Parent instructions). We found that, on an average, only about 60 percent of the instructions in the Window at any time are Parent instructions. Based on this, we presented the Need Based DL (NBDL)scheme that uses a smaller array to store dependent information and dynamically allocates Dlist storage slots to only Parent instructions. Our results indicate that we can save up to 50 percent of the wake-up energy, i.e. up to 15 percent of the overall Window power, compared to a processor using the conventional wake-up scheme. This is significant because the Window energy (IQ + ROB) is one of the highest consumers of energy in a superscalar. Like the DL-based scheme, our NBDL scheme is more scalable when compared to the conventional scheme with

increasing Window size and Issue Width.

## BIBLIOGRAPHY

- [1] D. Folegnani and A. Gonzalez. Energy-Effective Issue Logic. In *Proceedings of International Symposium on Computer Architecture*, pages 230–239, Jul 2001.
- [2] M. Huang, J. Renau, and J. Torrellas. Energy Efficient Hybrid Wakeup Logic. In *Proceedings of International Symposium on Low Power Electronics and Design*, pages 196–201, Aug 2002.
- [3] T. Sato, Y. Nakamura, and I. Arita. Direct Tag Search Algorithm on Superscalar processors. In *Proceedings of Workshop on Complexity Effective Design, International Symposium on Computer Architecture*, Jun 2001.
- [4] S. Onder and R. Gupta. Instruction Wake-up in Wide Issue Superscalars. In *Proceedings of European Conference on Parallel Computing*, pages 418–427, Aug 2001.
- [5] S. Palacharla, N. Jouppi, and J. Smith. Complexity-Effective Superscalar Processors. In *Proceedings of International Symposium on Computer Architecture*, pages 206–218, Jun 1997.

- [6] R. Canal and A. Gonzalez. A Low Complexity Issue Logic. In *Proceedings of International Conference on Supercomputing*, pages 327–335, Jun 2000.
- [7] R. Canal and A. Gonzalez. Reducing the Complexity of Issue Logic. In *Proceedings of International Conference on Supercomputing*, pages 312–320, Jun 2001.
- [8] J.L. Hennessey and D.A. Patterson. *Computer Architecture A Quantitative approach*. Morgan Kauffman, San Francisco, California, 1990.
- [9] R.M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units . *IEEE Journal Research and Development*, 11:25–33, Jan 1967.
- [10] J.E. Smith and A.R. Pleszkun. Implementation of Precise Interrupts in Pipelined Processors. In *Proceedings of International Symposium on Computer Architecture*, pages 36–44, Jun 1985.
- [11] G.S. Sohi and S.Vajaypayem. Instruction Issue Logic for High-Performance Pipelined Processors. In *Proceedings of International Symposium on Computer Architecture*, pages 27–36, Jun 1987.
- [12] D. Ponomarev, G. Kucuk, and K. Ghose. Energy-Efficient Design of the Reorder Buffer. In *Proceedings of the International Workshop on Power and Timing Modeling, Optimization and Simulation*, Sep 2002.
- [13] G. Kucuk, K. Ghose, D.Ponomarev, and P. Kogge. Energy-Efficient Instruction Dispatch Buffer Design for Superscalar Processors. In

- Proceedings of International Symposium on Low Power Electronics and Design*, pages 237–243, Aug 2001.
- [14] A. Buyuktosunoglu, T. Karkhanis, D.H. Albonesi, and P. Bose. Energy Efficient Co-adaptive Instruction Fetch and Issue. In *Proceedings of International Symposium on Computer Architecture*, pages 147–156, Jun 2003.
- [15] A. Buyuktosunoglu, S. Schuster, D. Brooks, P. Bose, P. Cook, and D. Albonesi. An Adaptive Issue Queue for Reduced Power at High Performance. In *Proceedings of Workshop on Power-Aware Computer Systems, ASPLOS*, Nov 2000.
- [16] S. Palacharla, N. Jouppi, and J. Smith. Quantifying the Complexity of Superscalar Processors. Technical Report CS-TR-96-1328, Dept of CS, Univeristy of Wisconsin-Madison, Nov 1996.
- [17] D. Burger and T.M. Austin. The SimpleScalar tool set: Version 2.0. Technical Report CS-TR-97-1342, Dept of CS, Univeristy of Wisconsin-Madison, Jun 1997.
- [18] D. Brooks, V. Tiwari, and M. Martonisi. Wattch: A Framework for Architectural Level Power Analysis and Optimizations. In *Proceedings of International Symposium on Computer Architecture*, pages 87–94, Jun 2000.

