MITIGATING THE EFFECT OF MISSPECULATIONS IN SUPERSCALAR PROCESSORS

By

Zhaoxiang Jin

A DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

In Computer Science

MICHIGAN TECHNOLOGICAL UNIVERSITY

2018

This dissertation has been approved in partial fulfillment of the requirements for the Degree of DOCTOR OF PHILOSOPHY in Computer Science.

Department of Computer Science

Dissertation Advisor:	*Dr. Soner Onder*
Committee Member:	*Dr. Zhenlin Wang #1*
Committee Member:	*Dr. Saeid Nooshabadi #2*
Committee Member:	*Dr. David Whalley #3*
Department Chair:	*Dr. Min Song*

Dedication

To my parents: Jin, Jianmin (Dad) and Xu, Huizhu (Mom)

You always encourage me to pursue my dreams and provide me everything I need. Without your inspiration and help, there was no way for me to get this far.

To my advisor: Önder, Soner

Who didn't hesitate to criticize my work at every stage - without which I would neither be who I am nor would this work be what it is today.

To my friends

Thank you for being my friends and never giving up on me. I have learned a lot things which you can never learn from any book in this world, from all of you. Without you, it's impossible for me to come with these innovative ideas.

Contents

List of Figures . . . . **xv**

List of Tables . . . . **xix**

Author Contribution Statement . . . . **xxi**

List of Abbreviations . . . . **xxiii**

Abstract . . . . **xxv**

1 Introduction **1**

1.1 A Superscalar Processor 5

1.1.1 The Front-end 5

1.1.2 The Back-end 6

1.1.3 Register Alias Table 7

1.2 Speculative Execution 8

1.2.1 Branch Prediction 8

1.2.2 Memory Ordering Prediction 11

1.3 Misspeculation Recovery 14

1.3.1 Recovering Branch Predictor State 16

1.3.2 Restoring the Map Tables 18

1.3.3 Eliminating Stale Instructions 19

1.4 Recovery Penalty Analysis 19

1.4.1 Misspeculation Detection Delay 20

1.4.2 Stale Instruction Elimination Delay 21

1.4.3 State Restoration Delay 21

1.4.4 Pipeline Fill Delay 22

1.5 Summary 23

2 Mower : A Walking-based Misspeculation Recovery Mechanism 25

2.1 Overview 25

2.1.1 Recovering F-RAT by Walking 26

2.1.2 Checkpointing Architectures 29

2.1.3 Motivation 31

2.2 Control Dependence Tracking 35

2.2.1 Branch Renaming 35

2.2.2 Disambiguating the Branch Ordering 36

2.2.3 Tracking Affected F-RAT entries 38

2.3 A Reverse Walking Procedure 39

2.3.1 Recovering F-RAT in a gradual manner 39

2.3.2 Eliminating Stale Instructions 40

2.4 Microarchitecture 42

2.4.1 Branch Tag Allocation and Release 43

2.4.2 Branch and Mapping Dependence Matrices 44

2.4.3 Reclaiming the Resources using a single Read Port 46

2.5 Evaluation 47

2.5.1 Evaluation Methodology 48

2.5.2 Branch Tag Size Effect 51

2.5.3 Invalid Register Mappings 52

2.5.4 Eliminating Invalid Instructions 53

2.5.5 The Front-end and Back-end depth 54

2.5.6 Energy Efficiency 55

2.6 Summary 58

3 Two-Phase Misspeculation Recovery 61

3.1 Overview 61

3.1.1 A Basic Recovery Mechanism 62

3.1.2 Analysis and Related Work 64

3.1.3 Key Observation 67

3.2 Preliminary Analysis 70

3.3 Microarchitecture 73

3.3.1 Initialization 73

3.3.2 Three Different Flushing Policies 75

3.3.3 Fetch Policy 76

3.3.4 F-RAT and Free Register Pool 77

3.3.5 Speculative Recovery 77

3.3.6 Complexity Comparison 78

3.4 Evaluation 80

3.4.1 Simulation Methodology 80

3.4.2 Recover Afterwards and Two-Phase 85

3.4.3 Recover Beforehand and Two-Phase 87

3.4.4 Allocation Algorithms in Checkpointing Architectures 89

3.4.5 Memory Latency Effect 90

3.4.6 ROB Size Effect 91

3.4.7 Issue Width Effect 92

3.4.8 Power Efficiency 92

3.5 Summary 93

4 Passing Loop : Reducing the Pipeline Fill Delay 95

4.1 Overview 95

4.1.1 Control Independence 96

4.1.2 The Convergence Point Prediction 97

4.1.3 The Affected Register Mapping 98

4.1.4 Correct Instruction Insertion 99

4.2 Related Work 99

4.2.1 Eager Execution 100

4.2.2 Control Independence 104

4.3 The Concept of Passing Loop 106

4.4 Microarchitecture 110

4.4.1 Front-end: Fetching Algorithm 112

4.4.2 Handling Memory Instructions 115

4.4.3 Back-end: Misprediction Recovery 116

4.4.4 CIDD redundancy 120

4.5 Experimental Results 121

4.5.1 Simulation Methodology 121

4.5.2 Performance Analysis 124

4.5.3 Energy Efficiency 132

4.5.4 Design Complexity 134

4.6 Summary 135

5 Dynamic Memory Dependence Predication 137

5.1 Overview 138

5.2 Motivation 140

5.3 The Concept of Memory Predication 144

5.4 Microarchitecture 148

5.4.1 Store Vulnerability Window 149

5.4.2 Tagged Store Sequence Bloom Filter 150

5.4.3 Load Re-execution 151

5.4.4 Memory Dependence Prediction 152

5.4.5 Memory Cloaking 153

5.4.6 Predication Insertion 155

5.4.7 Physical Register Reference Counter 157

5.4.8 Load Re-execution Filter 159

5.4.9 Silent Store Effect 160

5.4.10 Partial-Word Forwarding 161

5.4.11 Confidence Predictor 163

5.4.12 Memory Consistency 164

5.5 Evaluation Methodology 165

5.6 Experimental Results 169

5.6.1 NoSQ VS. Baseline 169

5.6.2 DMDP VS. Baseline 171

5.6.3 DMDP VS. NoSQ 172

5.6.4 DMDP VS. Perfect 173

5.6.5 Case Study in *bzip2* 176

5.6.6 Store Buffer Size Effect 177

5.6.7 Alternative Configurations 178

5.6.8 Energy Efficiency 179

5.7 Related Work 180

5.8 Summary 183

6 Conclusion 185

References 189

A Copyright Permission 205

List of Figures

1.1 The block diagram of a superscalar processor 5

1.2 From logical register space to physical register space 7

1.3 Gshare branch predictor 9

1.4 Branch Target Buffer 10

1.5 Return Address Stack 10

1.6 Store Queue 12

1.7 Load Queue 13

1.8 Store Set mechanism 14

1.9 Branch misprediction recovery 15

1.10 Recover Branch History Register 17

1.11 Recover Register Alias Table 19

2.1 Restoring the mapping table by history buffer 27

2.2 Renaming with a not fully correct F-RAT 33

2.3 The different walking direction in Mower 34

2.4 The branch dependence matrix 37

2.5 The mapping dependence matrix 38

2.6 Recovering F-RAT by walking from the misprediction to ROB tail . 40
2.7 Reclaiming the back-end resources via the walking process 42
2.8 The block diagram of Mower 43
2.9 Branch Dependence Register, tracking all of the unresolved branches 44
2.10 Branch Dependence Register collaborates with Branch Dependence Matrix 45
2.11 Sharing of one access port by the walker and the ROB tail 47
2.12 IPC vs number of unresolved branches 51
2.13 The number of invalid register mappings through the walking process 52
2.14 The average number of invalid instructions left in the pipeline when a misprediction is detected 53
2.15 Spec2006 Integer Speedup (Normalized to Baseline) 55
2.16 Spec2006 Float Speedup (Normalized to Baseline) 55
2.17 Power Evaluation (Normalized to Baseline) 56
2.18 EDP normalized to the baseline configuration 58

3.1 A basic recovery mechanism 63
3.2 The issue to dispatch new instructions into the LSQ when it is not fixed 66
3.3 The second phase of the recovery 68
3.4 Timeline of Recovery 69
3.5 Phase one to Phase two delay 72

3.6 The number of instructions between the cache miss load and the misprediction 73

3.7 Two phase recovery state machine 74

3.8 Overwriting the prediction 76

3.9 Misprediction due to speculative recoveries 77

3.10 The branch path tag for selective flushing 79

3.11 Recover Afterwards with/without executing bogus instructions vs. Two-Phase, normalized to Inf 86

3.12 RB-8-GA vs. RB-8-LA vs. Two-Phase, normalized to Inf 88

3.13 The extra cycles caused by Two-Phase compared with Inf 89

3.14 The Geometric mean of different checkpoint allocation algorithms, normalized to Inf. 90

3.15 The Geometric mean on different memory speed, normalized to Inf. 91

3.16 The Geometric mean on different rob size, normalized to Inf. 91

3.17 The Geometric mean on different issue width, normalized to Inf. . . 92

4.1 Control Dependence/Independence 97

4.2 CDCF 108

4.3 Passing Loop Microarchitecture Block Diagram 110

4.4 Recovery through walking 118

4.5 Detecting the damaged register mappings 120

4.6 Spec 2006 IPC speedup 125

4.7 Number of affected instructions per 1000 valid committed instructions 126

4.8 Speedup if all instructions retired are valid 127

4.9 The geometric speedup over different configurations for integer benchmarks 129

5.1 OC dependence caused strict ordering. 141

5.2 Load instruction distribution 142

5.3 Delayed loads vs. bypassing loads 143

5.4 Three different ways to read data for loads. 144

5.5 Memory dependence prediction results over low confidence loads . . 147

5.6 DMDP Microarchitecture 148

5.7 Memory Cloaking 154

5.8 Memory predication insertion 156

5.9 The producer counter 158

5.10 Load re-execution incurred by silent store 160

5.11 The decision tree of partial-word forwarding detection 161

5.12 Spec 2006 Speedup over the baseline 169

5.13 A *bzip2* code snapshot 176

5.14 32,64-entry SB VS. 16-entry SB 177

5.15 The EDP of **DMDP**, normalized to **NoSQ** 179

List of Tables

2.1 The processor configurations 50

2.2 Energy consumption and Power dissipation in Mower and Baseline . 57

3.1 The average misspeculation recovery ratio and the corresponding back-end recovery time 71

3.2 Processor Configuration 83

3.3 The misspeculations which have checkpoints 88

3.4 The EDP results of 2-phase with or without instruction reuse, compared with a 8-checkpoint architecture 93

4.1 Spec 2006 Benchmark Suite 122

4.2 The Processor Configuration 123

4.3 The branch predictor performance 124

4.4 The percentage of increased mispredictions by non-speculative Passing Loop 128

4.5 The speedup comparison between Gshare and TAGE, geometric mean over integer benchmarks 129

4.6 The speedup of different branch coverage (speculative RWalker) . . 130

4.7 The CIDD redundancy 131

4.8 The EDP compared with the baseline processor 133

4.9 Design Resource classification 134

5.1 The difference between NoSQ and DMDP on different loads 145

5.2 Load re-execution policy for different loads. 159

5.3 Baseline Processor Configuration 167

5.4 Average execution time of all loads 171

5.5 Average execution time of low confidence loads 173

5.6 Memory dependence misprediction rate 174

5.7 Load re-execution related stalls per 1k committed instructions . . . 175

Author Contribution Statement

The work in Chapter 2 was published in Proceedings of the 29th ACM on International Conference on Supercomputing. I am the first author of the publication and I contributed all of the ideas and the evaluations of the work.

The work in Chapter 3 will be published in Proceedings of the 32nd ACM on International Conference on Supercomputing. I am the first author of the publication and I contributed all of the ideas and the evaluations of the work.

The work in Chapter 5 will be published in Proceedings of the 45th Annual International Symposium on Computer Architecture. I am the first author of the publication and I contributed all of the ideas and the evaluations of the work.

List of Abbreviations

ROB	Reorder Buffer
RS	Reservation Station
LSQ	Load-Store Queue
RAT	Register Alias Table
F-RAT	Front-end Register Alias Table
R-RAT	Retirement Register Alias Table
CD	Control Dependent
CI	Control Independent
CIDD	Control Independent Data Dependent
CIDI	Control Independent Data Independent
RISC	Reduced Instruction Set Computer
CISC	Complex Instruction Set Computer
ILP	Instruction Level Parallelism
TLP	Thread Level Parallelism
BTB	Branch Target Buffer
PC	Program Counter
BHR	Branch History Register
PHT	Pattern History Table

RAS	Return Address Stack
SSID	Store Set ID
SSIT	Store Set ID Table
LFST	Last Fetched Store Table
WPE	Wrong Path Events
CAM	Content Addressable Memory
ISA	Instruction Set Architecture
IPC	Instructions Per Cycle
EDP	Energy Delay Product
CFG	Control Flow Graph
LLC	Last Level Cache
MPKI	Mispredictions Per 1k Instructions
DDG	Dynamic dependency graph
DMDP	Dynamic Memory Dependence Predication

Abstract

Modern superscalar processors highly rely on the speculative execution which speculatively executes instructions and then verifies. If the prediction is different from the execution result, a misspeculation recovery is performed. Misspeculation recovery penalties still account for a substantial amount of performance reduction. This work focuses on the techniques to mitigate the effect of recovery penalties and proposes practical mechanisms which are thoroughly implemented and analyzed.

In general, we can divide the misspeculation penalty into four parts: **misspeculation detection delay**; **stale instruction elimination delay**; **state restoration delay** and **pipeline fill delay**. This dissertation does not consider the detection delay, instead, we design four innovative mechanisms. Some of these mechanisms target a specific recovery delay whereas others target multiple types of delay in a unified algorithm.

Mower was designed to address the **stale instruction elimination delay** and the **state restoration delay** by using a special walker. When a misprediction is detected, the walker will scan and repair the instructions which are younger than the mispredicted instruction. During the walking procedure, the correct state is restored and the stale instructions are eliminated.

Based on Mower, we further simplify the design and develop a Two-Phase recovery mechanism. This mechanism uses only a basic recovery mechanism except for the case in which the retire stage was stalled by a long latency instruction. When the retire stage is stalled, the second phase is launched and the instructions in the pipeline are re-fetched. Two-Phase mechanism recovers from an earlier point in the program and overlaps the recovery penalty with the long latency penalty.

In reality, some of the instructions on the wrong path can be reused during the recovery. However, such reuse of misprediction results is not easy and most of the time involves significant complexity. We design Passing Loop to reduce the **pipeline fill delay**. We applied our mechanism only for short forward branches which eliminates a substantial amount of complexity.

In terms of memory dependence speculation and associated delays due to memory ordering violations, we develop a mechanism that optimizes store-queue-free architectures. A store-queue-free architecture experiences more memory dependence mispredictions due to its aggressive approach to speculations. A common solution is to delay the execution of an instruction which is more likely to be mispredicted. We propose a mechanism to dynamically insert predicates for comparing the address of memory instructions, which is called "Dynamic Memory Dependence Predication" (DMDP). This mechanism boosts the instruction execution to its earliest point and reduces the number of mispredictions.

Chapter 1

Introduction

In recent years, more and more processor cores are integrated on a single chip to exploit more thread level parallelism. During this period, it became clear to processor designers that the clock frequency can no longer be increased due to physical limitations. About ten years ago, we had Intel Pentium 4 (Cedar Mill) which was running under 3.6GHz at 65nm process node. Nowadays the top of the line desktop processor, Intel Skylake, is working under 4GHz at 14nm process node. We can have more transistors on a single die, but we can not make the transistors switch faster. On the other hand, it is inefficient to invest all of the on-chip resources on a single processor core in terms of performance and energy. Given the same amount of resources, a 2-core processor can easily outperform a single core processor in many parallel applications.

As a result, the number of cores per processor continued to increase gradually through the years. From the industry side, we saw 2-core, 4-core and 8-core processors released to the market. From the academic side, 100-core, 1000-core processors are foreseen in the future landscape [1, 2]. Surprisingly, the number of the cores haven't changed for a long time since the first 8-core processor was introduced, as the dominant desktop processor currently available is still an 8-core processor. Does multi-core processor fail the future computer architecture design? We need to take one step back to analyze the performance model first for a better understanding.

Amdahl's Law [3] is introduced to explain how parallelism helps to improve the performance. *Amdahl's Law* separates the program execution into 2 parts, one part that can be parallelized and the other that can not. The overall speedup is then expressed by formula (1.1), where f is the fraction of the program which can be parallelized by a factor of S. Ideally, if the parallelized section takes no time to execute, it yields the second equation (1.2). Obviously, the roof of the speedup is constrained by the sequential code in the program.

$$Speedup(f, S) = \frac{1}{(1-f) + \frac{f}{S}} \tag{1.1}$$

$$Speedup(f, S)_{S \to \infty} = \frac{1}{(1-f)} \tag{1.2}$$

Amdahl's Law has been used to study the performance of multi-core processors extensively. In [4, 5], three different multi-core processor models were evaluated, namely, Symmetric, Asymmetric and Dynamic. Symmetric means all of the processor cores have the same resources. Asymmetric means a big core is coupled with several small cores. Dynamic means the processor can form a single super-core during the serial execution and can be decoupled to multiple cores during parallel execution. The conclusion of these papers are that even for a parallelism rich application, it is still worthwhile to build an "energy inefficient" big core, since every second it saves from the serial section can be translated to more parallel work in the rest of the "energy efficient" small cores. In typical parallel programs, the parallel part of the program is thoroughly optimized but the serial part remains on the critical path. Therefore, this dissertation focuses on the techniques to optimize the single-thread program performance in the superscalar processor.

Speculative execution is the key component for modern superscalar processors to reach high performance. Predicting branch direction and the target address [6, 7, 8] helps the processor to keep fetching subsequent instructions when the branch is not resolved. Value prediction [9, 10, 11] is used to overcome the data dependence limitation in exploiting instruction level parallelism (ILP). Memory dependence prediction [12, 13] is designed to bridge the in-flight stores and the loads. With the given progress of speculative execution, it is believed that more and other types of speculations will be involved in designing future architectures.

Although speculation is a powerful mechanism for exploiting ILP, if the speculation is wrong, the instructions fetched after the misspeculation have to be eliminated. The processor has to roll back to the misspeculation point and restart fetching from the correct path. There are two ways to improve the performance, either improving the speculation accuracy or recovering the misspeculations more quickly. The former approach has been broadly investigated and there is not much room left for improvement. For example, Geometric History Length branch predictor [14, 15] provides very high branch prediction accuracy and the memory dependence predictor [16, 17, 18] achieves very high memory dependence prediction accuracy. The latter approach, namely, recovering from a misspeculation has not been thoroughly investigated and this dissertation concentrates on techniques to recover misspeculations quickly and efficiently.

The rest of this chapter is organized as follows. Section 1.1 gives background information about superscalar processors. We then describe speculation mechanisms used in superscalar processors in Section 1.2. Widely used recovery mechanisms are discussed in Section 1.3. Finally, we give a taxonomy of recovery penalties in Section 1.4.

1.1 A Superscalar Processor

A typical superscalar processor is composed of two major parts: the front-end and the back-end, as shown in Figure 1.1. The front-end of the pipeline is in charge of fetching, decoding, renaming and dispatching. The back-end of the pipeline is in charge of instruction execution and retirement. The instructions which are in-flight in the pipeline (front-end and back-end) are speculative. Roughly speaking, their results are invisible to the outside until they are retired. Thereafter, the results are committed to the in-order state which contains all the valid execution outcomes. The future state in the figure represents the state obtained by combining the in-order state with the speculative instructions left in the pipeline.

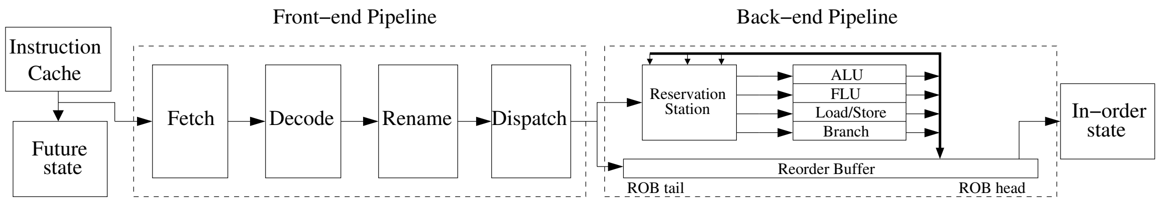


Figure 1.1: The block diagram of a superscalar processor

1.1.1 The Front-end

The front-end follows the program order so that the instructions flow sequentially similar to a queue. The fetch unit first fetches instructions from the instruction cache. The fetched instructions are then decoded. For a fixed-length instruction set, the decode stage is very straightforward. For a variable-length instruction set, decoding is much more difficult. Since the length of each instruction varies, most processors require a multi-cycle decode stage. The next stage is renaming and both instruction operands and destination are mapped to the physical space from the logical space. The details of the renaming process is explained later in Section 1.1.3. The last stage dispatches instructions to the corresponding components in the back-end.

1.1.2 The Back-end

The back-end of the pipeline is constructed to execute instructions in an out-of-order manner to exploit Instruction Level Parallelism (ILP). An instruction from the front-end is assigned to Reservation Station (RS) and Reorder Buffer (ROB) in parallel. RS acts like a shelf which maintains all the unexecuted instructions. Once the operands of an instruction are ready, the instruction is woken up to execute. Therefore, the execution obeys the data dependence and may change the original program order.

The ROB maintains the instruction state before they are retired. Instructions are allocated to ROB positions following the program order and they retire in program order as well. However, only completed instructions are allowed to retire. When an instruction is executed, its completion bit in ROB is set. Hence, an executed instruction is not retired until all preceding instructions are retired.

1.1.3 Register Alias Table

Before an instruction is dispatched to the back-end, it has to be renamed. Every destination logical register is renamed to a dedicated physical register in order to eliminate false data dependencies. A simple example is illustrated in Figure 1.2.

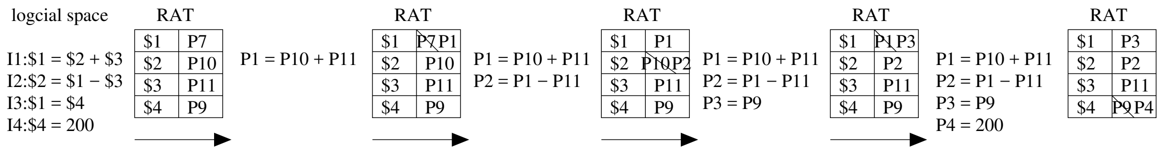


Figure 1.2: From logical register space to physical register space

The rename process consists two steps. First, the operands are renamed by reading the Register Alias Table (RAT). Second, the destination register of the instruction is assigned a new physical register from the free register pool and the new mapping is updated in the RAT. In the figure, the first instruction reads the current mappings of *$2* and *$3* and the instruction's destination register is assigned to *P1*. The updated mapping of *$1* is written back to the RAT to replace the old one (*P7*). The following instructions can find the correct physical register of *$1* through RAT. The fully renamed code is shown on the right of the figure as well as the corresponding RAT contents.

1.2 Speculative Execution

Speculative execution is a key component of superscalar processors. Generally speaking, an instruction's result is predicted upfront before its execution. Therefore, the dependent instructions are executed earlier with this prediction. In this work, two major speculation techniques have been evaluated, branch prediction and memory dependence prediction.

1.2.1 Branch Prediction

The branch result is unknown until it is resolved. However, the fetch unit needs to know the result immediately in order to fetch the subsequent instructions. Consequently, the branch is predicted once it is fetched. A common superscalar processor has a branch predictor to predict the direction of the branch instruction and a Branch Target Buffer (BTB) to predict the target address when the branch is predicted to be taken.

Typically the branch predictor considers the history information to make the prediction. Both the local history and the global history are considered. The local history can simply be the Program Counter (PC) of the branch or it can be a recorded history of this branch. The global history is usually the most recent branch predictions.

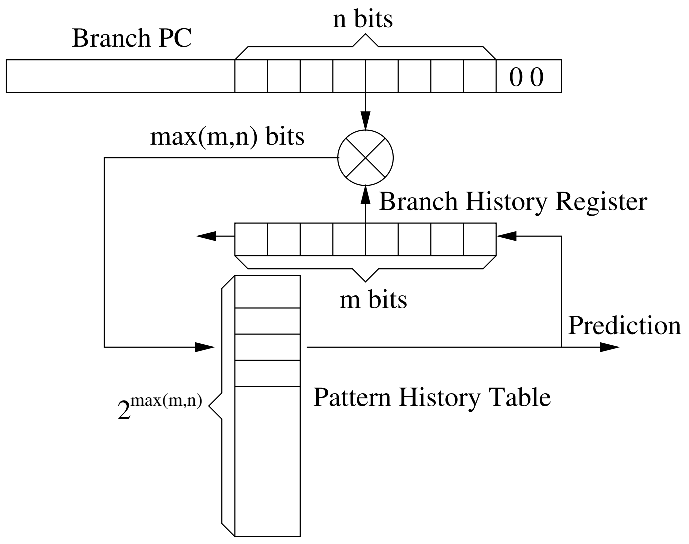


Figure 1.3: Gshare branch predictor

Figure 1.3 shows the structure of a correlating branch predictor called **Gshare** branch predictor. The Pattern History Table (PHT) is used to provide the prediction and it is indexed by exclusive ORing the branch PC and the Branch History Register (BHR). BHR keeps the recent branch prediction history. Every time a new prediction is made, the prediction is pushed into the BHR and the oldest prediction is eliminated. When a branch is mispredicted, the corresponding entry in PHT is updated. More advanced branch predictors are invented to consider a much larger number of history bits [19, 20].

The Branch Target Buffer (BTB) [21] provides the target address when the branch is predicted taken. Figure 1.4 illustrates a simple BTB design which works as a cache. The low bits of the branch PC are used to index BTB and the corresponding

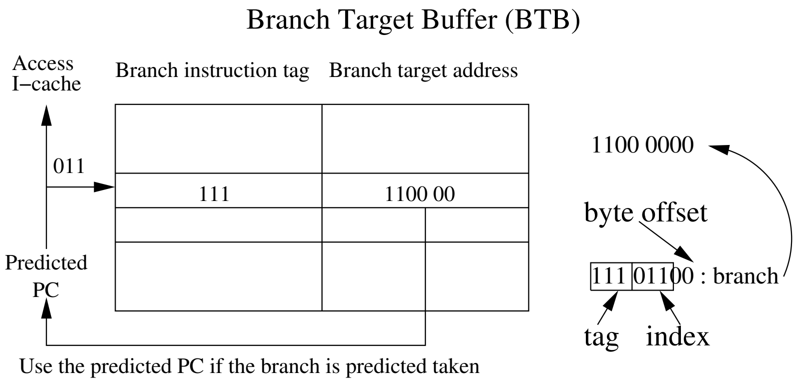


Figure 1.4: Branch Target Buffer

branch instruction tag entry contains the high bits. If the branch PC hits in BTB, the corresponding branch target address is read to fetch the subsequent instructions. In the figure, the branch located in *1110 1100* is predicted to jump to address *1100 0000*.

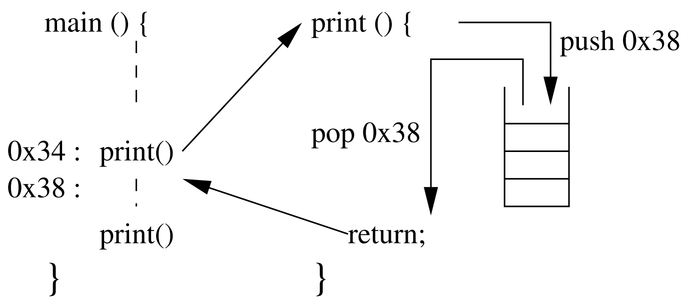


Figure 1.5: Return Address Stack

Other than the BTB, another dedicated component is designed to predict the return address of the function call, namely the Return Address Stack (RAS). Figure 1.5 shows a simple program in which the function “print” is called twice at different places. The return instruction in “print” has a single PC but two target addresses. A BTB does not work well when the target address varies and the RAS is designed to address this issue. When the first “print” is called, the PC of the next instruction, *0x38*, is pushed into the RAS. When the return instruction is executed, the target address is popped out of the RAS. Since the RAS is a stack, it always provides the return address of the most recent function call.

1.2.2 Memory Ordering Prediction

Memory instructions are different from other instructions as the data dependencies are unknown until load and store are computed. A load instruction is dependent on a prior store when they access the same memory address. However, in superscalar processors, instructions are executed in an out-of-order manner which makes memory disambiguation very difficult.

For the purpose of bridging the in-flight stores and loads, a typical superscalar processor contains a store queue and a load queue. The store queue maintains all the in-flight stores in program order. Once a store’s operands are ready, the store address and the store data are written into the corresponding entry in the store queue. The load queue maintains all the in-flight loads in program order. When a load is executed, the load address is written into the corresponding entry in the load queue.

When a load is executed, it simultaneously sends its address to the cache and the store queue. If the store queue does not have any store instruction matching the address, the data read from the cache is forwarded to the load. Otherwise, the closest preceding store's data is used.

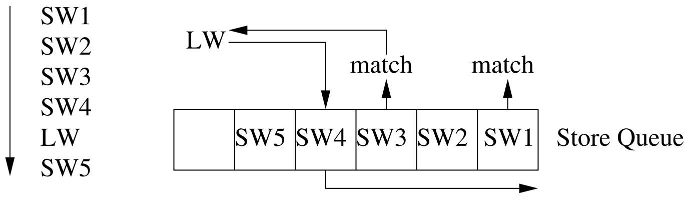


Figure 1.6: Store Queue

Figure 1.6 shows a simple example about how a load obtains its value from the store queue. The left side of the figure lists all the memory instructions in program order. When the load is decoded, the entry of the last store is kept with the load. In this example, **LW** knows the last store is **SW4** so the load only searches the stores which are older than **LW** (**SW1**-**SW4**). The youngest store is selected if there are multiple matching stores. In the figure, if both **SW3** and **SW1** match the address of **LW**, then **SW3** is selected. On the other hand, when a load is executed before a prior matching store updates the store queue, the load may read a wrong value and broadcast it to the dependent instructions. This load and the following instructions need to be re-executed. The memory ordering violation is detected through the load queue.

Figure 1.7 illustrates a load queue. The left side of the figure shows the original instructions following the program order. When **SW** is decoded, the location of the most recent load, **LW2**, is kept with the store. When a store updates the store queue,

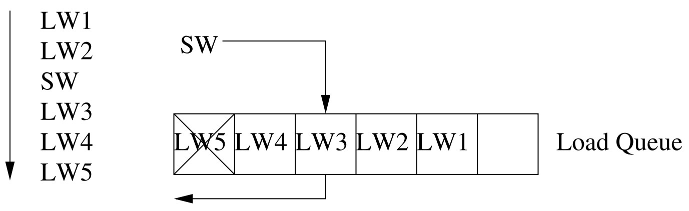


Figure 1.7: Load Queue

it only searches the loads which are younger than **SW** in the load queue. Any load instructions that executed before the store and had the same address violates the memory ordering. In this example, **LW5** which has the same address is younger than **SW** and had executed before **SW**. Therefore, this load has been misspeculated and a misspeculation recovery needs to be performed.

If a load executes without considering the prior stores, it is possible to have a memory ordering violation. However, if the load is forced to wait for all prior stores to execute, the resulting delay will be detrimental to the performance. Therefore, a feasible solution is to predict the dependencies for memory instructions. Roughly speaking, load instructions are blindly executed at the beginning assuming they are independent of prior store instructions. Once a dependence violation is detected, a correlation between the load and the conflicting store can be established. Thereafter, the load can wait for this particular store.

There are many mechanisms which maintain the correlations among load and store instructions, one such mechanism is the Store Set algorithm [16]. Store Set algorithm renames memory instructions using Store Set ID (SSID). If the store and the load had collided before, they are assigned the same SSID. The load will have to wait if any prior store with the same SSID has not executed.

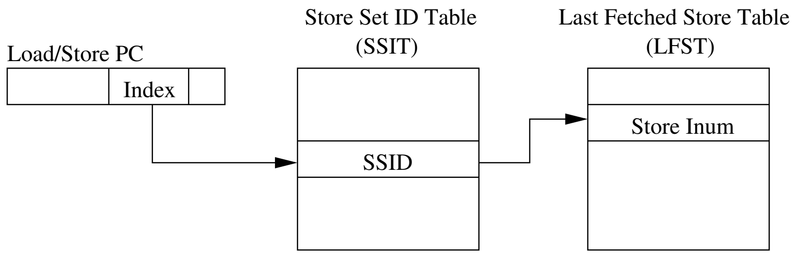


Figure 1.8: Store Set mechanism

Figure 1.8 illustrates the Store Set mechanism. Both store and load instructions get their SSIDs by reading the Store Set ID Table (SSIT). The store instruction uses its SSID to write a unique identity called Store Inum in the Last Fetched Store Table (LFST). The Store Inum is a hardware pointer such as the ROB entry number. The load instruction uses its SSID to read LFST to check if there is an in-flight store with the same SSID. If the store exists, the load can find the store using its Store Inum. Otherwise, the load can execute whenever the load address is known.

1.3 Misspeculation Recovery

Not every speculative execution is correct. Once a misspeculation happens, the recovery process is initiated: I) The branch predictor's state is restored back to the misspeculation point so that the correct instructions can be predicted properly; II)

The fetch engine redirects to the misspeculation point and fetches the valid instruction; III) The Register Alias Table (RAT) is reverted back to the misspeculation point so as to correctly rename the valid instructions; IV) The resources which are occupied by the invalid instructions are released. We have discussed two major types of misspeculations, branch mispredictions and memory dependence misspeculations. Although the same recovery process can be utilized for both types of misspeculations, memory dependence mispredictions can be handled by re-executing instructions which received the incorrect values, whereas in case of branch misspeculations incorrectly fetched instructions need to be eliminated from the pipeline.

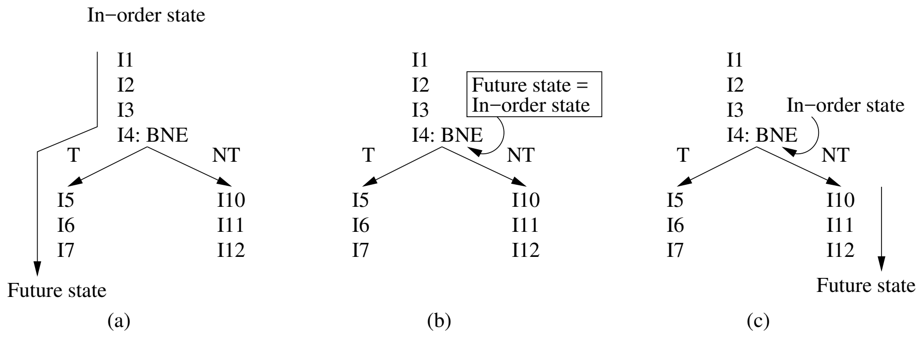


Figure 1.9: Branch misprediction recovery

Figure 1.9 shows a basic recovery process for a mispredicted branch. The process states described in Section 1.1 are also shown here. The in-order state represents the valid execution results. The future state represents the combination of the in-order state with the speculative state. In Figure 1.9(a), the branch **BNE** is predicted taken which turns out to be a misprediction. Therefore, instructions *I1-I4* are still valid but the following instructions *I5-I7* are invalid. The future state is damaged at this point, so the fetch engine stops. The valid instructions are allowed to retire and update the in-order state. When the last valid instruction *I4* is retired, the instructions left in the pipeline are all invalid and can be immediately eliminated. Simultaneously, the future state is restored back to the in-order state in Figure 1.9(b). Since the correct branch result is known, the fetch engine restarts to fetch the valid instructions *I10-I12* in Figure 1.9(c).

The basic recovery process for memory dependence misspeculation is similar. When a load instruction conflicts with a store instruction, the instructions before the load are still valid and retired. Remaining instructions are eliminated and the future state is rolled back to the misspeculation point. The misspeculated load is re-executed and will retrieve the correct value since all prior stores have retired.

1.3.1 Recovering Branch Predictor State

After a branch misprediction, the branch predictor would have been updated with the wrong values. Without correcting the branch predictor state, new branch predictions will be very inaccurate. Therefore, the first task in recovery process is to repair the branch predictor state. Typically, Pattern History Table (PHT) is designed to be updated at the retire stage by only valid instructions. Hence, PHT does not need to be modified. Since the Branch History Register (BHR) and the Return Address

Stack (RAS) are speculatively modified, these components need to be repaired.

Although the BHR keeps the m most recent predictions in order to predict the next branch, in a real hardware design, BHR is made larger so that it can be rolled back to a previous position.

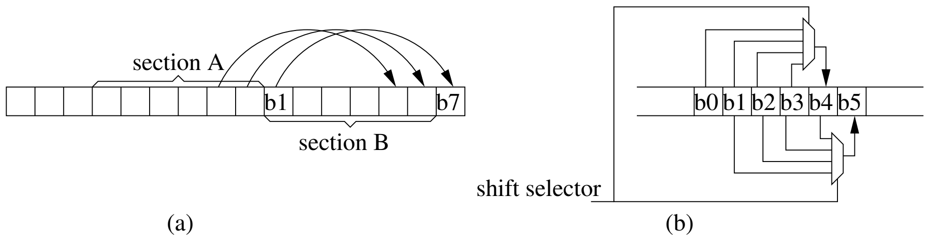


Figure 1.10: Recover Branch History Register

Figure 1.10(a) illustrates a BHR design in which the younger branches are listed to the right of the figure. When branch *b1* is predicted, section A in the BHR is the most recent branch history and this section is used to make a prediction. The following branches, *b2*-*b7*, are then predicted and pushed into the BHR. Branch *b1* is a misprediction, thus, the processor has to shift section A to the right in the BHR. Figure 1.10(b) shows a shifter design which can shift the BHR to the right by a variable number of bits.

The second part of the branch predictor recovery is the Return Address Stack (RAS) which is used to predict the return address from a function call. It is mentioned in [22] that it is possible to corrupt RAS during the misspeculation recovery. In most scenarios, a younger return instruction may overwrite an RAS entry which keeps another return address. Therefore, there is no way to roll back to a previous point during the recovery.

Dixon et al. implemented a practical solution to fix RAS. They used the same technique from Register Alias Table (Section 1.1.3) to map RAS entries to physical registers. By doing this, two different return addresses which are written to the same RAS entry are mapped to two different physical registers. In the recovery, the processor only needs to restore the correct mapping to fix RAS. The method to restore the mapping is similar to the RAT correction which is described in Section 1.3.2.

1.3.2 Restoring the Map Tables

We mentioned at the beginning of this section that the future state is restored by copying from the in-order state and RAT is repaired in this way. In most superscalar processors, there are two RATs serving different roles. The Front-end RAT (F-RAT) is located at the rename stage and is speculative. The Retirement RAT (R-RAT) is located at the retire stage which keeps the in-order state [23]. During the recovery, R-RAT is copied to F-RAT when the misspeculated instruction is retired. This mechanism is shown in Figure 1.11.

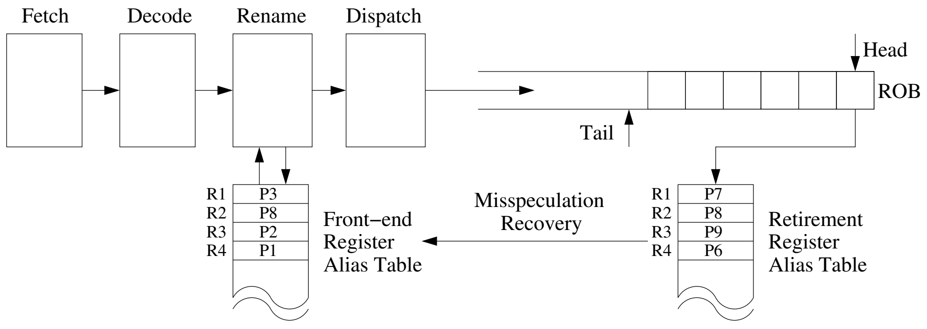


Figure 1.11: Recover Register Alias Table

1.3.3 Eliminating Stale Instructions

The instructions following the misspeculated instruction are marked as stale and cannot change the in-order state. The resources which were occupied by the stale instructions are released during recoveries. When a misspeculation is detected, the instructions left in the front-end are all stale. Consequently, the front-end is reset to eliminate all stale instructions. The situation in the back-end is different since it is mixed with valid and stale instructions. The basic recovery mechanism will not reset the back-end until all valid instructions are retired.

1.4 Recovery Penalty Analysis

We have described the basic recovery mechanism. In this section, the recovery penalty resulting from misspeculations is analyzed and a taxonomy of misspeculation recovery is given. This work divides the recovery penalty into different parts. These are the **misspeculation detection delay**, **stale instruction elimination delay**, **state restoration delay** and finally **pipeline fill delay**. For each part, we also give some of the related work in this section.

1.4.1 Misspeculation Detection Delay

A speculative instruction does not know its result until it is fetched, decoded, renamed and finally executed. Therefore, the misspeculation detection delay is the time spent for misspeculation to be detected since that misspeculation was fetched. In most cases, it is very difficult to reduce the misspeculation detection delay since the tasks associated with each instruction's execution cannot be reduced.

Armstrong et al. presented a mechanism to speculatively predict a misspeculation [24] by observing Wrong Path Events (WPE). WPE is classified into two categories: memory instructions and control flow instructions. Memory instructions include dereferences of NULL pointers, reads or writes to unaligned addresses, writes to a read-only page, reads from an executable page or TLB misses, etc. Control flow instructions include branch mispredictions and RAS underflow. When a WPE is observed, it is highly likely that there is a misspeculation. Therefore, even if the exact misspeculation point is not found, their mechanism can predict the misspeculation point and start to recover before the misspeculation is resolved.

1.4.2 Stale Instruction Elimination Delay

Section 1.3.3 has described a basic mechanism to eliminate stale instructions and release the occupied resources. This mechanism does not eliminate stale instructions until all the valid instructions are retired. If the retirement is halted by some long latency operations, such as a cache missing load instruction, the elimination delay may significantly affect the overall performance.

McIlvaine et al. designed a specific rename algorithm [25] to assign a dedicated **branch flush tag** to each branch. The tag is a bit vector which has only one bit set to high. Hence, the width of the bit vector is equivalent to the number of tags. Other than the **branch flush tag**, a **branch path tag** is also implemented. Each instruction has its own branch path tag indicating which unresolved branches it is dependent on. When a branch misprediction is detected, its branch flush tag is broadcast and the instructions which are dependent on this branch are selectively flushed.

1.4.3 State Restoration Delay

The basic recovery mechanism explained in Section 1.3 restores the future state by copying the in-order state and this is the state restoration delay. The retirement of the valid instructions also constrains the state restoration delay due to the same reason described before.

Checkpointing architectures [26, 27] were designed and developed to construct a larger instruction window. This mechanism also shortens the state restoration delay. When a branch is at the rename stage, a copy of the current F-RAT is kept as a checkpoint. If this branch is mispredicted, the checkpoint is used to instantly restore the F-RAT.

To keep a checkpoint for every branch is very costly. So a practical solution is to allocate a checkpoint only when a particular branch is likely to be mispredicted. A confidence predictor is used to predict how likely the coming branch is going to be a misprediction.

1.4.4 Pipeline Fill Delay

A misspeculation may cause the processor to eliminate a substantial amount of stale instructions. It takes many cycles to fill an empty pipeline and this is the pipeline fill delay. One of the solutions is to reuse the stale instruction computations.

Usually a program forks at the branch instruction and joins at a later point. The instructions fetched after the join point are control independent of the branch result. In other words, the control independent instructions are always executed regardless of the branch direction. In consequence, if the branch is mispredicted, some of the control independent instruction results can be reused. Such reuse mechanisms exploit Control Independence [28] of instructions.

1.5 Summary

In this chapter, we went over the basic components of a superscalar processor and explained the speculative execution including branch prediction and memory dependence speculation. We also elaborated in detail, a basic recovery mechanism and outlined the recovery process to restore branch predictor, return address stack and the front-end RAT. We also provided a taxonomy which classifies the recovery penalty into four parts: **misspeculation detection delay**; **stale instruction elimination delay**; **state restoration delay**; **pipeline fill delay**. In the rest of the dissertation, the proposed mechanisms are illustrated and analyzed thoroughly to address each of these delays.

Chapter 2

Mower : A Walking-based Misspeculation Recovery Mechanism

2.1 Overview

In the previous chapter we have described a basic recovery mechanism and presented a taxonomy of misspeculation recovery penalties. In this taxonomy, one of the delays is the **state restoration delay**, such as restoring the correct F-RAT. The basic

[0]The material contained in this chapter was previously published in Proceedings of the 29th ACM on International Conference on Supercomputing (ICS ”15) [29]. The copyright permission is listed in Appendix A.

recovery mechanism outlined in Chapter 1 does not recover F-RAT until all the valid instructions are retired. Therefore, the state restoration delay is highly dependent on the speed of valid instruction retirement. If the retirement is impeded by some long latency operation then the misspeculation penalty is exacerbated.

For the purpose of breaking the dependence between the recovery penalty and the valid instruction retirement, many innovative techniques were proposed. These techniques can be classified into two categories: walking and checkpointing. We briefly review both techniques in this section and bring in the concept of Mower, a walking-based recovery mechanism.

2.1.1 Recovering F-RAT by Walking

F-RAT contains the correct mapping table at the misspeculation point and the updates done by the instructions after the misspeculation point. A easy approach to restore F-RAT is to undo all the updates after the misspeculation. This mechanism acts like a walker walking back toward the misspeculation. When it is walking, any updates made by instructions to F-RAT is undone.

Moudgill et al. first designed a register renaming algorithm which is similar to the one widely used in contemporary superscalar processors [30]. Their work consists of a register mapping algorithm, a physical register release algorithm and an algorithm to handle precise interrupts. In order to recover F-RAT, a **history buffer** is implemented to keep the old physical register mapping of each instruction. During recovery, correct mapping table is restored by using the **history buffer** contents.

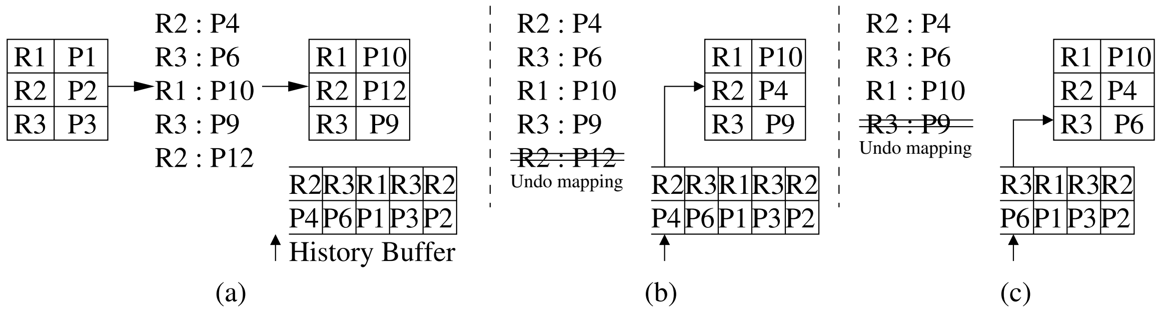


Figure 2.1: Restoring the mapping table by history buffer

Figure 2.1 is a simple example demonstrating how the history buffer mechanism works. Initially, **R1**, **R2** and **R3** are mapped to **P1**, **P2** and **P3** respectively in Figure 2.1(a). Whenever an instruction is renamed, its newly assigned physical register modifies the mapping table. Simultaneously, the old physical register number is pushed into the **history buffer**. For example, the first instruction maps **R2** to **P4** and the old mapping, **R2** to **P2**, is pushed into the **history buffer**. The resulting mapping table and **history buffer** contents are depicted on the right side of the figure.

When a misspeculation is detected, the walking process is triggered. Since the walking process operates in the reverse order, the update done by the last instruction is undone first as shown in Figure 2.1(b). The **history buffer** works like a stack and the last pushed content is popped out first. Therefore, the mapping of **R2** is restored to **P4**.

Walking on the **history buffer** eventually restores F-RAT back to the misspeculation point. The time consumed by the walking process is dependent on the number of instructions between the misprediction point and the end of the buffer.

As it can be seen, in case of **history buffer**, walking is done from F-RAT to the misspeculation point. On the other hand, it is also possible to “walk” from R-RAT (Retirement RAT) to the misspeculation point [26, 27]. In this case, the walker needs to have its own RAT, W-RAT(Walker RAT) and copies R-RAT to W-RAT at the beginning of the recovery. Thereafter, the walker goes through the valid instructions from the ROB head to the misspeculation. When an instruction is scanned, its current physical register rather than the previous physical register is used to update W-RAT. At the end, W-RAT has the correct mapping table, which is copied back to F-RAT.

As we have previously mentioned, the overhead of any walking mechanism depends on the number of instruction to scan. These two walking mechanisms which starts from F-RAT or R-RAT are selected due to the misspeculation position. In terms of state, if the misspeculation point is closer to F-RAT, the walker starts from F-RAT. Otherwise, the walker starts from R-RAT.

2.1.2 Checkpointing Architectures

An essential task for every recovery mechanism is to restore the future state back to a previous known state. Walking mechanism accomplishes this by undoing the incorrect updates. Checkpointing, on the other hand, can revert the future state back to any point as long as it has a checkpoint of the processor state at that point.

Hwu et al. proposed to use checkpoints for repairing misspeculations [31]. They also described the usage of **history buffer** to do a low overhead recovery. Akkary et al. designed an ROB-free architecture which uses checkpoints to recover from misspeculations [26, 27]. In order to better utilize resources, a branch confidence predictor can be implemented to allocate checkpoints for the branches which are likely to be mispredicted.

Other than a pure checkpointing architecture, many ROB-based architectures also incorporate a checkpoint-based design to optimize the state restoration delay. The MIPS R10000 [32] and Alpha 21264 [33] processors integrated checkpoints to do a fast repair of F-RAT.

Cristal et al. further optimized the checkpointing architecture [34, 35]. In their technique, a pseudo-ROB was implemented to reduce the number of necessary checkpoints. This pseudo-ROB could retire instructions disregarding their completion status and only assigned a checkpoint if a retired branch was not executed.

Zhou et al. proposed a special mechanism to allocate checkpoints at the misspeculation detection time [36]. Later when the correct F-RAT is restored, the difference between the correct F-RAT and the checkpoint is selected out which is the incorrect mapping updated by the invalid instructions. Additional *MOV* instructions are finally inserted to transfer the correct value to the physical registers mapped by the checkpoint. Since the incorrect mapping is marked ahead, this mechanism does not have state restoration delay.

BranchTap [37] was designed to allocate checkpoints efficiently in a different way. BranchTap would throttle the fetch stage when the number of unresolved, low confidence branches without checkpoints exceeds a threshold. The insight of their mechanism is as follows: the penalty to recover a misspeculation without a checkpoint is higher than the one with a checkpoint. Thus, the number of unresolved branches without checkpoints should be controlled since their recovery would do more damage to the performance. Turbo-ROB [38] was later developed to better recover a misspeculation without a checkpoint. The mechanism packed the ROB storage so that only the necessary mapping information is kept in Turbo-ROB. By doing this, the walking distance became much shorter. As a result, a much quicker recovery is achieved. A similar approach was also presented in CROB [39].

Other than compressing ROB to shorten the walking distance, Golander et al. designed an approach to reuse the instructions on the walking path [40]. Their mechanism can reuse the results of instructions and the outcome of branches obtained during the first run. They then designed an enhanced checkpoint allocation algorithm [41] which would allocate checkpoints according to dynamic events, such as second-level cache misses and rollback history.

Another cost efficient mechanism, CPROB [42], was proposed to release the physical registers belonging to a checkpoint even before the corresponding branch is resolved. In which case, the physical registers are released much earlier at the cost of a potentially more costly misspeculation recovery.

2.1.3 Motivation

Walking is a very general mechanism. Given the contents of **history buffer**, the processor can travel back in time to any point as long as the related record is still in the **history buffer**. The drawback of this approach is that the recovery delay is highly dependent to the walking distance. Furthermore, **history buffer** size restricts the number of speculative instructions. On the other hand, checkpointing based architectures can recover instantly when a misspeculated instruction has a checkpoint.

Even if the misspeculated instruction does not have a checkpoint, the recovery can start from a position closer to the misspeculation and therefore have a shorter walking distance. The shortcoming of this technique is the extra overhead in terms of hardware cost and energy consumption. For each checkpoint, a full copy of F-RAT is required and each allocation of the checkpoint consumes extra energy. As a result the mechanism is not efficient.

Mower is designed to address these problems in an efficient way. It is a walking-based algorithm so there is no checkpoints need to take, saving energy and design cost. No checkpointing means no instant F-RAT recovery, but that does not mean a longer state restoration delay. Mower has two unique features to curtail the delay. One is that it can rename instructions even before F-RAT is fully restored. The other one is that Mower can fully restore F-RAT in a much shorter distance than a conventional walking algorithm.

F-RAT used in Mower is different in that each logical register's mapping is treated as an individual entity. Therefore, even when the F-RAT is not fully recovered, the renaming continues as long as the instruction's operands have the correct mapping. One simple example is shown in Figure 2.2. The branch is predicted to take the left path so that **R3** and **R4** update F-RAT. When the misprediction of the branch is detected, the mapping in F-RAT updated by invalid instructions are marked to be invalid (**R3** and **R4**). The rest of the map table is still valid. The first newly fetched instruction reads the mapping of **R1** and **R2** which, in this example, are still valid. In other words, this instruction is allowed to go through the rename stage. The next instruction which reads the mapping of **R3** is stalled as the mapping is invalid at that point.

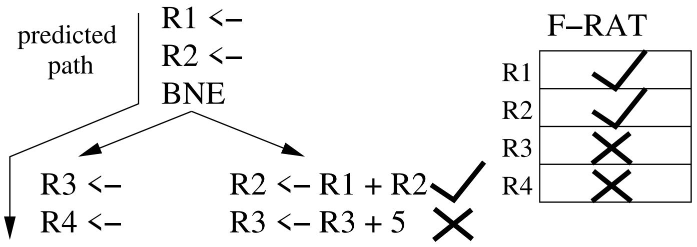


Figure 2.2: Renaming with a not fully correct F-RAT

With the knowledge of F-RAT changes in Mower, it becomes much easier to explain the walking process that is employed in Mower. A conventional walking process is shown in Figure 2.3(a). For simplicity, only **R3**'s updates are shown in the figure. When a misspeculation is detected, the walking process will undo the updates in reverse program order. Whenever an **R3**'s entry is encountered, F-RAT is changed. The correct mapping of **R3** is not determined until the walking process is finished since the walker has no knowledge which logical register will be changed next. At the end, F-RAT is fully recovered.

It is easy for Mower to have a short-cut because Mower can identify the valid mapping at a finer grain. Figure 2.3(b) shows the walking process of Mower in which the walker follows the program order starting from the misspeculation. When **R3** is encountered

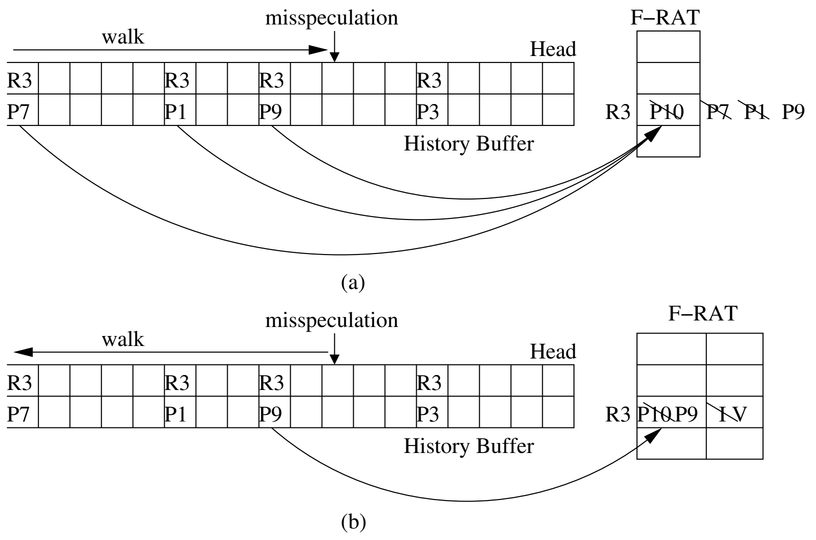


Figure 2.3: The different walking direction in Mower

the first time, Mower checks F-RAT and finds that it is an invalid mapping. Hence, Mower updates **R3** with **P9** and sets the mapping to valid. Because the mapping becomes valid, it is not changed any more for the rest of the walking process. The insight is, if the walking process follows the program order, the first entry of the invalid mapping register contains the correct mapping, since this entry holds the update made before the misspeculation.

If we consider both of the algorithms described above, it becomes clear that Mower can recover the register mapping much faster because each mapping has its own valid bit. Besides, when a single register mapping is recovered, it helps to rename the instruction which reads this logical register and everything is done without a fully recovered F-RAT.

2.2 Control Dependence Tracking

When a misspeculation is detected in Mower, invalid register mappings have to be identified immediately since the walker needs the validation information to recover. Thus in Mower, every branch is given a special tag. When the branch is mispredicted, the branch tag is broadcast and the affected register mappings are marked to be invalid. In this section, the related structures are thoroughly explained.

2.2.1 Branch Renaming

For every decoded branch, a special tag, branch flush tag, is allocated. A normal rename policy such as the register renaming algorithm does not fit well to rename branches. This is because, a single instruction may depend on multiple unresolved branches and this one to N mapping does not exist in register renaming.

McIlvaine et al. had a patent [25] to rename branches and their approach can solve this one to N dependence tracking effectively. The branch flush tag is a bit vector which has only one bit set high. Therefore, the width of the vector determines the total number of flush tags. For example, if the vector is 4-bit wide, the complete tag set is 1000, 0100, 0010, 0001. In other words, four branches at most can be renamed.

When a branch is executed, its flush tag is not needed any more so the tag is released and the tag can be allocated to a new branch. In summary, the branch flush tag's width decides the number of unresolved branches.

2.2.2 Disambiguating the Branch Ordering

In general, there are multiple unresolved branches in the pipeline and it is possible that more than one branch is mispredicted. Consequently, when a recovery process is underway, another misprediction can be detected. At that moment, the branch ordering is very important. An older misprediction should overwrite the current recovery process and a younger misprediction should be ignored. Normally, a superscalar processor would compare the sequence number of the branches to decide the branch age [43]. Instead, each branch instruction in Mower has a dedicated tag. This approach enables the use of a control dependence matrix to implement proper branch ordering.

Such dependency matrices were first proposed to replace the CAM-based reservation stations [44, 45]. Each row in the matrix represents an individual instruction and each column represents a dependent component. Whenever all the dependencies are cleared in one row, the corresponding instruction is ready to issue.

In Mower, a **Branch Dependence Matrix** (BDM) is used to identify the ordering of the branches. Figure 2.4(a) shows a simple example in which there are three branches renamed to "001", "010" and "100". The branch dependence matrix shown on the right side is a 3X3 matrix. Each row represents a branch and the values in the row represent the dependencies of this branch. If the value is high that means the branch is dependent on another branch which is on that column. For example, branch **B2** is dependent on **B1** as **B2** is fetched after **B1**. In the matrix, the row of **B2** is "001" and the corresponding column is **B1**. Branch **B3** is dependent on both **B1** and **B2**, its row then is "011".

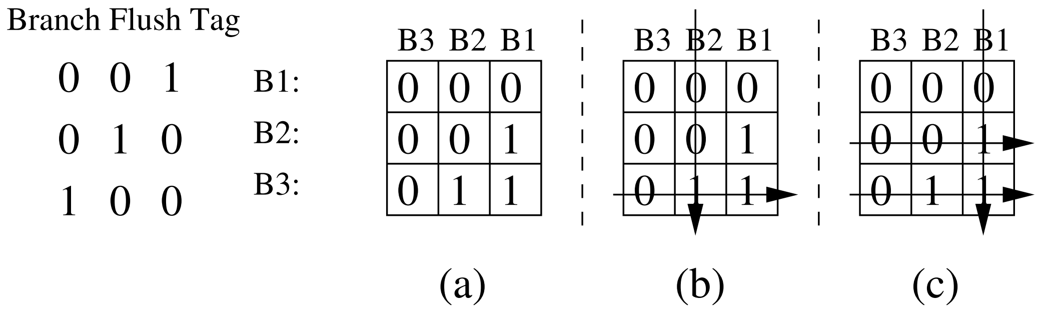


Figure 2.4: The branch dependence matrix

When a branch instruction is mispredicted, its flush tag is broadcast to invalidate those branches which were fetched afterwards. For example, in Figure2.4(b), branch **B2** is mispredicted and its column is invalidated. Wherever the entry is set high, the corresponding row will be invalidated. In this example, branch **B3** is invalidated. When **B1** is mispredicted, both **B2** and **B3** are invalidated (Figure 2.4(c)). Only a valid branch instruction can trigger a misprediction recovery.

2.2.3 Tracking Affected F-RAT entries

The dependence matrix can also be used to track the register mappings. As we have discussed before, Mower treats each logical register mapping as an individual entity. When a misprediction is detected, the mapping which is updated after the misprediction is invalidated. This is accomplished by using another dependence matrix.

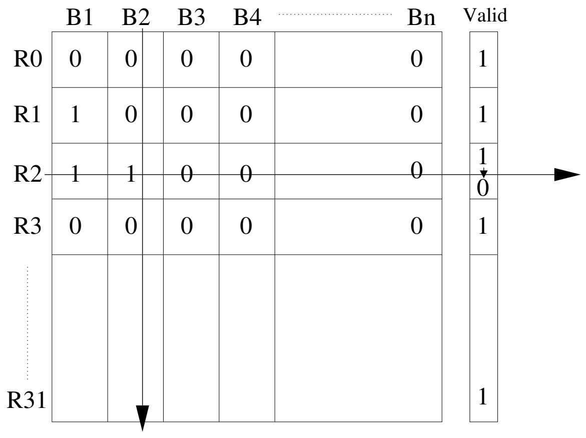


Figure 2.5: The mapping dependence matrix

Figure 2.5 shows a **Mapping Dependence Matrix** (MDM) which has 32 rows and n columns. Each row represents a logical register's dependencies and each column represents an unresolved branch. Thus at most n unresolved branches are included. When a logical register is renamed, all unresolved branch tags are written into the corresponding register row and the row is also marked as valid. Whenever a misprediction is detected, the corresponding column is broadcast and any row which has the dependence bit set is invalidated. In Figure 2.5, the mapping of **R2** is invalidated when branch **B2** is mispredicted.

2.3 A Reverse Walking Procedure

The walker in Mower has two major tasks during the recovery procedure. First, it scans the history buffer to restore F-RAT and second, it eliminates the stale instructions and reclaims the occupied resources. The walking process starts from the misprediction and moves towards the ROB tail. This walking direction is opposite to the previous walking mechanisms and it accelerates the F-RAT recovery.

2.3.1 Recovering F-RAT in a gradual manner

Once a misprediction is detected, MDM is searched and the register mappings which were modified after the misprediction are marked invalid. Figure 2.6 illustrates this recovery procedure. The instructions are placed in program order from left to right. The register mappings (from logical registers to physical registers) are shown in the first two rows. The row of “Previous” contains the previous physical register identity of the logical registers. For example, **R1** is mapped to **P6** in the first instruction and then mapped to **P8** in the 4th instruction. The previous register identity **P6** is kept in the 4th instruction. This approach works similar to a history buffer.

The register mappings of **R1** and **R3** are modified after **B1**, thus these two mappings

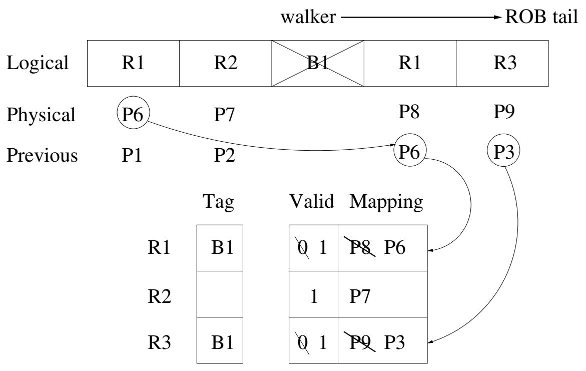


Figure 2.6: Recovering F-RAT by walking from the misprediction to ROB tail

are marked invalid before the walking process by broadcasting through MDM. When walker scans the invalid instructions, it undoes the updates with the previous physical registers. **R1**'s mapping is substituted with **P6** and this is the up-to-date definition before the misprediction. Since **R1**'s mapping has changed, its valid bit is set to high which means this mapping will not be modified by the walker during the recovery. This mechanism is also explained in Figure 2.3.

2.3.2 Eliminating Stale Instructions

In the back-end of the processor, the resources occupied by invalid instructions have to be released. Mower accomplishes this task when the walker scans the invalid instructions. The invalid instruction may be in one of the pipeline stages in the back-end. If the instruction has not been executed yet, it is still in the reservation station. If the instruction is woken up but not completed, it is in the execution unit.

A memory instruction always occupies an entry in LSQ. Based on the instruction state, a different elimination operation is triggered. Therefore, Mower keeps these states in ROB to track the resources occupied by every instructions.

If the scanned instruction holds a physical register, the physical register should be reclaimed as no valid instruction will read its value. The register number is pushed into the free register pool so that it can be reused by incoming instructions.

When the walker scans an ROB entry, it first checks the state of the instruction. The completed instructions are cleared from the ROB entry and if it is a memory instruction, the LSQ entry number is read to eliminate the instruction from LSQ as memory instructions are not removed from LSQ until they are retired. If the instruction has not issued yet, the corresponding RS entry number is read to eliminate it from RS. Other than these two cases, the instruction must have been issued but the computation has not finished yet. A naive solution is waiting for the computation to complete. The drawback of this solution is the potential long latency due to the operation time. Some of the floating point operations may take up to tens of cycles to complete. A more common case is a cache miss load which may end up with a delay of hundreds of cycles. In order to have a quicker walking process, the walker marks the destination physical registers of these instructions as invalid and skips these instructions. These invalid instructions are still allowed to execute, but when they complete, their physical registers are reclaimed instead of writing back their results.

Note that these physical registers are not reclaimed during the walking process but rather at their computation time.

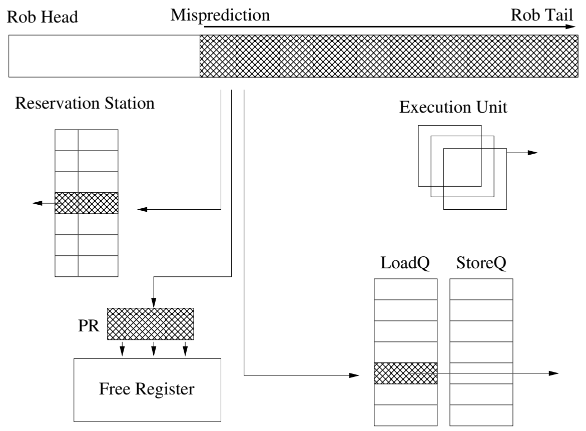


Figure 2.7: Reclaiming the back-end resources via the walking process

Figure 2.7 shows the mechanism to reclaim resources. The instruction state related information is held inside ROB such as the issued bit, the completion bit and the corresponding entry number.

2.4 Microarchitecture

In this section, a detailed hardware design of Mower is given. Figure 2.8 demonstrates the block diagram of Mower and distinguishes it from a normal superscalar processor. Mower has a valid bit for every register mapping in F-RAT. It scans the ROB entries to reclaim resources and eliminate invalid instructions.

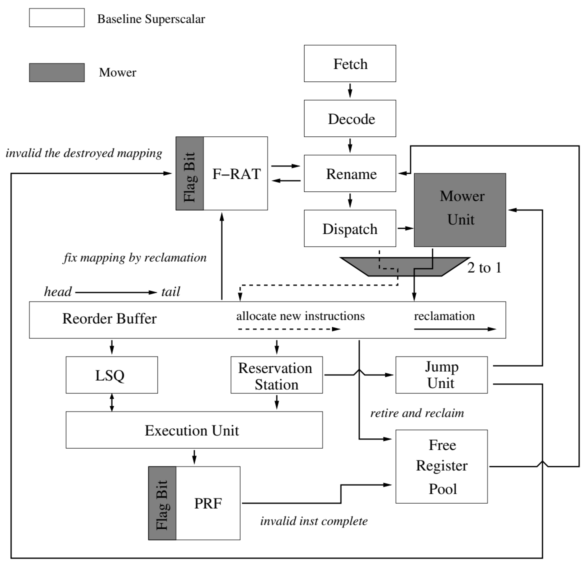


Figure 2.8: The block diagram of Mower

2.4.1 Branch Tag Allocation and Release

Mower renames branches with branch flush tags. Each branch has a dedicated flush tag which is composed of a bit vector. When all of the flush tags are allocated, the processor has to stop if it encounters another branch instruction. Whenever a branch is resolved, its flush tag is reclaimed irrespective of the outcome, whether it is mispredicted or correctly predicted.

The tag allocation happens at the rename stage. Another global structure, Branch Dependence Register (BDR), is also located at the rename stage which tracks all of the unresolved branch flush tags. Whenever a new flush tag is assigned, the corresponding bit is set and the bit is reset when the branch is resolved.

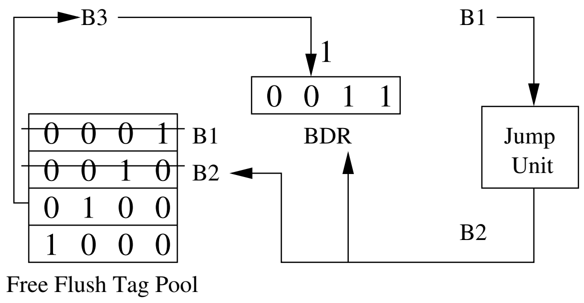


Figure 2.9: Branch Dependence Register, tracking all of the unresolved branches

Figure 2.9 shows a complete example about how a branch flush tag is assigned and released as well as the operations of BDR. Branch **B3** arrives at the rename stage and obtains a flush tag (0100) from the free flush tag pool. Simultaneously, this tag is written into BDR to indicate a new unresolved branch. Note this BDR update is a bit operation. Thus, only one bit is set to high. Branch **B2** is resolved and its tag (0010) is pushed into the free flush tag pool. The corresponding bit in BDR is also reset. Finally in the next cycle, the BDR value is 0101 which means **B1** and **B3** are unresolved. Branch flush tags 0010 and 1000 are available.

2.4.2 Branch and Mapping Dependence Matrices

The BDR holds the tags of all the unresolved branches, thus the BDR value is updated to the value in BDM when a branch is renamed. Figure 2.10 shows a simple example about how the BDR works with the BDM. In the first part, Branch **B1** is renamed to 0100. But before the tag is updated to BDR, the current BDR is written into BDM in the row of **B1** indicating its branch dependence. This is because there is only one unresolved branch 0001 on which **B1** is dependent. Next, the BDR is updated to 0101 after **B1** is renamed.

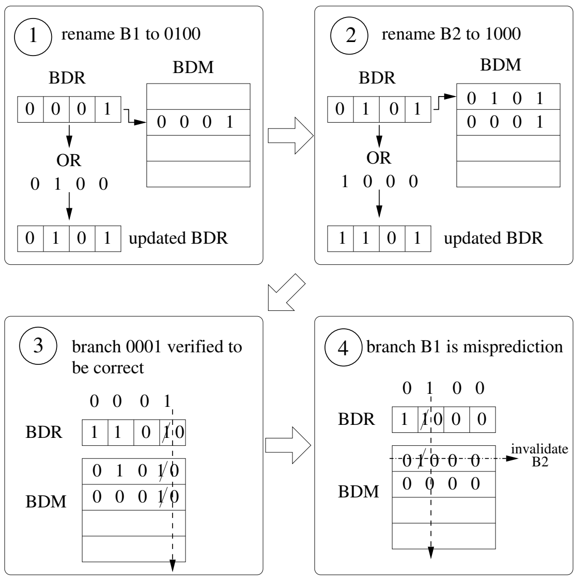


Figure 2.10: Branch Dependence Register collaborates with Branch Dependence Matrix

In the second part of the figure, branch **B2** is renamed and the current BDR is written into the row of **B2**. Part three shows the first branch 0001 is resolved and the tag is reclaimed. Part four depicts that branch **B1** is resolved and it is a misprediction. Therefore its flush tag is reclaimed and branch **B2** is invalidated.

MDM works similar to BDM in that the mispredicted branch broadcasts its flush tag and invalidates the mappings which are dependent on this branch. During the walking process, the walker scans through the invalid instructions. If it is an incomplete branch instruction, its branch flush tag is reclaimed as well. Therefore the tag is broadcast through BDM and MDM to clear the dependence.

2.4.3 Reclaiming the Resources using a single Read Port

In general, walker needs an access port to read the instruction state from the ROB such as the logical register and physical register number, the completion bit and the entry number of RS. A conventional superscalar processor typically implements two ports, one is for the ROB tail to allocate new instructions and the other is for the ROB head to retire completed instructions. If the walker is implemented with a dedicated port, then this may result in a very complex hardware design and can potentially affect the cycle time.

As it is known, the front-end of the pipeline is flushed at the beginning of the recovery. Therefore, the ROB tail will not allocate any instruction until the first newly fetched instruction arrives at the dispatch stage. In other words, the port which is occupied by the ROB tail is free and can be leveraged by the walker. It is possible that the number of invalid instructions is too large that the walking process is still in progress when the ROB tail needs to allocate new instructions. When this happens, Mower can still share the single port in a time sharing fashion. The first cycle, the walker uses the port and the next cycle, the ROB tail uses the port. Since the total number of invalid instructions is limited (no more than the ROB size), the sharing phase will not last too long.

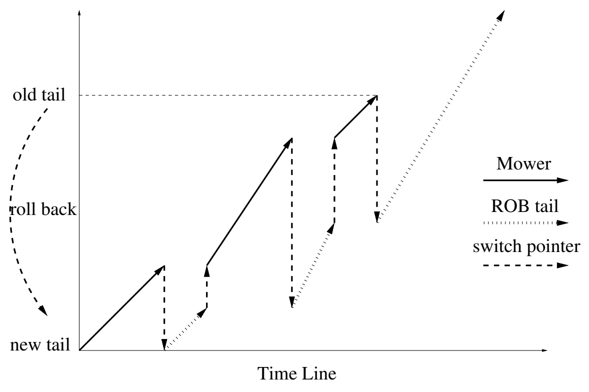


Figure 2.11: Sharing of one access port by the walker and the ROB tail

Figure 2.11 illustrates the sharing operations through the timeline. The walker is initially at the point of the misprediction and scans towards the old ROB tail. When the new instructions arrive at the dispatch stage, the control of the port is switched to the ROB tail, thus the dashed line indicates that new instructions are dispatched to the ROB entries starting from the point of the misprediction. Thereafter, the control of the port is switched between the walker and the ROB tail until all the invalid instructions are eliminated by the walker, then the control of the port is exclusively turned to the ROB tail.

2.5 Evaluation

The evaluation of Mower shows that it can outperform other recovery techniques in an effective and efficient way. In this section, the evaluation methodology and a detailed analysis of the results are provided.

2.5.1 Evaluation Methodology

The Architecture Description Language (ADL) [46] is used to design the processor models. ADL is a language designed to simulate modern processor performance. It gives researchers the flexibility to design their own ISA (Instruction Set Architecture). With the given ISA, researchers can use ADL to develop the processor model and generate the simulator, assembler and disassembler in a simple step.

In this work, we choose to use MIPS-I as our simulation ISA [47]. It is a 32-bit RISC instruction set which has 32 integer registers and a co-processor to deal with floating point computation. The delay slot of the branch instructions is removed to simplify the design.

SPEC 2006 [48] benchmark suite is selected for our evaluation. Since our linker cannot handle some of the benchmarks, these benchmarks are excluded. The workable benchmarks are listed below. We simulated the benchmarks with their "ref" input.

Integer: *perlbench*, *bzip2*, *gcc*, *mcf*, *gobmk*, *hmmer*, *sjeng*, *libquantum*, *h264ref*, *astar*.

Float: *bwaves*, *milc*, *gromacs*, *leslie3d*, *namd*, *GemsFDTD*, *tonto*, *lbm*, *wrf*, *sphinx3*.

This work also incorporated a power model from Wattch [49] to evaluate the energy consumption. The event counters were collected during the simulations and reported to Wattch which can generate a complete energy consumption file. The simulated results were validated against McPAT [50] with a very similar superscalar pipeline to ensure the correctness. Additionally, non-ideal clock gating option is enabled in Wattch (cause only 10% power usage when a particular memory port is not used) as well as the dynamic activity factors (precharging the bit line only if the content is zero in the last cycle). BDM and MDM were implemented with CAM in Wattch to simulate their power behavior.

We used GCC 4.3 to compile SPEC 2006 benchmarks with the highest optimization flag (-O3) turned on. Binutils 2.22 was used as the software environment and UClibC 0.9.33 was used to link the benchmarks. Operating system kernel was not simulated. For each benchmark, the first 500 million instructions were simulated as warm-up phase and their results were not collected. The next one billion instruction results were reported.

Baseline: An 8-issue superscalar processor with a Gshare branch predictor. The detailed configurations is listed in Table 2.1.

EMR: Eager Misprediction Recovery (EMR) [36] was proposed to take a checkpoint of the current F-RAT whenever a misprediction is detected. This checkpoint would compare with the correct F-RAT which was repaired later. If the register mapping is different in these two tables that means the register mapping is damaged. EMR would need the second checkpoint only if another misprediction is detected during

Table 2.1
The processor configurations

Processor Configuration	
Physical Register size	128
LSQ size	32
Fetch width	8
Decode width	8
Issue width	8
Int adder/subtractor	1 cycle
Int multiplier	3 cycles
Int divider	7 cycles
Float adder/subtractor	7 cycles
Float multiplier	7 cycles
Branch Predictor	4KB GShare 14 bits global branch history shift register

the first misprediction recovery procedure. In this evaluation, EMR is implemented with 4 checkpoints. The invalid instructions are immediately eliminated.

CPR: A checkpoint architecture [26] was used which has 8 checkpoints altogether. Whenever a misprediction is detected, the program is rolled back to the closest checkpoint prior to the misprediction. The checkpoint is not released until all the instructions between this checkpoint and the next checkpoint are completed without an exception.

Perfect: The invalid instructions are removed instantly and F-RAT is also restored immediately. This shows the performance upper limit of any misprediction recovery technique.

2.5.2 Branch Tag Size Effect

When a branch is renamed, it requires a flush tag. When there is no flush tag available, the processor has to stall. Therefore a large number of flush tags is preferred. On the other hand, the matrix (BDM, MDM) size is highly related to the number of flush tags. A smaller matrix is preferred from a hardware perspective.

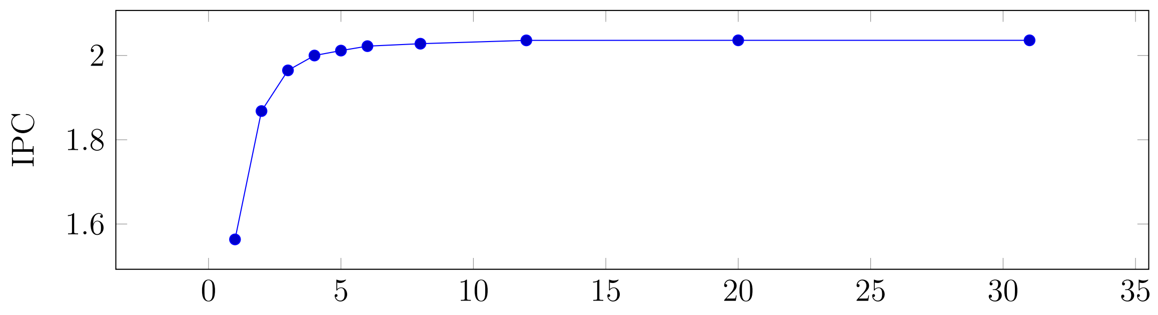


Figure 2.12: IPC vs number of unresolved branches

Figure 2.12 illustrates the performance of Mower with different numbers of flush tags. We use IPC (Instructions Per Cycle) to measure the performance which counts the number of executed instructions per clock cycle. Even when Mower is provided with 4 tags, the processor can still achieve 98.2% of the performance as a processor with infinite number of tags. In the rest of the evaluation, 12 flush tags are implemented in Mower as at this number the performance approaches to the upper limit of the performance (99.99%) and the size is reasonable.

2.5.3 Invalid Register Mappings

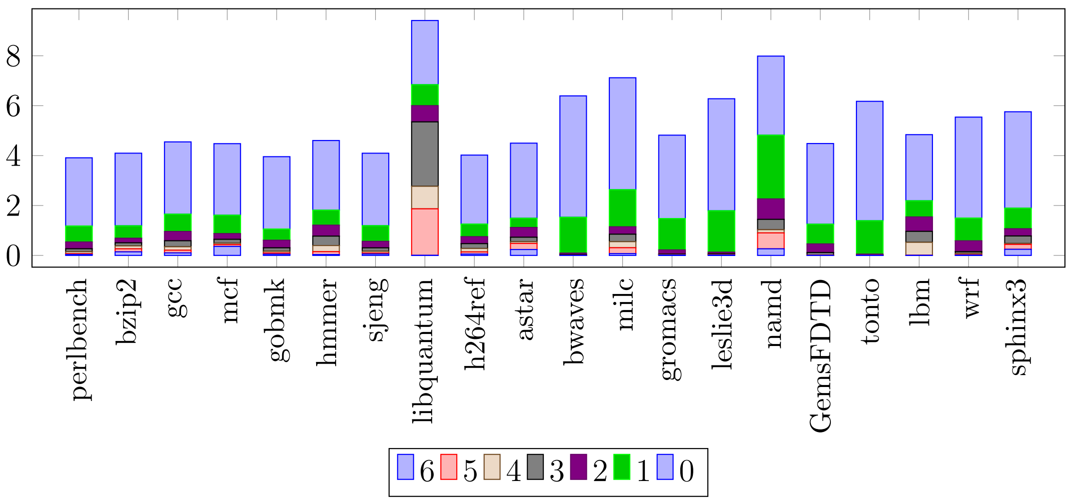


Figure 2.13: The number of invalid register mappings through the walking process

Mower is able to repair invalid register mappings through the walking process. Figure 2.13 shows the average number of invalid mappings through the walking process. The top bar which is marked as “0” is the number of invalid mappings when mispredictions are detected. The second from the top bar which is marked as “1” represents the number of invalid mappings left after one cycle of walking recovery. Accordingly, the bar of “6” represents the number of invalid mappings left after six cycles of walking recovery. Apparently, most of the benchmarks have fewer than 0.1 invalid mappings after six cycles of repairs. In other words, if the front-end of the pipeline has six or more stages, the correct F-RAT is mostly restored before the first new instruction arrives at the dispatch stage.

In Figure 2.13, most of the invalid mappings are corrected at the beginning of recovery. The first cycle of the walking recovers the most number of mappings in all benchmarks. This explains that most of the invalid mappings are defined very close to the misprediction point, thus walking from the misprediction instead of the ROB tail can significantly improve the recovery efficiency.

2.5.4 Eliminating Invalid Instructions

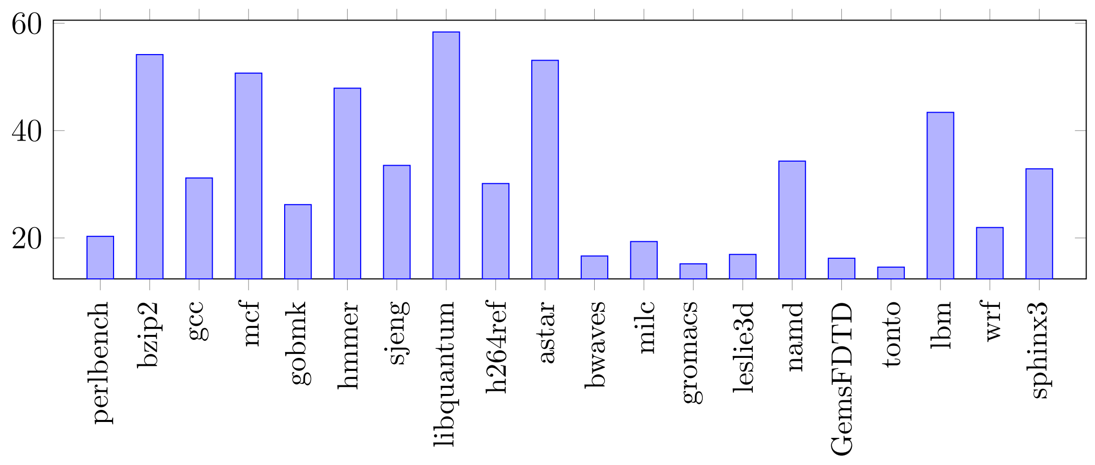


Figure 2.14: The average number of invalid instructions left in the pipeline when a misprediction is detected

Figure 2.14 shows the average number of invalid instructions left in the pipeline when mispredictions are detected. This is also the number of instructions walker has to eliminate in each recovery. Some benchmarks, such as *libquantum*, have up to 60 invalid instructions to be eliminated and other benchmarks, such as *bwaves* and *tonto*, have fewer than 20 invalid instructions.

Note that the number of invalid instructions are much greater than the invalid mappings (Figure 2.13 and 2.14). This is partially because some instructions do not require physical registers, like branches. Another reason is the same logical register is defined multiple times. A conventional walking process has to undo all the modifications but Mower only needs to undo at most once on any logical register.

2.5.5 The Front-end and Back-end depth

Figure 2.15 and 2.16 show the geometric mean (Gmean) of the performance normalized to the baseline. The x-axis is the pipeline depth which is composed of two parts, the front-end depth and the back-end depth. The first number is the front-end depth and the second number is the back-end depth. The back-end depth means the number of cycles for an instruction to commit after it is computed. For example, the configuration of "3-1" indicates the processor has a 3-stage front-end and takes one cycle to commit an executed instruction.

Mower works much better than the baseline when the processor has a short front-end and a long back-end. This is because a longer front-end can help to overlap more recovery delay in the baseline. In other words, the benefits provided by Mower are diminished. On the other hand, a higher back-end depth is detrimental to the performance of the baseline as baseline has to wait for each invalid instruction to

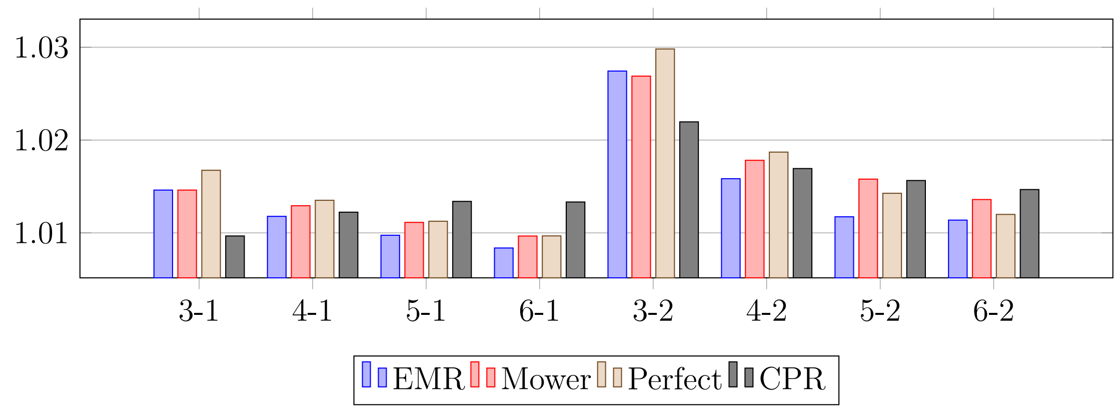


Figure 2.15: Spec2006 Integer Speedup (Normalized to Baseline)

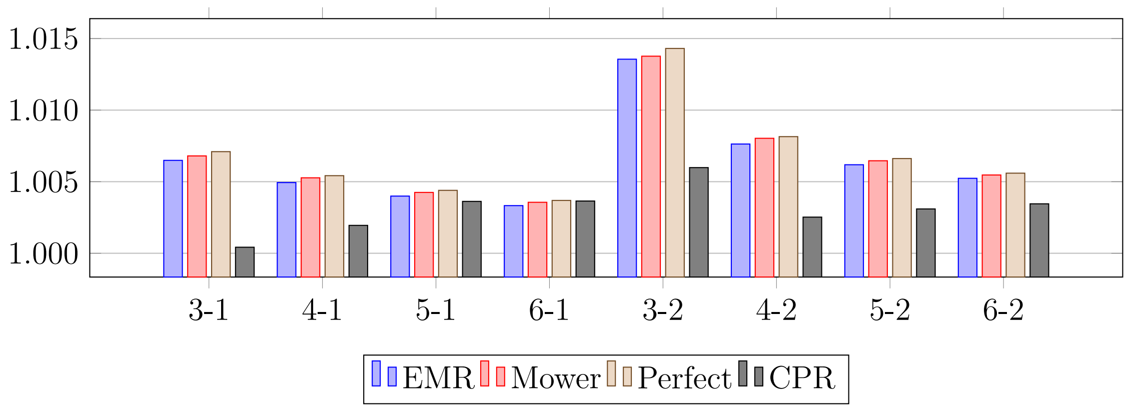


Figure 2.16: Spec2006 Float Speedup (Normalized to Baseline)

retire. Most notably, Mower performs very close to the perfect recovery in every scenarios.

2.5.6 Energy Efficiency

Mower consumes less energy as it reclaims invalid instructions much earlier than baseline. Figure 2.17 illustrates the energy consumption and the power dissipation of Mower normalized to baseline. Mower consumes less power in every benchmark except *libquantum* and *lbm*. In those two benchmarks, very few mispredictions are observed, therefore very few invalid instructions are eliminated. On the other hand, branches are still renamed and assigned flush tags, BDM and MDM are still updated which all consume extra energy.

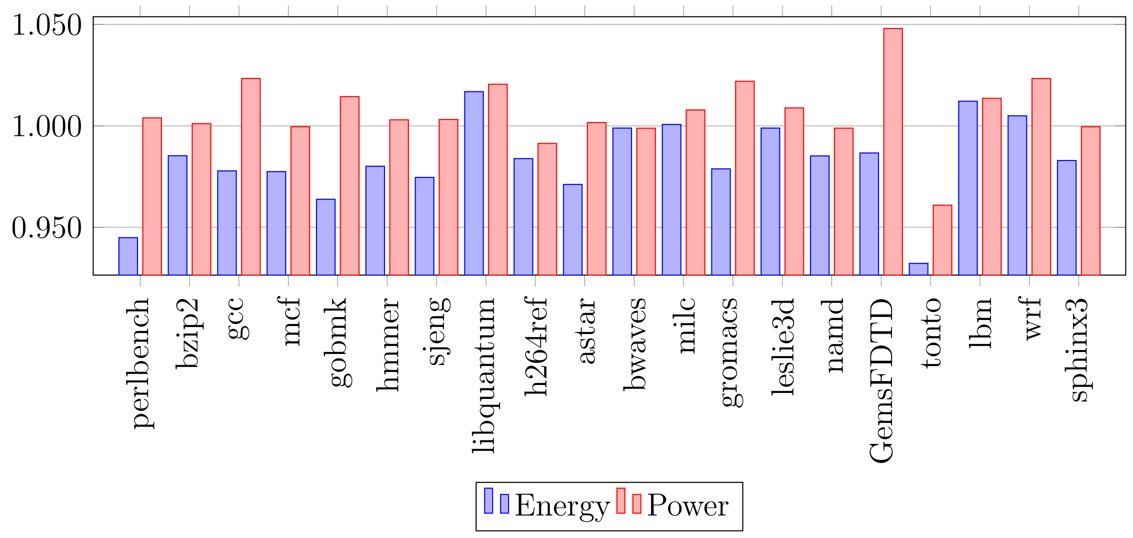


Figure 2.17: Power Evaluation (Normalized to Baseline)

Since Mower can recover mispredictions much quicker which leads to more instruction executions per cycle. In terms of the power dissipation, Mower is worse than the baseline in most benchmarks. On the other hand, the baseline would just stall the pipeline until the misprediction is recovered. Furthermore, Mower needs to access BDM and MDM every cycle which consumes extra energy.

The detailed energy and power results are listed in Table 2.2. MDM dominates the additional power dissipation due to its large size compared with BDM. Moreover,

Table 2.2
Energy consumption and Power dissipation in Mower and Baseline

	Mower				Baseline	
	energy	power	BDM power	MDM power	energy	power
perlbench	39414282.82	23.0847	0.0178	0.3418	41711982.94	22.9941
bzip2	19912545881	36.5107	0.0099	0.4674	20209901773	36.4707
gcc	20577678552	31.064	0.0177	0.4205	21044400878	30.3547
mcf	24399551282	23.9638	0.0125	0.2964	24962176205	23.9732
gobmk	21246144709	30.1403	0.0116	0.4082	22043497993	29.7116
hmmer	20169386181	52.1055	0.0108	0.7009	20578874932	51.9498
sjeng	20952348479	45.881	0.0243	0.5581	21498326343	45.7369
libquantum	18367126463	56.8546	0.014	0.7518	18062344589	55.7114
h264ref	19009792993	44.2755	0.015	0.5757	19321361564	44.6614
astar	25288238076	47.5951	0.0157	0.6439	26039881396	47.5177
bwaves	23545019645	49.1315	0.0012	0.5251	23570556909	49.1887
milc	22033327175	40.0966	0.004	0.4567	22017599284	39.7845
gromacs	20251109581	41.9003	0.0329	0.5177	20689221337	40.9992
leslie3d	22490620504	44.733	0.0043	0.5086	22515068357	44.3398
namd	19011620317	47.6229	0.0348	0.5776	19297024881	47.6767
GemsFDTD	18450836352	42.359	0.0428	0.5504	18700506380	40.4183
tonto	21579146650	45.8823	0.022	0.605	23147948645	47.7491
lbm	21052658006	34.911	0.003	0.3709	20799850538	34.4426
wrf	23438584264	56.3007	0.015	0.5877	23322660370	55.017
sphinx3	20007152673	45.5453	0.0387	0.5723	20354623969	45.564

MDM is updated more frequently than BDM as there are more ALU and Load instructions than branch instructions. An alternative solution would be to use banking to access a sub-matrix in MDM [51].

Figure 2.18 illustrates the Energy Delay Product (EDP) of Mower normalized to baseline. When Mower eliminates more invalid instructions, it reduces the energy consumption and improves the performance. As a result, it is more power efficient. The same result is also observed in Figure 2.17 that only *libquantum* and *lbm* are less power efficient (<1%).

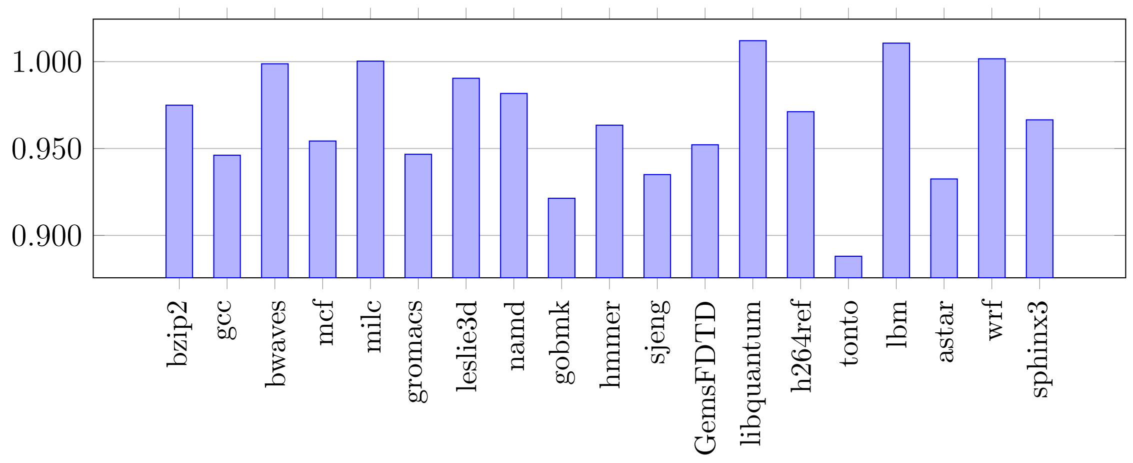


Figure 2.18: EDP normalized to the baseline configuration

2.6 Summary

Mower is an innovative mechanism to provide better branch misprediction recovery for superscalar processors. It has mainly three distinct properties to achieve this target: I) A reverse walker is used to repair F-RAT along with the front-end fill time; II) During the walking process, the resources occupied by the invalid instructions are reclaimed; III) A single access port is shared between the ROB tail and the walker in order to maximize the resource utilization.

Comparing with a checkpointing architecture, Mower does not require any checkpoints and still can provide a quick F-RAT recovery. The evaluation results showed that its recovery penalty is very close to a perfect recovery mechanism which can instantly recover F-RAT. Mower is also very energy efficient due to the fact that it can eliminate invalid instructions through the walking process. Mower provides a new method for efficiently recovering branch mispredictions.

Chapter 3

Two-Phase Misspeculation Recovery

3.1 Overview

In Chapter 2, we have developed a general mechanism which targets the **state restoration delay**. The mechanism affects the delay on two parts. First, it develops a microarchitecture mechanism where the **state restoration delay** can be partially overlapped with new instruction fetching. This is accomplished by developing an effective fine-grain state maintenance algorithm [36] which can maintain a

[0]The material contained in this chapter will be published in Proceedings of the 32nd ACM on International Conference on Supercomputing (ICS ”18).

fine-grain state at all times. Second, it develops an effective walking mechanism which reduces the **state restoration delay** itself.

In this chapter, we target the **stale instruction elimination delay**. Doing so, we first revisit the basic recovery mechanism and show that due to the deep front-end and back-end, this mechanism works well in most cases, except when the back-end is blocked by cache miss loads. We then develop a "two-phase recovery" mechanism which keeps the simplicity of the basic recovery technique while significantly shortening the **stale instruction elimination delay**. Furthermore, this mechanism can completely overlap misspeculation and cache miss penalties with each other.

3.1.1 A Basic Recovery Mechanism

A basic recovery mechanism is illustrated in Figure 3.1. The left hand side of the figure represents the instruction state in the pipeline. The front-end includes fetch, decode, rename and dispatch stages. The back-end includes the Reorder Buffer (ROB), Reservation Station (RS) and Load/Store Queue (LSQ). The right hand side of the figure depicts the control flow graph (CFG) of the executing program fragment, including all the instructions currently in the pipeline.

Figure 3.1(a) shows the initial state at which *I1* is the oldest instruction in the back-end. *I5* is a branch which was predicted to take the left path followed by

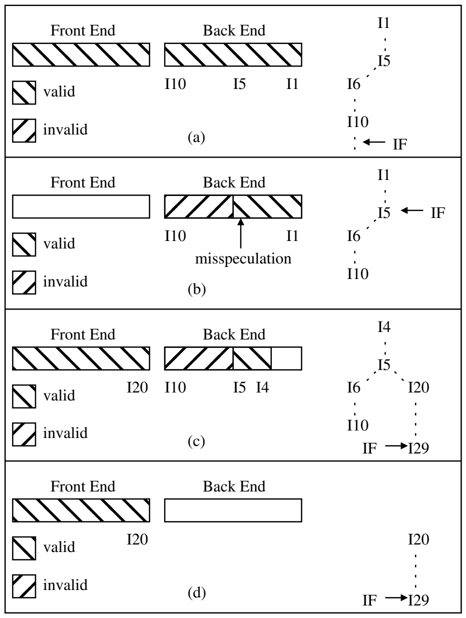


Figure 3.1: A basic recovery mechanism

I6. This branch is found to be a misprediction in Figure 3.1(b). Therefore, the instructions following *I5* all become invalid and the fetch engine is rolled back to the mispredicted branch *I5*. Since the instructions left in the front-end are invalid, they can be cleared immediately. However, the back-end contains a mixture of valid and invalid instructions. Therefore, it can not be flushed. In Figure 3.1(c), *I1, I2* and *I3* are retired and new instructions starting with *I20* fill the front-end. Since the back-end has not fully recovered, new instructions are not allowed to be dispatched to the back-end. Finally, in Figure 3.1(d), the last two valid instructions left in the back-end, *I4* and *I5*, are retired followed by a flush to clear the entire back-end. From this point on, the recovery procedure is completed and the new instructions will move forward to the back-end and start executing. This mechanism combines the **stale instruction elimination delay** with **state restoration delay** and is very simple to implement.

3.1.2 Analysis and Related Work

The **stale instruction elimination delay** is not necessarily longer than the time to fill the front-end with new instructions. However, if the back-end recovery is stalled by some long latency operation such as an LLC (last level cache) load miss, the penalty can be detrimental to the overall performance.

As a result, several techniques were designed to address this issue specifically. One approach is to selectively flush the invalid instructions once a misprediction is detected. For example, the mechanism described in Section 2.2.1 uses branch flush tags to rename every branch [25]. When a misprediction is detected, the flush tag of the mispredicted branch is broadcast through the entire pipeline to selectively eliminate the dependent instructions. Golander et al. uses checkpoint tags, instead of branch flush tags, to selectively flush misprediction dependent instructions [52]. Their mechanism has to broadcast all of the checkpoint tags following the misprediction sequentially as each tag can only control the instructions in its own section.

Another approach is to mark instructions differently such that valid and invalid instructions can be simultaneously executed. Kyker et al. proposed a mechanism to assign each instruction with a path color [53]. When a misprediction is detected, the new instruction stream is assigned a different path color. Hence, the new instruction stream can be dispatched to the back-end even if it has not been cleared. At the retire stage, the instructions which belong to the old path are retired until the mispredicted instruction is reached. Thereafter, the back-end contains stale instructions belonging to the old path and the valid instructions belonging to the new path. The stale instructions are retired without updating any architectural state. In order to accelerate the process of retiring stale instructions, several approaches are proposed. One of these mechanisms is to broadcast the old color to eliminate every instruction on the old path simultaneously. The second is to give the old path instructions priority to execute regardless of their operand availability.

One drawback of this mechanism is, it can delay the elimination process for ROB and RS, but not for LSQ since it is searched associatively. For example in Figure 3.2, a new path (blue) is assigned when the misprediction in the old path (red) is detected. The instructions between the head and the misprediction are valid and the rest of the instructions (red) are invalid. If the new load instructions (blue) are dispatched to the damaged LSQ, they may receive incorrect values. In this example, correct execution requires forwarding of the value from **SW1** to **LW**. However, **SW2** is closer and its value may be incorrectly forwarded to **LW**. Since **SW1** and **SW2** have the same path color, there is no easy way for the processor to distinguish them. In summary, the LSQ needs to be repaired or at least marked differently before any new memory instruction are dispatched.

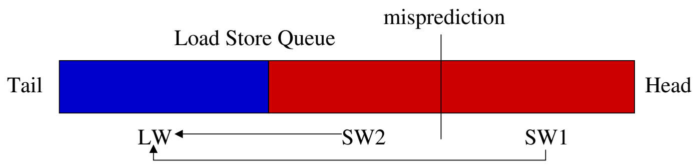


Figure 3.2: The issue to dispatch new instructions into the LSQ when it is not fixed

We assessed these techniques and found that all of them require special storage and/or a broadcasting network to eliminate stale instructions dynamically. For example, in [25], each unresolved branch has its own branch tag. The instructions have a set of branch tags indicating which branch they are dependent on. However, unlike Mower, when a branch is mispredicted, the corresponding branch tag is broadcast to the entire pipeline which either selectively eliminates or marks its dependent instructions. All of these mechanisms also have to allocate renaming tags up-front for each instruction which may lead to a misspeculation, such as branches. On the other hand, if the instruction is not assigned a special tag, such as a load instruction, the corresponding misspeculation has to be recovered using a different mechanism. Furthermore, the hardware cost of selective flushing is non-negligible. For every entry within the components in the back-end, a tag storage and a comparator needs to be added. A typical superscalar processor might have hundreds of these entries (Intel Skylake, 224 entry ROB, 97 entry RS, 72 entry Load Buffer, 56 entry Store Buffer). Moreover, these comparator matrices consume additional energy during the recovery.

3.1.3 Key Observation

A neglected fact is that selective flushing speeds-up recovery only when the back-end recovery is blocked by some long latency operation and this is not the common case in real execution. Back in Figure 3.1, if the mispredicted instruction *I5* can be retired fast enough, the back-end can be reset without blocking the front-end. The reset process efficiently eliminates every instruction left in the back-end. In summary, when the back-end recovery time is short, the basic recovery mechanism works well. When the back-end recovery is long, a better recovery mechanism which is also cost-effective is desirable. We therefore develop a two-phase recovery mechanism which optimizes the recovery time for these two scenarios.

In our two-phase mechanism, when a misprediction is detected, the first phase of the recovery is initiated. This phase is nothing but the basic recovery mechanism shown in Figure 3.1. The second phase will not be triggered until the back-end recovery procedure is stalled by a long latency operation. Among all the long latency operations, loads which miss in the LLC are the most detrimental to the back-end recovery. Therefore, when the ROB head reaches an LLC miss load, the second phase of recovery is triggered. **The second phase simply treats the missing load as if it is a load misspeculation and restarts fetching the stream from the load**. When the newly fetched stream arrives at the end of the front-end, the back-end is reset and the recovery procedure terminates. During the second phase of recovery, the load and the following instructions left in the back-end can neither be retired nor update any architectural state. When the mispredicted instruction which triggered the recovery procedure is fetched again during the second phase of recovery, instead of the predicted values, the correctly computed values are substituted. For example, a mispredicted branch will use its computed result instead of the prediction provided by the branch predictor.

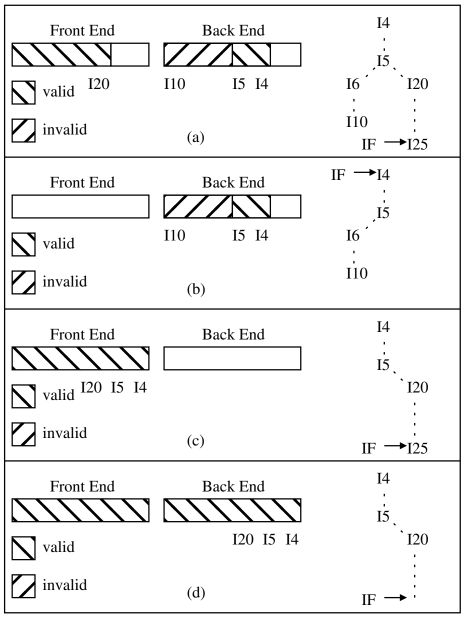


Figure 3.3: The second phase of the recovery

Figure 3.3 illustrates the second recovery phase. The first phase is shown in Figure 3.3(a) in which *I4* is an LLC miss load which stalls the retire stage. This load triggers the second phase which is shown in Figure 3.3(b). The front-end is flushed again and this time the fetch engine is redirected to the instruction which stalls the back-end, *I4*. When the mispredicted instruction *I5* is fetched, it uses its computed result and takes the right path. Figure 3.3(c) shows that when the front-end is filled with the correct instructions, the back-end is reset and the recovery procedure terminates. If the memory response time of *I4* is long enough, the whole pipeline is stalled in Figure 3.3(d). When that happens, the branch misprediction recovery penalty completely overlaps with the cache miss delay.

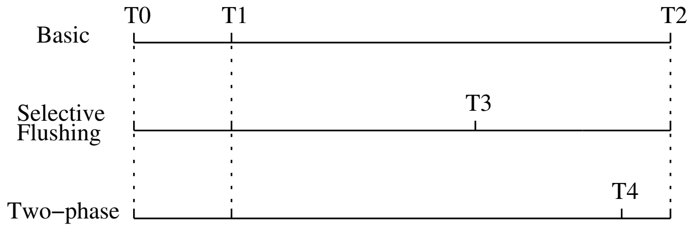


Figure 3.4: Timeline of Recovery

For the three recovery techniques, basic, selective flushing and two-phase, the recovery timeline is shown in Figure 3.4. **T0** is the point at which the misspeculation is detected and the recovery procedure commences. **T1** is the time when ROB head reaches the load that missed in the cache and **T2** is the time the load obtains its data from the memory. For the basic mechanism, from **T0** to **T2**, the processor cannot dispatch new instructions to the back-end as the back-end has not recovered. In case of selective flushing mechanism, the assumption is the flushing operation takes place instantly. Therefore, it takes **T0**$\sim$**T3** to fetch and dispatch the new stream starting from the misprediction point and fill the back-end. For the two-phase mechanism, the second phase does not launch until **T1** is reached. Thereafter the processor takes **T1**$\sim$**T4** to fetch and dispatch instructions starting with the missed load and fill the pipeline. Since the two-phase mechanism dispatches more instructions, (**T4-T1**) is greater than (**T3-T0**). As long as the cache miss latency is longer than this time, the performance of selective flushing and the two-phase recovery would be identical.

3.2 Preliminary Analysis

In this section, a preliminary analysis is provided to quantitatively assess the differences between the proposed mechanism and other techniques.

Table 3.1 lists the recovery related statistics. The evaluation is accomplished by simulating a basic recovery mechanism using the configuration specified in Section 3.4.1. The first part of the data shows the number of mispredictions per 1k retired instructions (MPKI). The data is divided into two subcategories, the back-end recoveries stalled by missing loads and the recoveries which are not stalled. Clearly, those cases which are not stalled represent the common case and the conclusion is that the basic recovery mechanism should work well in most cases. The second part of the data shows the number of cycles spent between the misprediction detection and retirement of the mispredicted instruction, namely, the back-end recovery time. This result is also divided into two subcategories. Note that the back-end recovery has a very large delay when it is stalled by cache miss loads except in *libquantum* and *h264ref*. The average delay is 105.3 cycles through all simulated benchmarks. On the other hand, the average recovery delay is only 4 cycles for those which are not blocked by a cache miss. Given that modern superscalar processors have 10 or more pipeline stages to fetch, decode, rename and dispatch instructions, the back-end recovery delay will be mostly overlapped by the front-end fill delay when the back-end is not stalled by a cache miss.

Table 3.1
The average misspeculation recovery ratio and the corresponding back-end recovery time

	misprediction ratio		average cycles	
	w/ miss (MPKI)	w/o miss (MPKI)	w/ miss (Cycles)	w/o miss (Cycles)
perlbench	0.1972	7.6372	140.7410	4.1971
bzip2	0.2008	10.7664	87.5747	3.3479
gcc	0.1965	2.0736	121.0998	4.0351
mcf	2.2920	7.0885	117.3951	2.9718
gobmk	0.0900	11.4311	90.9878	4.0375
hmmer	1.1071	10.0676	101.7260	2.8986
sjeng	0.0668	8.9227	121.4520	4.2428
libquantum	0.0445	0.0234	7.9040	2.0131
h264ref	0.1252	1.6037	27.2903	4.9313
astar	0.5413	10.7609	84.3921	2.5816
bwaves	0.0254	0.4362	113.1180	2.2977
milc	0.0002	0.3065	145.5395	2.3869
zeusmp	0.0329	2.4058	112.1057	2.8289
gromacs	0.0011	28.6441	104.3921	2.2456
leslie3d	0.0164	12.0679	125.6261	2.1740
namd	0.0066	1.2174	93.2071	4.9071
GemsFDTD	0.0024	0.0064	124.3514	20.4624
tonto	0.0099	4.0393	136.5524	2.9060
lbm	0.0061	0.0103	141.6629	2.0402
wrf	0.0116	1.0228	125.1721	3.1938
sphinx3	0.3948	1.3044	89.9858	3.3013
Average	0.2557	5.8017	105.3465	4.0000

Figure 3.5 shows the average number of cycles spent between misspeculation detection and the point when the back-end is blocked by a long latency instruction. In our

simulation, the long latency instruction is an LLC miss load. The corresponding delay is equivalent to the data shown in Figure 3.4 as **T0∼T1**. It is clear that if the recovery procedure is blocked by a cache miss, the time to arrive at that point is very short. The average delay through all benchmarks is only 2.57 cycles. Moreover, *hmmer* and *gromacs* have smaller than one cycle delay, which means in most cases the back-end has already stalled when the misprediction is detected.

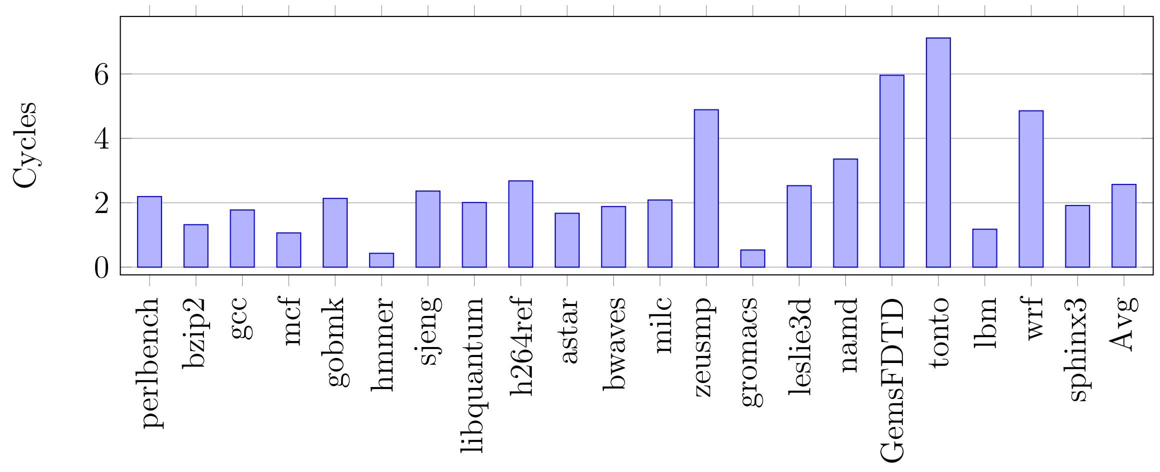


Figure 3.5: Phase one to Phase two delay

Figure 3.6 illustrates the number of instructions left in the ROB between the misprediction and the missing load. This number varies from 8.43 to 201.35 and the average is 86.92. The basic recovery mechanism simulated in this work is an 8-issue superscalar processor. Approximately, 10.86 cycles are used to fetch the extra instructions compared to a selective flushing mechanism. When we combine this delay with the time used to enter phase two recovery in Figure 3.5, the average number of additional cycles is 13.4. This is the extra delay for the two-phase recovery mechanism, compared against an architecture which can instantly eliminate invalid instructions. This delay can also be found in Figure 3.4 as **T4 - T3**.

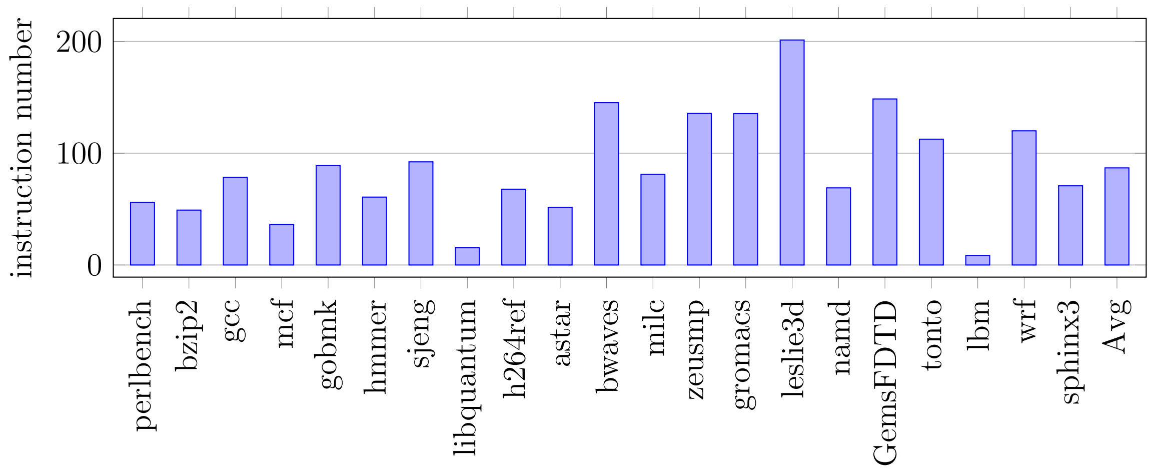


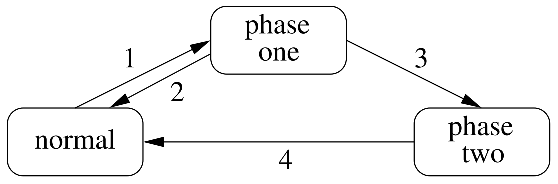
Figure 3.6: The number of instructions between the cache miss load and the misprediction

3.3 Microarchitecture

In this section, the microarchitecture details of the two-phase recovery mechanism is elaborated. Figure 3.7 demonstrates the state machine diagram of the mechanism. The phase one of the recovery is so pervasive that we will not describe it. We are going to focus on the explanation of the phase two recovery.

3.3.1 Initialization

The fetch engine is redirected to the PC of the load which blocks the ROB head. The instructions left in the front-end and the back-end are all treated as invalid



1. A misprediction is detected.
2. The misprediction is retired, the back–end is flushed.
3. The ROB head is stalled by a cache miss load.
4. The mispredicted instruction is re–fetched and the invalid instructions are flushed.

Figure 3.7: Two phase recovery state machine

instructions. Hence, they are not allowed to retire nor update any architectural state. The state of the branch predictor also needs to be recovered. The most important component is the branch history register (BHR) which contains the previous branch predictions. We have explained the recovery process for BHR in Section 1.3.1. Each branch instruction has a pointer pointing at the BHR position when the branch is predicted. So if the branch is mispredicted, its pointer is used to recover the BHR. In order to recover the BHR during phase two recovery, we need a new pointer to keep the pointer of the most recent retired branch. It is the retirement BHR pointer (R-BP) which works similar to R-RAT. Whenever a branch is retired, its BHR pointer is copied to the R-BP. At the beginning of the phase two recovery, the R-BP is used to recover the BHR. Another very important component is the Return Address Stack (RAS) which is used to provide the return address after a function is called. It is possible that RAS is corrupted if the branch inside the function is mispredicted and the return address in RAS has been overwritten. A renamed version of RAS [54] is implemented to overcome this problem, hence the processor can restore RAS after any misprediction. We implement a retirement mapping table for the RAS similar to R-RAT. Therefore, RAS can be repaired in phase two of the recovery.

3.3.2 Three Different Flushing Policies

Since instructions left in the pipeline are invalid, they can be immediately evicted during phase two recovery. However, in many cases the two paths diverging from a mispredicted branch instruction converge. As a result, even after a misspeculation was detected, the execution of invalid instructions might still warm up the data cache correctly. Consequently, we investigate three different flushing policies. The *conservative policy* is to flush both the front-end and the back-end, which wastes the least energy to execute stale instructions. The *moderate policy* only flushes the front-end so that stale instructions left in the back-end are allowed to execute. The *aggressive policy* does not remove any instructions and keeps dispatching stale instructions left in the front-end until the back-end fills up. If the dispatch stage is stalled by stale instructions, these instructions left in the front-end are removed. In this policy, the front-end is mixed with newly fetched valid instructions and invalid ones. A one bit coloring mechanism is thus added. Once valid instructions reach the back-end, all stale instructions are eliminated irrespective of the policy that is used.

3.3.3 Fetch Policy

In phase two recovery, the misspeculated instruction will be re-fetched and re-executed. If the default prediction is used, this instruction will be mispredicted again and cause a deadlock. Therefore, the execution result should be used to overwrite the prediction for mispredicted instructions.

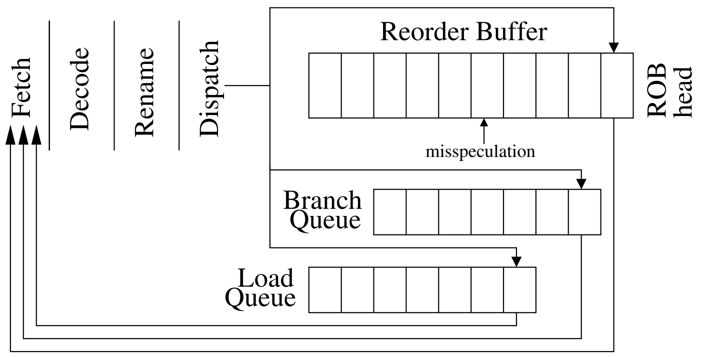


Figure 3.8: Overwriting the prediction

Figure 3.8 illustrates the overwriting mechanism. The task of finding the first misspeculation is done by scanning the ROB entries. Each ROB entry has one bit indicating the misspeculation status. Every fetched instruction has to read its corresponding entry in the ROB and if the entry has the misspeculation bit set, the executed result is reused instead of the prediction. From this point on, no instructions are reused. The scan operations are already implemented in the basic recovery mechanism, thus, there is no extra overhead. The misspeculation bit is set when the instruction is executed and reset when the ROB entry is assigned to a newly dispatched instruction.

3.3.4 F-RAT and Free Register Pool

While the new instruction stream reaches the rename stage, F-RAT is repaired by copying from R-RAT. This is exactly the same procedure which is used in the basic recovery mechanism. There is no need to take checkpoints or implement a walking process. The free register pool is repaired in the same way as the basic recovery mechanism.

3.3.5 Speculative Recovery

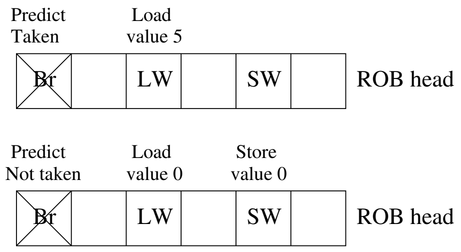


Figure 3.9: Misprediction due to speculative recoveries

Most recovery mechanisms recover in program order such that the oldest misspeculation is repaired. In our mechanism, we assume the instructions between the misspeculation and the ROB head do not cause any more misspeculations during a phase two recovery. Therefore, it is speculative and has its own side effects. For example, the top half in Figure 3.9 shows a branch which was predicted taken. This branch is mispredicted as one of its operands comes from a load and the load reads a value of five. Thus a recovery procedure is initiated and finally a phase two recovery is issued. The bottom half shows the instruction execution after recovery. The load has the same address as the store which was not executed before the recovery. This time the load is forwarded from the store through the store queue and gets a value of zero. As a result, the branch is mispredicted again. The fact is, although the original prediction was correct, due to the load misspeculation, this is not correctly detected. Therefore, the branch instruction ends up triggering the recovery procedure twice. Our evaluation shows that such cases are rare (fewer than 0.01% of the total misspeculations), so their impact is negligible.

3.3.6 Complexity Comparison

The hardware complexity should be taken into account when we are designing a recovery mechanism. Therefore, the overhead among different techniques are compared in this section.

The first part is the overhead to eliminate stale instructions. The selective flushing mechanism which uses the branch path tag [25] has to keep a tag for every entry in the back-end, such as ROB, RS, LSQ, etc. Whenever a branch is resolved, a single bit is broadcast to reclaim the bit in the branch path tag. If the branch is mispredicted, another bit is broadcast to selectively flush the entries which have the corresponding tag bit set. Figure 3.10 shows a branch path tag with three bits which are implemented by D-type registers. Initially, the tag was 111 which means the corresponding instruction was dependent on branch 001, 010 and 100. The right side of the figure shows the waveform of the simulation. In the first cycle, branch 001 is resolved and it broadcasts X[0] to reclaim the bit. In the second cycle, branch 010 resolves and it is mispredicted. Therefore, the misprediction bit is set. Since the tag indicates this instruction is dependent on the branch 010, then the "reset entry" signal is triggered to flush the corresponding entry. The number of the D-type registers is determined by the maximum number of unresolved branches and the number of the entries in the back-end.

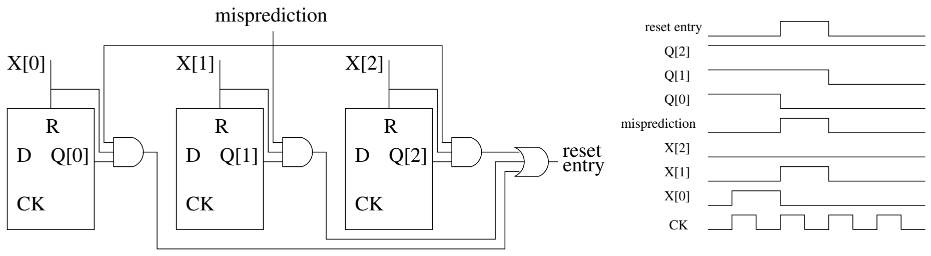


Figure 3.10: The branch path tag for selective flushing

The second part is the overhead to recover F-RAT. We assessed two commonly used mechanisms, checkpointing and walking. Each checkpoint has to have the same amount of storage as F-RAT. When a checkpoint is allocated, the copy process from F-RAT to the checkpoint has to be done in one cycle or an extra delay will be incurred at the rename stage. Since keeping a checkpoint for every branch is costly, a selective checkpoint allocation algorithm is widely used [26]. Therefore, a confidence predictor is required and its accuracy will have an immense impact on the utilization of checkpoints. On the other hand, the walking process needs at least an additional RAT to walk with. At the beginning of the recovery, this RAT is copied from R-RAT or F-RAT depending on the walking direction. The walker then walks through the ROB entries to collect the updates of register mappings. If the ROB head is retiring instructions and updating R-RAT at the same time, a second read port to the ROB is necessary for the walker. **Mower [29] solves this problem by sharing the port between the ROB tail and the walker**.

The two-phase recovery mechanism reuses most of the resources needed for a basic recovery technique. The only additions are a two-phase state machine and a detection structure to overwrite the prediction of mispredicted instruction. Table 3.1 shows that less than 5% of the recoveries were obstructed by LLC miss loads. The total overhead to optimize these 5% cases should be minimized.

3.4 Evaluation

3.4.1 Simulation Methodology

The two-phase recovery mechanism is simulated by using the MIPS-I ISA without delayed branching. This ISA is very similar to PISA ISA, used by SimpleScalar [55].

GCC 4.9.2 tailored to this ISA is used to compile the benchmarks and generate binary code with the highest optimization ("-O3") set. We choose Spec 2006 as our benchmark suite. All simulation models were designed with Architecture Description Language (ADL) [46]. The ADL compiler can automatically generate the assembler, the disassembler and a cycle-accurate simulator which respects timing at the register transfer level from the description of the microarchitecture and its ISA specified in ADL.

In order to efficiently simulate our mechanisms, we incorporated Simpoint 3.2 [56, 57] to minimize the simulation time. For each benchmark, a set of checkpoint images were generated where each checkpoint image contains the complete memory data segments, the register file and the program counter (PC). Other architecture related structures were not included, such as cache, branch predictor, memory dependence predictor, etc. Hence, the simulation of each interval has a cold start. In order to compensate for this effect, a large section of 100 million retired instructions was selected to simulate each interval. Since each interval simulation was independent of others, all of the intervals can be simultaneously simulated to further shorten the simulation time. Currently, the file descriptors are not kept in the checkpoint. Therefore, if the interval has file operations and the file descriptor was created before the checkpoint, the simulation of that interval will be wrong. When this happened, that checkpoint interval is replaced with the dominant checkpoint in that benchmark. In *h264ref*, one checkpoint (0.92% weight) was substituted with the dominant checkpoint (18.14% weight) and in *hmmer*, two checkpoints (0.22%, 0.43% weight) were substituted with the dominant checkpoint (98.9% weight). As the replaced intervals have very limited weights (<1.0 %), the impact of this substitution is expected to be negligible. The power evaluation was finished by a modified version of McPAT 1.4 [50] to evaluate the dynamic energy consumption. DRAMSim2 [58] was also embedded to evaluate the memory subsystem behavior.

The benchmarks we simulated in this work are:

Integer: *perlbench*, *bzip*, *gcc*, *mcf*, *gobmk*, *hmmer*, *sjeng*, *libquantum*, *h264ref*, *astar*.

Float: *bwaves*, *milc*, *zeusmp*, *gromacs*, *leslie3d*, *namd*, *GemsFDTD*, *tonto*, *lbm*, *wrf*, *sphinx3*.

Table 3.2
Processor Configuration

ROB / RS / PRF	256 / 64 / 320
Fetch / Decode / Issue	8 / 8 / 8
Cache	32KB 8-way set associative iL1; 32KB 8-way set associative dL1; 512KB 8-way set associative L2, 10 cycles hit latency;
Memory	16GB DDR3L-1600, 2 channels, 2 ranks, 8 banks, open page, up to 64 pending requests [59]
Recovery Penalty	minimum 15 cycles
Int ALU / Int Mul	1 cycle / 3 cycles
Int Div, FP ALU	7 cycles
Branch Predictor	8 kB TAGE [60]
Memory Ordering	Store Sets [16]
Tech node	22nm
Clock frequency	3.2GHz

All of the benchmarks were simulated with the "ref" input. The remaining missing benchmarks were not included due to the linker's inability to link them. The configuration shown in Table 3.2 was shared by all the simulation models. The implementation details of different recovery mechanisms are as follows:

1. Recover Afterwards: In this model, the stale instructions are handled after the misprediction is retired. Path information [53] is carried by every instruction. The new instruction stream fetched after a misprediction was detected is assigned a new path color. So after the mispredicted instruction retires, the remaining instructions belonging to the wrong path are invalid. A walker is implemented to recover F-RAT walking from the ROB head towards the misprediction. The stale instructions are retired without updating any architectural state. There are two methods implemented to retire stale instructions sooner. One is to eliminate them when the mispredicted instruction is retired. This policy is named as *Recover Afterwards without executing Bogus instructions* (RA-WOB). The other is to execute a stale instruction unless it is an LLC miss load where the load is forwarded with a bogus value. This policy is called *Recover Afterwards with executing Bogus instructions* (RA-WB).

2. Recover Beforehand: This architecture selectively flushes the stale instructions before the misprediction is retired. A branch flush tag is assigned to the branch when it is renamed and a checkpoint of F-RAT is taken with this branch. If this branch is mispredicted, its tag is used to evict the stale instructions and its checkpoint is used to recover F-RAT. There are two different configurations to assign flush tags. One is to greedily assign whenever the flush tags are available. The other is to assign tags only for the low confidence branches. The confidence predictor implemented in this work is derived from a TAGE branch predictor with its internal storage [61]. These two policies are *Recover Beforehand with Greedy Allocation* (RB-X-GA) and

Recover Beforehand with Low confidence Allocation (RB-X-LA). The 'X' represents the number of flush tags which is also equivalent to the number of checkpoints. Both configurations keep dispatching branches with no checkpoints when there is no free checkpoint. Note, these approaches cannot recover other types of misspeculations and are limited to the branch mispredictions which have been assigned flush tags. A basic recovery mechanism is used for the misspeculations which are not covered by checkpoints.

3. Two-Phase: This is the proposed mechanism which can recover from any type of misspeculation and does not need special tags or checkpoints.

4. Inf: This simulation model has an infinite number of checkpoints. Therefore, it can recover from any type of misspeculation.

3.4.2 Recover Afterwards and Two-Phase

Figure 3.11 shows the IPC results normalized to the Inf configuration for RA-WOB, RA-WB and Two-Phase. Using geometric means, the results in RA-WOB, RA-WB and Two-Phase are 98.84%, 99.75% and 99.42% respectively. In the figure, RA-WB outperforms the architecture with an infinite number of checkpoints in *gcc* and *mcf*. The reason is RA-WB can execute stale instructions and is able to warm up the cache. However, Inf has to flush stale instructions and the future loads experience a longer latency. From our experiments, the average load latency in RA-WB is 36.6 cycles for *gcc* and 104.7 cycles for *mcf*. The latency in Inf is 37.7 cycles for *gcc* and 108.1 cycles for *mcf*.

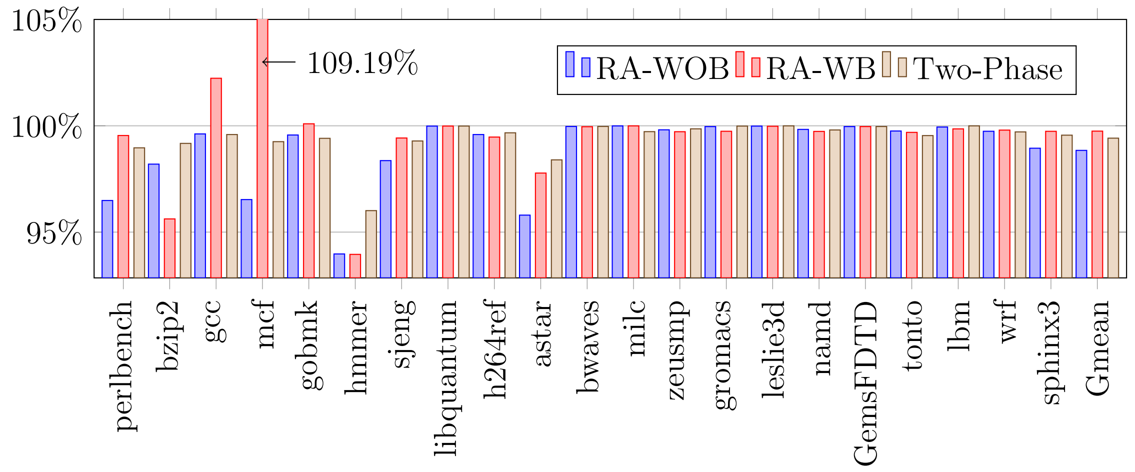


Figure 3.11: Recover Afterwards with/without executing bogus instructions vs. Two-Phase, normalized to Inf

RA-WB surpasses RA-WOB in most benchmarks except in *bzip2* where executing stale instructions negatively affects the performance. Stale instruction execution may warm up the cache, but it can also evict some useful cache lines. From the experiments, the average load latency for *bzip2* is 26.9 cycles in RA-WOB and 28.7 cycles in RA-WB. It is clear that more useful cache lines were evicted by stale instruction execution in *bzip2*.

Using geometric means, RA-WB works better than Two-Phase, but only marginally. Note that this advantage is mostly contributed by a single benchmark, *mcf*. Therefore, we calculated the Gmean again excluding *mcf* and Two-Phase was about 0.1% better than RA-WB. The performance of RA-WB highly relies on the behavior of stale instruction execution. The additional simulation showed that an architecture with the basic recovery mechanism is able to outperform RA-WB in *bzip2*. On the other hand, Two-Phase consistently worked better than this basic recovery mechanism in all benchmarks.

3.4.3 Recover Beforehand and Two-Phase

Figure 3.12 shows the IPC for RB-8-GA, RB-8-LA and Two-Phase normalized to the Inf models. The Gmean is 99.53%, 99.23% and 99.42% respectively. The performance gap between a checkpointing architecture and Inf is caused by the misspeculations which are not covered by checkpoints. The number of the misspeculations which have checkpoints in RB-8-GA is shown in Table 3.3. The column of "total" means the total number of Misspeculations Per 1k retired Instructions (MPKI), including branch misspeculations and memory ordering misspeculations in this work. The column of "ratio" represents the percentage of misspeculations which have checkpoints. As shown, a very large number of misspeculations are assigned checkpoints in *hmmer*, thus the performance is very close to Inf. On the other hand, *perlbench*, *mcf* and *astar* have fewer misspeculations covered by the checkpoints so the performance gap enlarges.

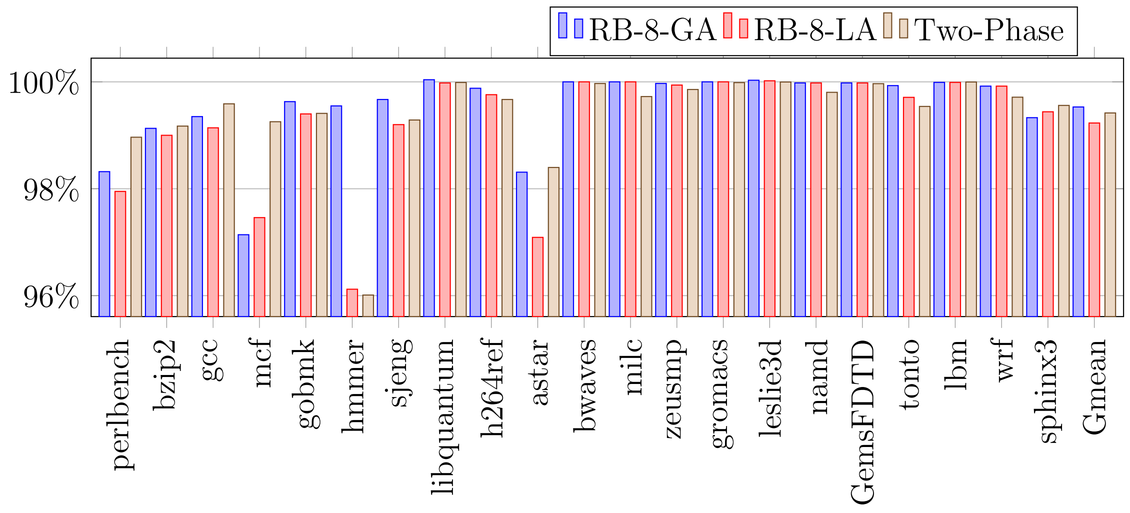


Figure 3.12: RB-8-GA vs. RB-8-LA vs. Two-Phase, normalized to Inf

Table 3.3
The misspeculations which have checkpoints

	total (MPKI)	ratio		total (MPKI)	ratio
perlbench	7.9478	88.48%	bzip2	10.9672	89.85%
gcc	2.2935	85.64%	mcf	9.3805	77.87%
gobmk	11.5669	94.08%	hmmer	11.1747	96.86%
sjeng	9.0844	87.42%	libquantum	0.0679	42.00%
h264ref	1.7351	90.45%	astar	11.3023	93.09%
bwaves	0.4615	97.23%	milc	0.3066	99.07%
zeusmp	2.4386	97.35%	gromacs	28.6453	99.78%
leslie3d	12.0843	98.04%	namd	1.224	98.47%
GemsFDTD	0.0088	99.93%	tonto	4.0508	98.20%
lbm	0.0163	99.90%	wrf	1.0344	97.40%
sphinx3	1.6993	89.92%	Average	6.0710	91.48%

The performance gap between Two-Phase and Inf in Figure 3.12 is due to the extra delay in phase one and phase two of the recovery. In phase one, even if the ROB head is not blocked by any LLC cache miss load, retiring instructions before the misprediction may still take a long time. If the time to retire these instructions is longer than the time it takes for the valid instructions to go through the front-end, the front-end has to stall. In phase two, it takes extra cycles to enter phase two (Figure 3.5) and additional cycles to re-fetch the valid instructions before the misprediction (Figure 3.6).

Figure 3.13 depicts the extra cycles incurred by Two-Phase for every 1k retired instructions. This delay is divided into two parts, phase one and phase two. Apparently, the delay in phase two dominates the total delay. In the figure, *mcf*, *hmmer* and *astar* have the most extra delays, thus Two-Phase has the largest performance gap in these benchmarks compared with Inf.

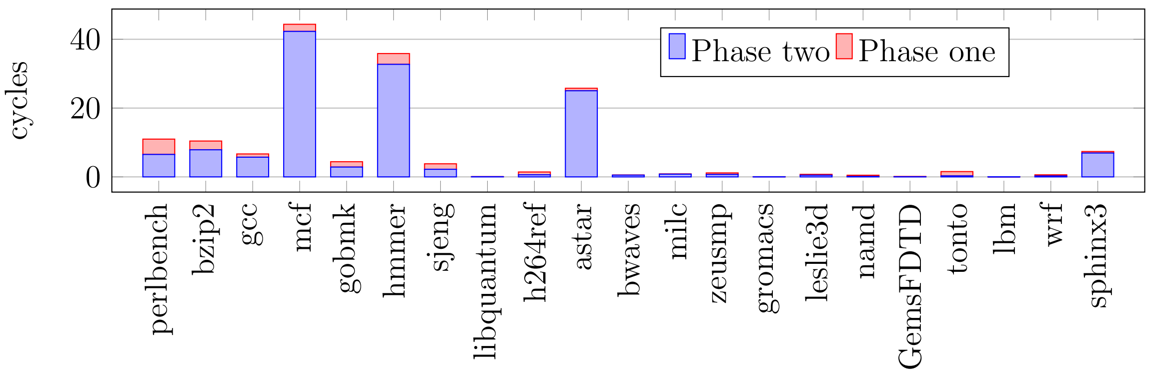


Figure 3.13: The extra cycles caused by Two-Phase compared with Inf

3.4.4 Allocation Algorithms in Checkpointing Architectures

Figure 3.14 illustrates the Gmean of architectures with different checkpoint sizes and different allocation algorithms, normalized to Inf. Note that with 4 checkpoints, the low confidence allocation algorithm works slightly better than the greedy algorithm. But in case of 8 checkpoints, the greedy algorithm works better. This is because when the resources are constrained, a more efficient allocation algorithm is better. Otherwise, allocating resources greedily is better as the low confidence allocation may not fully utilize the available resources.

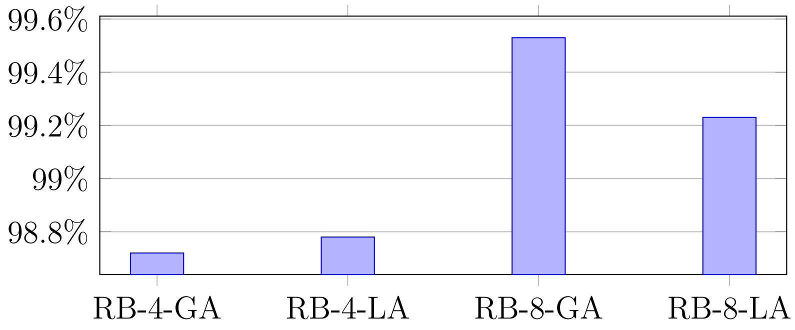


Figure 3.14: The Geometric mean of different checkpoint allocation algorithms, normalized to Inf.

3.4.5 Memory Latency Effect

The performance of Two-Phase is very sensitive to the memory response time, thus we simulated Two-Phase with different memory frequencies and the results are shown in Figure 3.15. Note that Two-Phase gets closer to Inf as the memory latency increases. From the timeline in Figure 3.4, it is clear that Two-Phase is more likely to enter the same state as Inf if the memory latency is longer. As a result, more recovery penalties overlap with cache miss penalties.

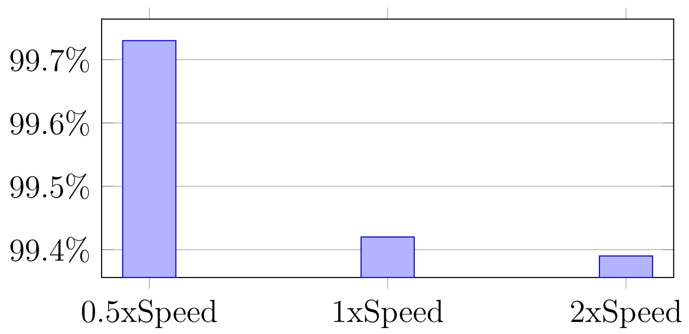


Figure 3.15: The Geometric mean on different memory speed, normalized to Inf.

3.4.6 ROB Size Effect

Figure 3.16 shows the performance of Two-Phase with different ROB sizes, normalized to Inf. The performance gets closer to Inf when the ROB size is reduced. The reason is similar to the case of memory frequency. A smaller ROB is much easier to fill when a cache miss load blocks the commit stage. Therefore, more branch mispredictions overlap with the cache miss latency. On the other hand, Inf can fetch and execute instructions earlier than Two-Phase when the pipeline is not stalled with a larger ROB.

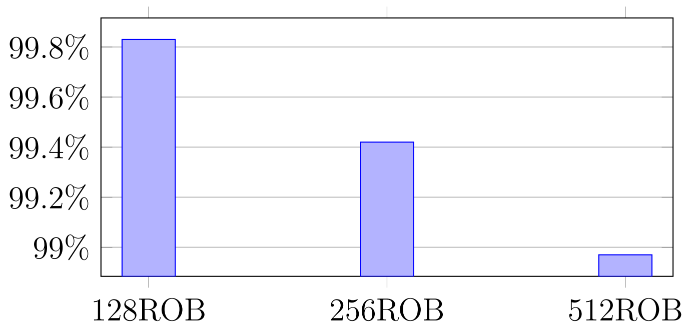


Figure 3.16: The Geometric mean on different rob size, normalized to Inf.

3.4.7 Issue Width Effect

We simulated the effect of issue width as well and the results are illustrated in Figure 3.17, normalized to Inf. A more aggressive processor works better as the time to re-fetch and re-execute valid instructions decreases. Moreover, the retire width is set to be the same as the issue width. Therefore, the time to enter the phase two mode is also reduced in a more aggressive processor.

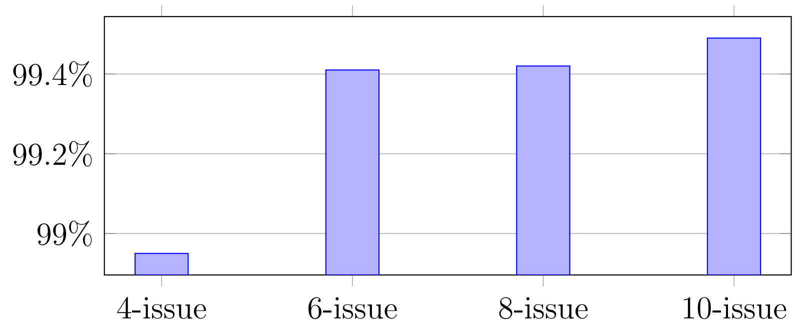


Figure 3.17: The Geometric mean on different issue width, normalized to Inf.

3.4.8 Power Efficiency

The EDP of Two-Phase was simulated and compared with an 8-checkpoint, greedy allocation architecture. The column of "2-phase" in Table 3.4 shows the relative results. Two-Phase algorithm saves about 1% of the EDP using geometric means. The Two-Phase algorithm has to re-fetch and re-execute valid instructions during phase two recovery, thus the wasted energy increases when more instructions are re-executed, such is the case with *hmmer*. An alternative solution is to reuse the instruction results between the cache miss load and the misprediction. When the valid instructions are re-dispatched to the back-end, they are assigned the same entries in the ROB and LSQ. Therefore, the computed result is reused if the instruction has been executed. The column of "reuse" in the Table shows the EDP result compared with the checkpoint architecture. This new design is more energy efficient.

Table 3.4
The EDP results of 2-phase with or without instruction reuse, compared with a 8-checkpoint architecture

	2-phase	reuse		2-phase	reuse
perlbench	96.99%	96.49%	bzip2	98.99%	98.35%
gcc	97.82%	97.42%	mcf	100.30%	98.19%
gobmk	97.65%	97.48%	hmmer	106.06%	103.02%
sjeng	97.37%	97.20%	libquantum	98.23%	98.23%
h264ref	98.66%	98.61%	astar	101.07%	99.16%
bwaves	99.57%	99.53%	milc	100.12%	100.04%
zeusmp	98.62%	98.54%	gromacs	96.63%	96.63%
leslie3d	97.66%	97.63%	namd	99.31%	99.30%
GemsFDTD	99.96%	99.96%	tonto	98.89%	98.86%
lbm	99.96%	99.95%	wrf	99.84%	99.79%
sphinx3	99.61%	99.04%	gmean	99.19%	98.72%

3.5 Summary

The proposed architecture Two-Phase attempts to reduce the misprediction recovery penalty in two different scenarios. For the most common case, a simple and efficient basic recovery mechanism is used. For the case in which the retire stage is blocked by a long latency operation, a *refetch-all* recovery mechanism is used to overlap with the long latency execution. This Two-Phase recovery algorithm has a performance very close to the state-of-the-art recovery mechanisms but its implementation is significantly simpler. The structures which are used in phase one recovery can be fully reused during phase two of the recovery in an efficient way. No special tags nor checkpoints are necessary and no selective flushing is required.

Chapter 4

Passing Loop : Reducing the Pipeline Fill Delay

4.1 Overview

Reducing the pipeline fill delay is possible by exploiting the control independence of instructions [28, 62]. This is because after a branch misprediction, instructions which are control independent of the mispredicted instruction are fetched again. Since many of the instructions' operands also remain the same during the misprediction, re-executing them will yield the same results. Exploiting control independence can be accomplished by either reusing the result of the instructions or fetching only control dependent instructions and re-executing them. Therefore, we classify the recovery techniques into two categories. If the computation results on the wrong path are reused during the recovery, the technique is classified as **Repair-Recovery**. If the computation results are all eliminated during the recovery, the technique is classified as **Restart-Recovery**.

4.1.1 Control Independence

Repair-Recovery is attractive since a large fraction of instructions have the same computation results irrespective of the branch direction. This is because the diverged paths after a branch instruction converge at a later point and the instruction execution on this converged path is not controlled by this branch instruction. If these instructions are both Control Independent and Data Independent (CIDI), they do not need to be re-executed.

Figure 4.1 shows an example. When the branch at the top of the hammock is mispredicted to be taken, it follows the left path and incorrectly executes I1 in the Control Dependent (CD) region. In the figure, I2 and I3 are in the CD region of the alternative path, which is also the correct path. I4 and I5 are in the Control Independent (CI) region but their operands come from the CD region so they are Control Independent but Data Dependent (CIDD). I6 is CIDI as its operands do not come from either CD

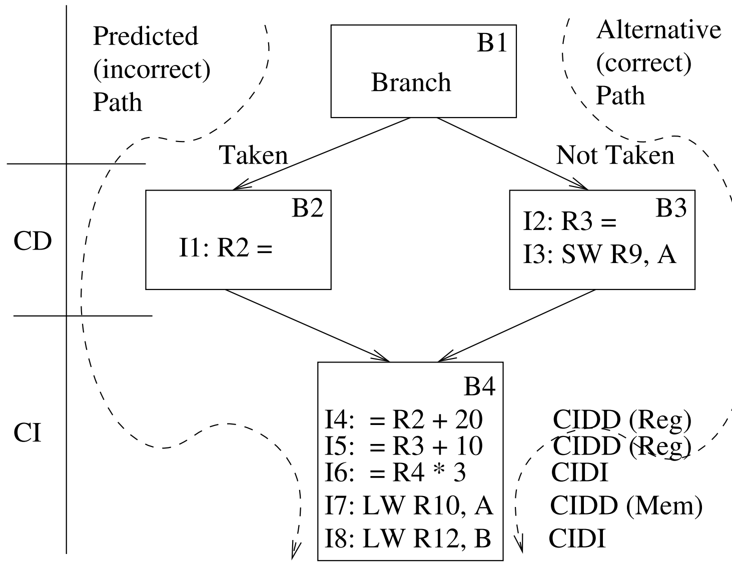


Figure 4.1: Control Dependence/Independence

region. In addition, memory instructions such as I7 and I8 pose a specific challenge since identifying the set of memory instructions which need to be replayed requires further analysis. In the above example, although I7 computes the same address in both paths, a write to this address by a store alongside the alternative path incurs the re-execution of I7 whereas I8 does not need to be replayed as it is a CIDI instruction. With appropriate mechanisms in place, the misprediction can be repaired by fetching and executing the correct CD region and re-executing the CIDD section.

4.1.2 The Convergence Point Prediction

In order to reuse the squashed instructions, the first task is to provide the convergence point after which the CI instructions are located. The convergence point is not a fixed position for every branch in the run time. Therefore, the convergence point has to be predicted ahead and verified later. Even though the accuracy of the convergence point prediction is very high, the related hardware cost is not avoidable.

4.1.3 The Affected Register Mapping

The detection of the CIDD instructions is relied on the CD instructions' observation. The logical registers which are defined in the CD region, including the correct and incorrect paths, are all marked poisoned. Any CI instructions which consume the poisoned registers are labeled as CIDD as their computation results are still dependent on the branch direction. Moreover, the logical registers which are defined by the CIDD instructions are also marked poisoned. Other than CIDD instructions, the rest CI instructions are CIDI instructions.

In reality, each branch has its own CD region and CI region. It is very costly to keep the poisoned register numbers and the CIDD instructions for every branch. Therefore, a typical procedure is to predict the poisoned registers and CIDD instructions when a misprediction is detected. The hardware cost for the number of CIDD instructions can be enormous.

4.1.4 Correct Instruction Insertion

A common control independence recovery is composed of several steps: eliminating the wrong path instructions in the CD region; inserting the correct path instructions in the CD region; re-executing the CIDD instructions with the correct operands. Notice that the correct CD instructions are older than the CI instructions in program order. Therefore the insertion is out-of-order and has to be cautiously handled.

For a ROB-based superscalar processor, the instructions are maintained in program order in ROB. Therefore, the resources required by the wrong path instructions have to be preserved upfront. By doing this, the processor will have enough room to insert the correct CD instructions and still maintain the instructions in program order. However, if the branch is correctly predicted, the preserved resources will not be used. This can be detrimental to the overall performance as the effective ROB entries shrink.

4.2 Related Work

In the last section, a brief overview about control independence is described. In this section, the detailed history and related work is explained. In general, there are two different approaches aiming at control flow instruction execution: Eager Execution and Control Independence. Eager Execution would execute both paths of instructions when a branch was encountered. The processor would eliminate the wrong path instructions while the branch is resolved. Control Independence would only execute the predicted path instructions and reuse the CIDI instructions during the recovery.

4.2.1 Eager Execution

Riseman and Foster did a limited study assuming a machine had infinite resources and could execute each instruction at the earliest possible moment [63]. What they found is that even with a machine that has infinite resources, the control dependence immensely restricts the potential ILP (Instruction Level Parallelism). Their evaluation showed that if the instructions after a branch had to wait until the branch was resolved, this infinite machine only ran 1.72 times as fast as a conventional machine. On the contrary, if the branch directions were known ahead, this infinite machine ran 51 times as fast as a conventional machine. The assumption in this work seemed to be impractical but it showed the potential to go beyond control dependence limitation.

Lam and Wilson did another study to analyze a more reasonable processor [64]. Their conclusions were that: I) Local regions of instructions have limited parallelism; II) Higher performance can be achieved by executing independent regions of instructions concurrently; III) Speculative execution is very important to break control dependence constraints. Their work had a similar conclusion as the previous one (Riseman and Foster). It also gave a new perspective to exploit different region parallelism which is the TLP (Thread Level Parallelism).

Todd and Gurindar [65] designed the dynamic dependency graph (DDG) to analyze a sequential execution of the program. Their studies indicated that there was a useful amount of parallelism in the benchmarks. But to fully expose this parallelism requires large instruction windows and the ability to rename both registers and memory.

Augustus described a hardware solution to reduce control-flow inhibitors of concurrency in sequential instruction streams in his dissertation [66]. Thereafter, ShouHan and Augustus proposed Eager Evaluation to insert predicates to convert control dependence into data dependence [67, 68]. The idea of eager execution is to execute the instructions on both paths after a branch and only commit the correct path instructions when the branch is resolved. When multiple unresolved branches are in the pipeline, this is multi-level eager execution. Uht et al. then developed Disjoint Eager Execution (DEE) to more efficiently exploit instruction level parallelism [69, 70]. DEE combined branch prediction with eager execution and a cascaded branch prediction accuracy is provided. Due to the increase of the depth of branch predictions, this accuracy is significantly decreased. Therefore, an alternate path from a previous branch may have a more competitive prediction accuracy and should be fetched next.

A similar approach was implemented later [71].

Since DEE was developed, many other multi-path execution techniques were implemented. Heil and Smith proposed Selective Dual Path Execution (SDPE) [72] to execute the instructions from both paths when a low confidence branch was encountered. A branch forking policy was incorporated when a low confidence branch prediction was met while two paths were already being executed. Tyson et al [73] evaluated the potential of dual path execution coupled with a branch confidence predictor. their results implied that dual path execution, which was thought to be excessively resource consuming, might be a worthy approach if restricted with an appropriate predicting set. Ahuja et al. did a similar research to assess the potential of multi-path execution [74]. They showed in their evaluation that using four paths and a relatively simple confidence predictor, multi-path execution gathered speedup of 14.4% for the Spec Int suite. Klauser et al. designed PolyPath architecture to execute instructions from multiple paths simultaneously in the same processor pipeline [75].

Dual path or multiple path execution requires the processor to have a high throughput fetch unit as more instructions are fetched. Artur and Dirk [76] designed a mechanism to support multi-path instruction fetching with realizable hardware cost. In order to efficiently execute dual-path instructions and minimize the design overhead, Aragon et al. designed Dual Path Instruction Processing (DPIP) [77]. The alternative path instructions were fetched, decoded, renamed but not executed. These instructions were not executed until the branch is mispredicted.

Mahlke et al. considered a partial predicated execution and compared it against a fully predicated execution [78]. Partial predicated execution, such as conditional moves, requires very little change to the existing architectures and still can provide substantial performance improvement. Modern processors have the ability to execute multi-thread programs in one core. Therefore, a more effective way to execute dual-path instructions is to dispatch the different paths onto different threads. Wallace et al. proposed to use multi-thread processor to execute multiple path instructions [79].

Selective predication execution is an alternative of eager execution. Srinivas and Alexandru had a method by using profile data to selectively do if-conversion [80]. Similarly, Weihaw and Brad developed a mechanism to predict predication [81]. Their mechanism could provide a predicted value for the predicate before it is computed. Therefore, the data dependence was collapsed. The recovery process was also very simple since the data dependence was already integrated in the predicate code. Kim et al. designed the mechanism of Wish Branches [82] to include both branches and predications in the binary code. During the run time, the selection was done by the branch confidence prediction. If a branch was less likely to be correctly predicted, its predicated code would be issued. Quinones et al. optimized the predicate prediction mechanism [83].

4.2.2 Control Independence

First of all, Avinash and Gurindar proposed dynamic instruction reuse [84] to reuse any instructions which have the same inputs. Clearly, CIDI instructions have the same inputs and their computation results were reused in this technique. Rotenberg et al. then adopted this idea and applied it to exploit control independence [28, 62]. In their work, they discussed the important implementation issues and some possible hardware solutions. They showed that exploiting control independence can close the performance gap between real and perfect branch prediction by as much as half. Chou et al. had a similar study to exploit the benefit of control independence [85].

Since the concept of control independence was first introduced, many related mechanisms were proposed to take advantage of it. Chen-Yong and Vijaykumar designed Skipper [86] to skip over control dependent region when a low confident branch was encountered. Therefore, the CI section was fetched first and the correct CD section was not fetched until the branch was resolved. This mechanism is very similar to the idea of branch delay slot [87] except Skipper can fetch CI section dynamically and branch delay slot has to do it in the compiling time and the number of CI instructions is also limited.

In all techniques which exploit control dependence, the convergence point is needed and should be provided in an efficient way. Collins et al. developed a method to predict the convergence point [88] which can achieve 95% accuracy with 4KB storage.

Gandhi et al. had a different perspective and designed selective branch recovery (SBR) [89] to focus on the exact convergence. The exact convergence means the convergent point happens to be at the beginning of the alternative path. In other words, the alternative path does not have any instruction in the CD region. SBR does not need to insert correct CD instructions and the detection of CIDI instructions is also straightforward.

Al-Zawawi et al. proposed transparent control independence (TCI) [90] to maintain all of the potential CIDD instructions in an external buffer. Therefore, during the recovery, the CIDD instructions were filtered out from this buffer which is much faster than scanning the whole CI section. Andrew and Amir designed Ginger [91] to exploit control independence. Ginger could rewrite the dependent operands for the CIDD instructions so that the re-execution phase was much quicker during the recovery process.

SYRANT [92] was proposed later by Nathanael and Andre. This mechanism was used to preallocate resources for alternative path instructions and one unique feature of SYRANT is the CI instructions would be allocated to their original entries in ROB and LSQ. This simplified the recovery procedure significantly.

4.3 The Concept of Passing Loop

At a high level, a branch can control two CD dependent blocks of code, such as blocks B2 and B3 in Figure 4.1. But at the machine-level, a branch instruction can only control the execution of a single block of code, namely, the code in its fall-through. More complicated control-flow structures are synthesized by placing control-flow instructions such as branches and jumps in the CD section of another control-flow instruction. Such instructions are defined as Control Dependent Control Flow instructions (CDCF). Without CDCF instructions, repair-recovery is fairly straightforward. Either the branch was predicted *not-taken* and is actually *taken*, or, it was predicted *taken* and is actually *not taken.* The former is named as *Scenario-1* and the latter as *Scenario-2.* A Scenario-1 misprediction can be handled by invalidating the fall-through block and a Scenario-2 misprediction can be handled by inserting back the omitted fall-through block while establishing correct data dependencies.

Unfortunately, the presence of CDCF instructions makes such a simple repair-recovery impossible to implement, unless the control-flow structure of the program is analyzed and the hardware is designed appropriately to handle a specific control-flow structure, such as an if-then-else construct.

Passing Loop approaches this problem from a unique perspective. Instead of targeting specific high-level control-flow structures and recognizing them, we attempt to buffer the CD section of all short-distance control-flow instructions and attach a *validity guard* to each fetched instruction. Supported by a guarded processor back-end, the processor continues to fetch sequentially upon encountering a short-distance forward branch irrespective of its predicted direction. If the branch is predicted to be not-taken, any instructions fetched until the target is reached are marked *valid*. Otherwise, they are marked *invalid*. In other words, each short-distance forward branch *guards* its CD section with the inverse of the predicted direction. If there is a misprediction, the validity bit of the corresponding CD instructions can be toggled. Hence a Scenario-1 misprediction is handled by simply changing the validity of the fall-through block to *invalid* and a Scenario-2 misprediction is handled by changing the marking of the block to *valid*.

A transition from *valid* to *invalid* can be handled by undoing the effects of those instructions and eliminating them while a transition from *invalid* to *valid* necessitates re-execution of the instructions in the block. Passing Loop's novelty rests on how CDCF instructions are treated and handled since it can "re-execute" a short-distance CDCF instruction, just like any other instruction by toggling the validity information of dependent instructions.

Figure 4.2 shows an example where all control-flow instructions jump a short-distance. The validity of instructions for any branch direction combination is illustrated on

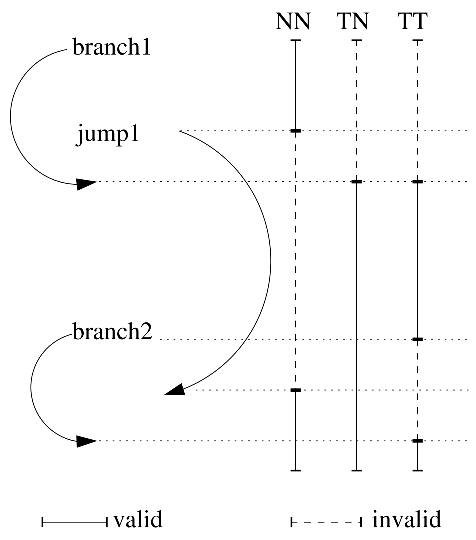


Figure 4.2: CDCF

the right. If a misprediction is detected, Passing Loop will transition the validity information from the predicted direction (say TT) to the actual column (e.g., TN or NN). When all CDCF instructions are short-distance, the mechanism can recover from any misprediction with repair-recovery. The fetch unit combines the current validity information with the branch predictor output to set-up the initial validity bits. For example, if *branch1* is not taken, CDCF instruction *jump1* will be *valid*. Since it is valid, its CD section will be invalid, making *branch2* an invalid CDCF instruction, which in turn makes NT an invalid combination. Passing Loop therefore forces all invalid CDCF instructions to be not-taken, ignoring the branch-predictor output. If the invalid CDCF instruction jumps a short distance and becomes valid later, repair-recovery can wait for the actual outcome (non-speculative repair), or, use the branch predictor's initial prediction at repair time (speculative repair) to repair it. If the invalid CDCF jumps a long distance, a restart-recovery may be necessary when such a branch becomes valid.

As it can be seen, this fetch mechanism hits many birds with one stone: (1) Since the mechanism always yields a sequential image of the relevant section of the code in the issue window, it permits "re-execution" of CDCF type branch instructions. In order to "re-execute" any branch, it is sufficient to toggle the validity status of instructions encapsulated in the CD section of that branch. (2) Due to its ability to "re-execute" branches, Passing Loop can recover from a misprediction of any short-distance branch *without detecting the shape of the control-flow*; (3) Although we can recover from the misprediction of any short-distance forward branch, there is no exponentially growing state complexity as in multi-way fetching; (4) The *confluence point* is always the target address of the branch and there is no need to predict the confluence point; (5) Finally, due to sequential fetching and allocation of resources such as reorder buffer positions or load store queue entries to all instructions irrespective of their validity, there is no need to pre-calculate the necessary resources and reserve them up-front in preparation for a misprediction.

4.4 Microarchitecture

Figure 4.3 illustrates the Passing Loop microarchitecture. The architecture incorporates a *Store Set* memory dependence predictor [16] for handling memory dependencies and the Global Branch History Register (GHR) is updated speculatively as illustrated. The architecture is a *guarded architecture*. A valid bit is kept in each ROB and LSQ entry. Only valid instructions can update the Front Register Alias Table (F-RAT), Reservation Station (RS), Last Fetched Store Table (LFST) and Global Branch History Register (GHR) in the front-end and only valid instructions are renamed by the renamer. Invalid instructions are allocated a physical register but their source operands are not renamed. A special register called *Dummy Fetch Target* (DFT) is used to track the target address of predicted taken short distance control flow-instructions. In the back-end, only valid instructions can update the Retirement Register Alias Table (R-RAT) and write to memory.

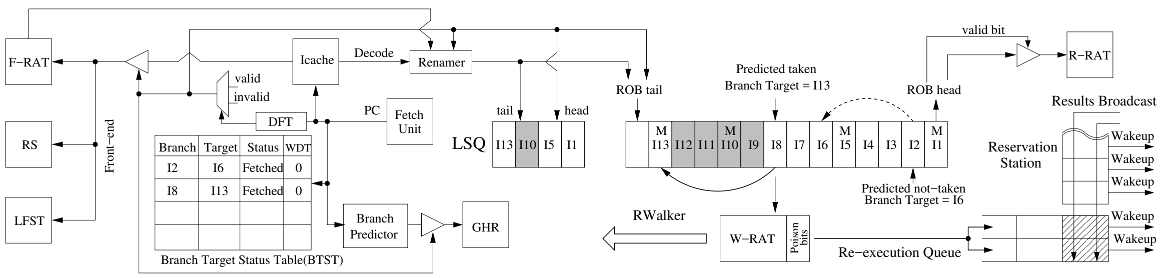


Figure 4.3: Passing Loop Microarchitecture Block Diagram

A set of *Re-execution Queues* shown on the right of the figure are used to dispatch instructions which need to be re-executed during a repair-recovery. Only the head instruction is checked for readiness. These queues are manipulated in-order in a manner similar to [93] with the exception that queue heads are not directly connected to execution units. Instead, each queue head is treated as if it is a reservation station entry by the superscalar issue logic. As a result, the re-execution path shares the same select logic and adds minimal complexity. We also use slightly modified heuristics during instruction steering: a dependent instruction is always steered to the queue where the producer is. The instructions in Re-execution Queues are given priority during selection as they are usually older than the instructions in RS.

The fetch unit shown in the middle of the figure supplies the PC to the I-cache, BTB and the Branch Predictor. A very small, fully associative table called *Branch Target Status Table* (BTST) is used for tracked control-dependent blocks and establishing instruction validity information. Only short-distance forward control-flow instructions update the BTST.

In the figure, the shaded entries represent invalid instructions. I2 is a branch which is predicted to be not-taken and I8 is another branch which is predicted to be taken. As a result, I3 to I5 are valid and I9 to I12 are invalid. I1, I5, I10 and I13 are memory instructions marked in the ROB and the LSQ. Since I10 is invalid its corresponding entry in LSQ is also shaded.

In order to properly rename and execute CD and CIDI instructions, Passing Loop relies on in-order processing by using a special *Walker* which we refer to as the Recovery Walker (RWalker) to differentiate it from the Walker in prior work which speeds-up restart-recovery [26]. Rwalker shown at the bottom of the figure scans the instructions between the ROB head and the ROB tail using its own register alias table (W-RAT) where each entry is extended with a single *poison bit*. In this manner, CIDD instructions which are register dependent are easily detected using the *poison* bit and tracked poisoned instructions, also in program order. By walking from the head of the ROB towards its tail, RWalker renames the alternative CD path and CIDD instructions to their correct mappings and if any instruction needs to be re-executed, sends it to the re-execution queue from which the instruction is rescheduled to issue and execute. Required in-order processor state for RWalker for renaming and issuing is always available in W-RAT which runs ahead of R-RAT and permits misprediction recovery before branches reach the head of ROB.

4.4.1 Front-end: Fetching Algorithm

Passing loop follows a very simple fetch policy. Whenever a short-distance forward control-flow instruction is encountered, the target address of the instruction is saved in BTST and the fetch unit continues to fetch sequentially until the target is encountered.

Each entry in the table contains information about a short-distance, forward, unresolved branch, supplying 4 fields, *Branch*, *Target*, *Status* and *Timer*. The *Branch* field contains the unresolved branch tag which is used to signal the status changes to the branch. The *Target* field contains the branch target address. The *Status* field indicates the current state of the branch, and can assume one of two values, *Pending* and *Fetched*. An initial allocation sets the value to *Pending* and when the branch target is reached, the entry is updated to *Fetched*. Allocated BTST entries are removed when the corresponding branch instruction resolves or the branch is disqualified. The information that the current control-flow instruction is a short-forward branch or an unconditional direct jump is kept in the Branch Target Buffer (BTB) by using a single bit and all qualified branches are allocated BTST entries unless BTST is full.

As each instruction is fetched, it searches BTST using its PC and a hit signals the end of the block to the fetcher. However, this algorithm alone does not guarantee that the branch target will be met before a misprediction is detected. For instance, if there is a function call in the CD section of a short forward branch and the function contains more instructions than the processor pipeline can hold, the branch target may not be met before the misprediction is signalled. BTST *Status* bit must be therefore set to *Fetched* before a repair-recovery can be initiated. The *Timer* entry is used by a watchdog timer which counts the cycles used for meeting the target. If the timer exceeds a pre-configured threshold, the associative branch is disqualified. As it should be clear, this timer is not required for correctness but it helps to minimize unnecessary buffering of large CD sections under such scenarios. Note that if a branch is disqualified from repair-recovery, the processor state is always appropriate for applying a conventional restart-recovery. Therefore, when a branch is disqualified, no further action is necessary.

During the fetching process, if a conditional branch instruction is predicted to be taken, or an unconditional jump is being handled, the code in the fall-through is the alternative path and these instructions are marked as *invalid* in the ROB. All other instructions will be *valid*. In the predicted taken cases, while seeking the branch target, it is possible that additional branch instructions are encountered in the region that would be marked as invalid, such as the section(*I9-I12*) in Figure 4.3. The invalid branch instructions are treated as if they are predicted not-taken including unconditional jumps and the output from the branch predictor is ignored as discussed previously. This approach guarantees that the target address of the branch (i.e., *I13*) is definitely reached and becomes part of the program execution path. As a result, there will be at most one pending predicted taken branch instruction which target is being sought. This target is kept in the Dummy Fetch Target (DFT) register. Once the fetch stream reaches the DFT value, the following instructions are marked as *valid* and new branch instructions are predicted normally.

Passing Loop implements a unified resource allocation policy. All fetched instructions are allocated an ROB entry and all fetched memory instructions are allocated an LSQ entry. Similarly, all result producing instructions are allocated a physical register, irrespective of their validity status. These reserved resources are used when the corresponding branch is mispredicted and invalid instructions become valid. This approach significantly simplifies the resource management during the recovery process, particularly for ROB and LSQ. When invalid instructions are retired, their physical registers are released except that they release their current physical registers instead of the physical registers previously assigned to the same logical register. All instructions release their ROB and LSQ entries when they retire.

4.4.2 Handling Memory Instructions

When a load instruction is decoded, it consults the store set memory dependence predictor. If the load is not dependent to any prior stores, it is free to issue when its operands are ready. A speculative load buffer is implemented as the one described in [94]. This algorithm saves the address and the hardware pointer for the load into a *Speculative Loads Table* (SLT) whenever a load issues. The allocated entry is removed when the load is committed. Committing store instructions write to the data cache and check SLT with their addresses. A hit indicates a memory order violation and the exception bit of the corresponding load instruction is set.

Load misspeculations are treated as CIDD instructions by *RWalker*. Therefore when

RWalker encounters a load with its exception flag set, it re-executes the load and poisons its destination logical register, which in turn forces to re-execute load instruction's dependent instructions.

In Passing Loop, a store instruction in the CD section may alter memory ordering. If a store instruction's validity changes from *valid* to *invalid*, *RWalker* clears its dependent loads so that these loads can issue earlier. It also needs to flag as exception any of the dependent loads to which store might have forwarded the store data. To identify any of the dependent load instructions, *RWalker* searches SLT and sets the exception flag for any matching stores. If the validity of the store instruction changes from *invalid* to *valid*, it is impossible to signal the younger load instructions which may be dependent on this store and as a result a memory order violation may occur. The SLT mechanism detects such violations at load retire time.

4.4.3 Back-end: Misprediction Recovery

Repair-recovery is implemented by *RWalker* which runs ahead of the ROB head pointer, examines and recovers each instruction in program order towards the tail. During this process any incomplete non-branch instruction can be skipped over since the design incorporates a separate R-RAT and ROB head pointer. If a skipped incomplete instruction gets an exception later, exceptions will still be precise and taken care of in a conventional manner.

Upon encountering an unresolved branch instruction, there are two options for *RWalker*. Either the walker waits for the branch resolution before moving to the next instruction, or the walker can rely on initial branch prediction and continue. The drawback of waiting for branch resolution is the extra delay on recovery process. The drawback of speculative processing is, a repair-recoverable branch left behind is no longer repair-recoverable. In order to minimize the number of restart-recovery recovered mispredictions in the speculative case, we reset RWalker back to the ROB head when a walk is completed, reset the poison bits, copy R-RAT onto W-RAT and resume walker operation. We evaluate both the non-speculative and speculative versions as outlined.

Another design configuration is *RWalker* Fetch Unit interaction. Since repair-recovery uses instructions already in the pipeline, it is possible to continue fetching new instructions while the repair-recovery proceeds, unlike restart-recovery. Alternatively, instruction fetching can be halted until the recovery is completed, as it is the case with restart-recovery. The former maximizes fetch utilization but will result in inaccurate global branch history. The latter misses the opportunity to fetch more instructions during recovery but when the recovery is complete the processor state is precise. We have experimentally verified that the former almost always results in worse performance due to additional branch mispredictions. Therefore, when a repair-recoverable misprediction is detected by *RWalker*, new instruction fetching is stopped until the recovery is complete, i.e., *RWalker* reaches ROB tail. At that time, the precise state is copied from W-RAT to F-RAT and instruction fetching resumes.

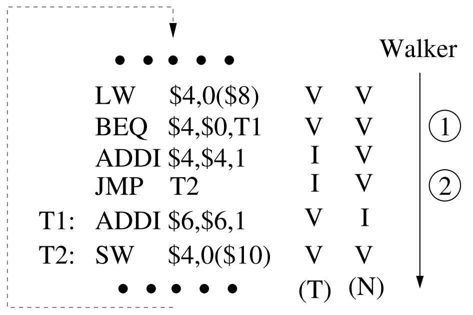


Figure 4.4: Recovery through walking

To illustrate *RWalker*'s operation, we use the code fragment shown on the left in Figure 4.4 which adds one to the non-zero elements of an array and counts zero elements. The example contains a CDCF, JMP T2 in the CD section of BEQ. The validity bits are shown next to each instruction for both taken and not taken scenarios. A misprediction implies the fetch unit assigned bits were the set (N) but should have been (T) for Scenario-1 and they were the set (T) but should have been (N) for Scenario-2.

When *RWalker* encounters the branch at point 1, it will wait for the branch to execute since it is a *valid* branch. If the computation result is taken, *RWalker* will record its target PC and mark the following instructions as *invalid* until the branch target is encountered in T1. Otherwise, the instructions will be marked as *valid*. If an instruction becomes *invalid*, the instruction is invalidated in the rest of the pipeline, including ROB and LSQ. If the *invalid* instruction is still in RS it will be removed. The destination physical registers of these instructions are also marked as *unavailable*. A typical case involving a Jump instruction is at 2. If the Jump is *invalid* as it would be in the taken case, it is ignored like any other *invalid* instruction. Otherwise, it is treated as a taken branch, the instructions between the JMP's target and JMP are marked as *invalid*.

During the recovery process, W-RAT supplies the correct processor state for renaming instructions and tracking damaged values through the *poison bit*s attached to each logical register entry. A simple state machine drives the updating of poison bits. Upon an *invalid* to *valid* transition, the instruction is renamed using W-RAT, the poison bit of the destination logical register is set in W-RAT and the instruction is sent for execution. A *valid* to *invalid* transition causes the destination logical register of the instruction to be poisoned. Finally, upon a *valid* to *valid* transition, the source operands of the instruction are checked. If they are poisoned, the instruction is CIDD. The destination logical register is poisoned, then the instruction is renamed by W-RAT and scheduled for re-execution. If the source operands of the instruction were not poisoned, the instruction is CIDI and the poison bit of its destination logical register is cleared.

The renaming of instructions during recovery is illustrated in Figure 4.5. Initial *valid* bits and the updated *valid* bits are displayed in the middle of the figure and the

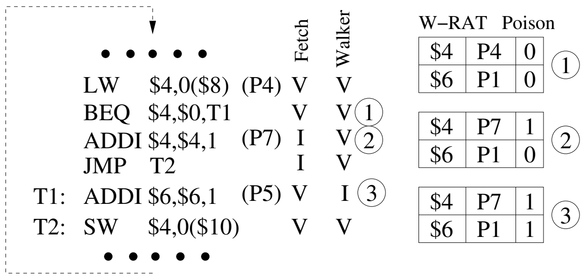


Figure 4.5: Detecting the damaged register mappings

corresponding W-RAT and poison bits are shown on the right. At point 1, $4 is mapped to P4 and $6 is mapped to P1 and neither of them are poisoned. At point 2, the instruction becomes *valid*. Therefore, $4 is mapped to P7 and it is now poisoned. This instruction needs to be executed. In 3, the instruction becomes *invalid* and $6 is poisoned. Any future reference to $6 needs to be re-executed and the correct mapping is P1. The last store instruction reads a poisoned operand $4 so it is renamed to P7 and re-executed.

4.4.4 CIDD redundancy

In some cases, poisoning logical registers is unnecessary due to *value redundancy*. The first case is when an instruction transitions from a *valid* state to an *invalid* state. If the value the *invalid* instruction updates is the same as the correct value, the destination logical register does not need to be poisoned. Although the register mapping is incorrect, the register value is correct. The second case occurs when *RWalker* finds an incomplete CIDD instruction, which implies none of the dependent instructions have executed yet. Under this condition, the CIDD instruction needs to be renamed and re-executed but its destination logical register does not need to be poisoned since its dependent instructions can get their operands from the re-executed result.

4.5 Experimental Results

4.5.1 Simulation Methodology

MIPS I Instruction Set with delayed branching removed is used as the simulation ISA. GCC 4.9.2 has been modified to generate and optimize for this instruction set. To evaluate the performance of Passing Loop, ADL [46] is used to generate both the baseline and Passing Loop cycle-accurate simulators. These simulators respect timing at the RTL level. The baseline processor uses centralized scheduling to broadcast computation results and wake-up/select is completed in a single cycle. Load and store instructions are issued directly to memory units since address computation is done via splitting the computation into another μ-op.

Power values have been obtained by adapting Wattch[49] to the ADL simulator framework. The power results have been validated against the McPAT[50] tool tested with a very similar superscalar pipeline to ensure correctness using the baseline processor model. Passing Loop design power evaluation was carried out by collecting different structures' active counters and feeding them into the core of Wattch through which the power consumption is evaluated. Wattch is configured to consume less power(10%) when a particular port is not used in that cycle. For Passing Loop, extra energy is consumed by RWalker when it repairs the pipeline. The energy consumption for re-renaming an instruction is equivalent to a front-end renaming and the re-issuing energy consumption is equivalent to front-end issuing. The expanded ROB is fully modelled and the BTST is modelled as a CAM structure.

Table 4.1
Spec 2006 Benchmark Suite

Spec 2006 Int	perlbench bzip2 gcc mcf gobmk hmmer sjeng libquantum h264ref astar
Spec 2006 Float	bwaves milc zeusmp gromacs leslie3d namd GemsFDTD lbm wrf sphinx3

For power and performance evaluations, Spec2006 benchmark suite has been used (Table 4.1). The suite was compiled with gcc version 4.9.2 and the maximum optimization setting(-O3). Binutils 2.24 was used as the software environment and the benchmarks were linked to the compact system library uClibC version 0.9.33. All of the benchmarks were run with the ref inputs and the first 500 million instructions were used to warm up the branch predictor and the cache. Performance data was collected over the next 1 billion instructions.

The baseline superscalar processor configuration is listed in Table 4.2. Passing Loop has an identical configuration with the exception of structures specific to Passing

Table 4.2
The Processor Configuration

ROB / RS / LSQ / PRF	128 / 64 / 32 / 128
Fetch / decode / issue	8 / 8 / 8
Cache	32KB L1 Icache/Dcache, 512KB L2 Cache, 10 cycles L1 miss, 100 cycles L2 miss
Pipeline length	15 cycles
Int ALU / Int Mul	1 cycle / 3 cycles
Int Div, FP ALU	7 cycles
Branch Predictor	8 kB TAGE
Minimal Recovery	15 cycles
Memory Ordering	Store Set, 4K SSIT, 256 LFST
Branch Recovery	Run-ahead Walker

Loop. Passing Loop has a 16 entry BTST and 2 re-execution queues as a default setting. The walker implemented in Passing Loop has the same width as the fetch unit allowing it to scan 8 instructions per cycle. Additionally, the recoverable forward branches can only contain up to 16 instructions between the branch target and the branch itself. The watchdog timer is set to 4 cycles. Moreover the CIDD redundancy mechanism mentioned in Section 4.4.4 is also implemented in Passing Loop to reduce the number of unnecessary CIDD instructions.

A TAGE predictor was integrated [14] and the simulation result of the 1st Championship Branch Prediction matched the reported log file attached with the design file. Table 4.3 illustrates our 8kB TAGE predictor misprediction rates compared with a 4kB Gshare predictor and the one reported in SYRANT [92]. In most of the benchmarks, TAGE predictor has less than half the mispredictions provided by the Gshare predictor with exception of *401.bzip2*, *456.hmmer* and *473.astar*. These 3 benchmarks contain a large number of unpredictable branches. The TAGE predictor used in SYRANT works much better than our TAGE predictor. Possible reasons include the larger storage used to track a longer history in SYRANT implementation as well as the use of Simpoints in SYRANT implementation.

Table 4.3
The branch predictor performance

MPKI	4kB Gshare	8kB TAGE	32kB SYRANT
perlbench	4.80	2.17	0.33
bzip2	11.12	9.05	2.91
gcc	7.40	2.42	0.78
mcf	3.05	1.67	8.32
gobmk	21.04	11.86	6.93
hmmer	11.81	11.01	9.05
sjeng	11.20	5.61	4.01
libquantum	0.01	0.01	0.05
h264ref	5.15	2.52	0.60
astar	31.90	26.86	11.14

4.5.2 Performance Analysis

Figure 4.6 illustrates the IPC speedup normalized to the baseline processor for both non-speculative and speculative RWalker. The geometric means for integer benchmarks are 3.29% (non-speculative) and 5.85% (speculative), for FP benchmarks are 4.14% (non-speculative) and 3.73% (speculative). Figure 4.7 shows the distribution

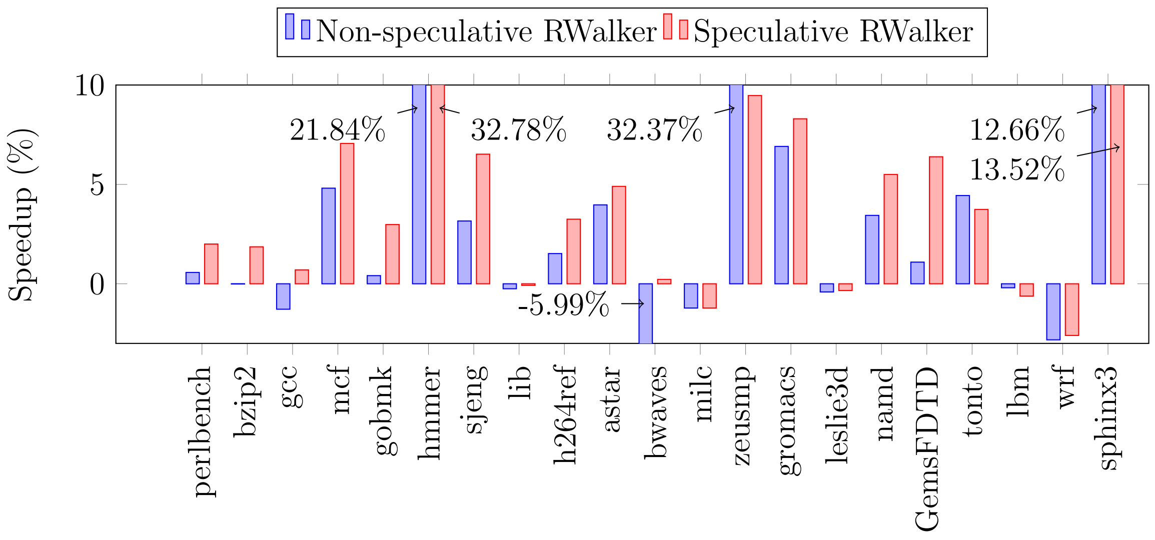


Figure 4.6: Spec 2006 IPC speedup

of 5 affected types of instructions per 1000 committed instructions for both non-speculative and speculative RWalker simulations. Wasted CD instructions are invalid when they are fetched and retired meaning they do not contribute anything and even occupy some resources. V2I instructions were valid in the fetch phase but they were invalidated during the recovery. Similarly, I2V instructions are those which were validated during the recovery. Through all the benchmarks, *astar* has more than 1000 affected instructions per 1000 instructions due to only counting valid instructions. Instructions such as wasted CD and V2I CD instructions are not counted as valid committed instructions.

In most of the benchmarks, the speculative RWalker performs much better than the non-speculative version since it can bypass unresolved branches and recover more quickly. The only exception is *zeusmp* where non-speculative RWalker improves the

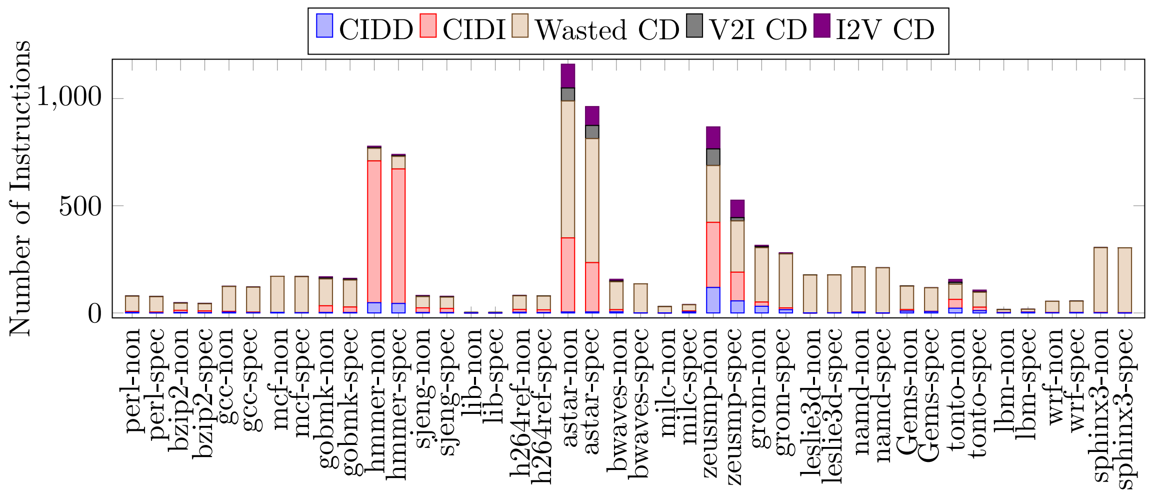


Figure 4.7: Number of affected instructions per 1000 valid committed instructions

performance by 32.37% but the speculative RWalker has less than 10% speedup. This large performance gap is caused by the number of skipped unresolved branches. It was noticed that speculative RWalker skips unresolved branches and these branches would not be re-scanned until RWalker recovers the whole pipeline and resets back to ROB head. If these skipped branches are mispredicted and reached by ROB head ahead of RWalker, they have to pay a full flushing misprediction penalty. This explanation is also supported by Figure 4.7. In *zeusmp*, the speculative RWalker has less than half the CIDI instructions of the non-speculative version. The rest of the benchmarks show similar behavior. Speculative RWalker skipping of unresolved branches result in fewer repair-recovery actions and hence fewer CIDI instructions. In Figure 4.6, some of the benchmarks perform worse, such as *bwaves*, in which the non-speculative RWalker is 5.99% worse than the baseline processor. We have described that before the pipeline is recovered, Passing Loop blocks the front-end to fetch and decode. In this scenario, a baseline processor can continue the fetch process although it has to reset its pipeline. Therefore if it takes too long for Passing Loop to recover, the baseline processor performs better. Speculative RWalker helps but not every time, such as in *milc*, *lbm* and *wrf*.

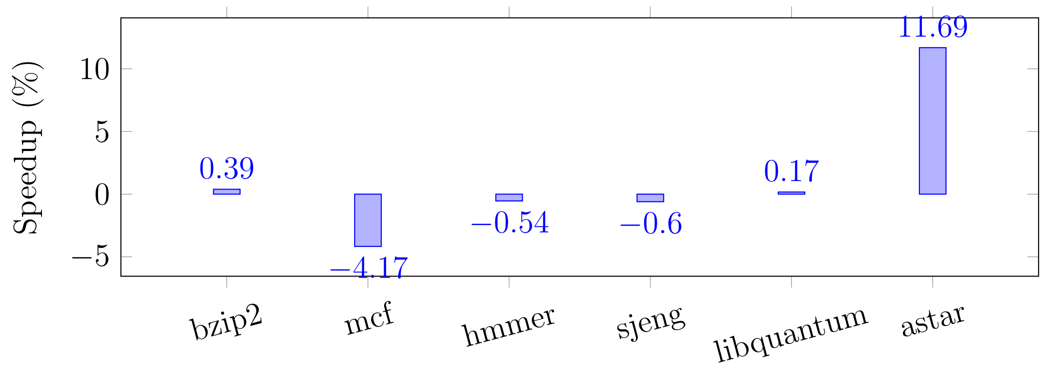


Figure 4.8: Speedup if all instructions retired are valid

Figure 4.7 shows a large number of wasted instructions were fetched, which consumed a substantial amount of resources such as ROB entries, which prevent other useful instructions. We analyzed this effect further and present it in Figure 4.8. We traced all the branch outputs and used a trace file to decide when the invalid fall-through instructions were fetched. This trace based mechanism does not work with some benchmarks which use random functions. Therefore those integer benchmarks are not shown. Once a qualified branch is predicted taken and the trace file confirms it is a correct prediction, no fall-through instructions are fetched and the program jumps to the branch target directly. Otherwise Passing loop would fetch the fall-through parts as they are required for recovery. From the figure, we see that *astar* achieves 11.69% speedup and Figure 4.7 shows a large amount of wasted instructions in *astar*.

We also found that *mcf*, *hmmer* and *sjeng* lose performance instead of gaining. We traced this to an unexpected effect of wasted instructions. They can warm up the instruction cache and do not affect the performance so long as the pipeline is not stalled by insufficient resources.

Table 4.4
The percentage of increased mispredictions by non-speculative Passing Loop

perlbench	-0.27%	bzip2	6.82%
gcc	3.77%	mcf	11.14%
gobmk	8.61%	hmmer	10.31%
sjeng	5.24%	libquantum	0.0%
h264ref	1.51%	astar	**69.03%**

Passing Loop has more branch mispredictions than the baseline processor and Table 4.4 displays the percentage of increased mispredictions compared to the baseline processor. Note that a recoverable misprediction is also counted as a misprediction in Passing Loop. The increased mispredictions are due to imprecise branch history information. Once a misprediction is detected, the younger branches in the pipeline have been predicted based on an incorrect branch history and there is no easy way to re-predict these branches. As a result, they are more likely to be mispredicted as the baseline processor can always use the correct history to predict branches due to its restart-recovery mechanism. We found that *astar* has close to 70% more mispredictions which explains why its speedup is limited even though it has a large number of CIDI instructions.

In order to examine the impact of issue width and ROB size, we evaluated Passing

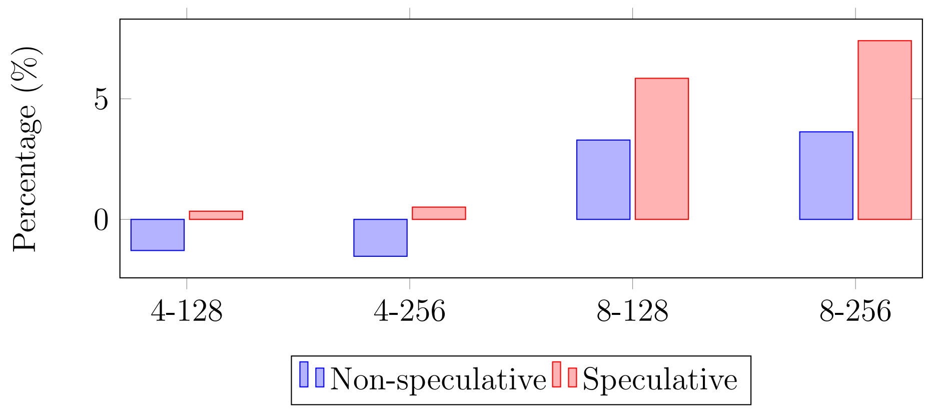


Figure 4.9: The geometric speedup over different configurations for integer benchmarks

Loop with issue widths of 4 and 8, and ROB sizes of 128 and 256 and the results are presented in Figure 4.9. RWalker width is equivalent to issue width and the physical register file size, LSQ entries are doubled in 256 ROB cases. Obviously, issue width significantly affects the performance, which means Passing Loop is highly dependent on the speed of RWalker recovery. Generally speaking, Passing Loop performs better with a larger ROB size as more mispredictions can be discovered and repaired in this configuration.

Table 4.5
The speedup comparison between Gshare and TAGE, geometric mean over integer benchmarks

	Non-speculative	Speculative
4kB Gshare	4.05%	5.76%
8kB TAGE	3.29%	**5.85%**

Originally we speculated that since a better branch predictor would have fewer mispredictions, Passing Loop's speed-up would be smaller. However from our evaluation results, it is clear that the prediction accuracy is not correlated to the potential speedup. Table 4.5 shows the two different branch predictor outcomes and the miss rates are presented in Table 4.3. Apparently, the speedup of the two different predictors are very close to each other, especially in speculative RWalker mode. The reason is Passing Loop's ability to recover short forward branch mispredictions and a significant fraction of these branch instructions are data dependent. Neither the Gshare predictor nor the TAGE predictor can predict these branches well. Hence Passing Loop is able to cover the mispredictions which are likely to be mispredicted.

Table 4.6
The speedup of different branch coverage (speculative RWalker)

	PL	PL+NDC	PL+NDC+BW
perl	2%	1.96%	2.19%
bzip2	1.86%	2.69%	2.83%
gcc	0.7%	0.64%	0.68%
mcf	7.06%	7.06%	7.06%
gobmk	2.98%	3.52%	3.68%
hmmer	32.78%	32.76%	32.77%
sjeng	6.52%	6.76%	7.15%
libquantum	-0.08%	-0.08%	-0.08%
h264	3.25%	3.3%	3.43%
astar	4.9%	4.9%	4.9%
Gmean	5.85%	6.01%	6.12%

Passing Loop only recovers short forward branches which are necessary for predicted taken branches. However in a predicted not taken case, the branch does not need to be a short distance forward branch, so long as the branch target can be met quickly. In other words, the diverged program can converge quickly which can be inspected by the watchdog timer. Therefore, we simulated a Passing Loop design in which every predicted not taken branch is qualified to assign a BTST entry and the speedup result is illustrated in Table 4.6 (No Distance Check). These results are normalized to the baseline processor. On the other hand, we found that a lot of mispredictions come from short backward branches such as a loop which has variable number of iterations at run time. These backward branches can also be covered by Passing Loop with some easy modifications. If a backward branch is predicted not taken, Passing Loop is unable to recover this misprediction as the CD instructions are not in the pipeline. If a backward branch is predicted taken, Passing Loop can record the next instruction address in BTST and jump to the branch target. If the recorded address is encountered within the watchdog timer threshold, it can be recovered by Passing Loop when it is mispredicted. We also evaluate Passing Loop covered by No Distance Check and Backward branch in Table 4.6. From the table, most of the benchmarks have the same performance improvements or very similar outcomes which means that short forward branches dominate the recoverable branches.

Section 4.4.4 discusses CIDD redundancy and we evaluate the value redundancy distribution in Table 4.7. It shows the fraction of CIDD instructions eliminated if the registers of some qualified instructions are not poisoned. The data is collected with a non-speculative RWalker as it has more CIDD instructions than speculative RWalker configuration. CD means that V2I CD instructions may not poison their destination logical registers if they compute the same results as the correct value. CI means a CIDD instruction does not need to poison its register if the instruction is not

Table 4.7
The CIDD redundancy

	CD	CI	CD & CI
perl	-0.46%	-0.52%	-0.97%
bzip2	-0.82%	-1.19%	-1.89%
gcc	-1.14%	-13.44%	-14.78%
mcf	0%	0%	0%
gobmk	-6.92%	-5.02%	-11.79%
hmmer	-0.29%	-7.82%	-8.02%
sjeng	-11.24%	-13.48%	**-22.24%**
libquantum	0%	-0.28%	-0.28%
h264	0.08%	-5.47%	-4.52%
astar	-3.14%	-1.13%	-4.26%
Gmean	-2.46%	-4.97%	-7.15%

completed at the time RWalker visited the instruction. The last column shows the combined results. Obviously, CI can reduce almost twice the number of CIDD instructions than CD and in *458.sjeng* close to one quarter of CIDD instructions are eliminated by this technique which reduces the pressure on the Re-execution Queues immensely.

4.5.3 Energy Efficiency

The energy-delay product (EDP) results are listed in Table 4.8 for both the non-speculative RWalker and the speculative RWalker. Passing Loop saves up to 40.49% EDP in *456.hmmer*. Given the IPC improvement of 32.78% from Table 4.6, we know that in this specific benchmark Passing Loop not only reduces the execution time but also saves the total energy consumption. This achievement is reached by reducing the

Table 4.8
The EDP compared with the baseline processor

	non-speculative	speculative
perlbench	3.75%	-2.08%
bzip2	0.7%	4.07%
gcc	3.89%	0.9%
mcf	-6.13%	-9.56%
gobmk	0.46%	-2.51%
hmmer	-35.77%	**-40.49**%
sjeng	-1.14%	-4.93%
libquantum	1.67%	1.57%
h264ref	0.04%	-1.81%
astar	-17.84%	-11.76%
Gmean	-5.92%	-7.65%

number of misprediction flushing penalties. If there are not a lot of recoverable mispredictions, Passing Loop might have a worse EDP such as *462.libquantum*. Table 4.6 shows that Passing Loop can barely have any speedup with *462. libquantum* but since BTST and other additional structures consume energy consistently, Passing Loop is not as power efficient with such benchmarks. Note that in most benchmarks, speculative RWalker is more power efficient than non-speculative RWalker except in *401.bzip2* and *473.astar*. In speculative mode, one CIDD instruction may be re-executed multiple times if there are several preceding mispredicted branches. From Table 4.4 we know that *401.bzip2* and *473.astar* have relatively more mispredictions. As a result the likelihood that CIDD instructions are executed multiple times increases, leading to more energy consumption.

4.5.4 Design Complexity

Table 4.9 shows the extra design resources implemented for these techniques and make a comparison between them.

Table 4.9
Design Resource classification

	Front-end	Back-end
Passing Loop	1 extra bit per BTB entry for short forward branch identification, CAM-like structure for BTST and each entry contain 30 bits target PC, 1 bit status, 2 bits watchdog timer and 7 bits target tag.	For each ROB entry, both operands' logical register numbers are added, plus one valid bit. Each logical register has 1 poison bit. Re-execution Queue.
Predicate Prediction	Multiple checkpoints of register mapping table in Rename-Replay mode or 2 dedicated tags per operand 1 dedicated tag per destination in Selective-Replay mode.	Recovery Queue which is the ROB in Passing Loop.
TCI	Re-convergent points predictor, influenced register set(IRS), Control-Flow Stack(CFS), 16-bit poison vector per logical register, Selective Re-execution Buffer(RXB) in which source operand's value is stored	Another CFS to detect the end of correct CD section, repair rename map, another set of poison vectors
SYRANT	Re-convergent points detection ABL/SBL, Resource Allocation on Not-taken and Taken paths table(RANT). Rename Sequence tag	The detail of the re-execution functionality is not elaborated in the paper.

4.6 Summary

As it is well-known, contrary to other types of speculation such as load speculation and value speculation, control speculation has the undesired property that after a misspeculation, the validity of instructions already in the pipeline becomes uncertain. We believe one of the most important contributions of the Passing Loop concept is its ability to make control-misspeculations similar to load or value misspeculations, albeit for a subset of control misspeculations. Because of this, it is possible to aim for a unified replay mechanism built around simplicity which can recover from all types of misspeculations. Indeed, Passing Loop's recovery mechanism built around a *RWalker* is a practical design which favors simplicity over performance. Speculative *RWalker* is an augmented design to further speed up the recovery process. The only shortcoming is *RWalker* has to reach ROB tail before it can be reset which is unnecessary and may not work well if the ROB size is too large. In the future, this issue should be discussed and further improved.

It is shown that Passing Loop is good at unpredictable branch recovery as evidenced by our experiments involving two different branch predictors. Passing Loop can always provide significant improvement in terms of performance and power efficiency, especially with a wider issue width and a large ROB.

Passing Loop takes advantage of both speculation and predication. It predicts the branch result to speculatively execute the following instructions. If the prediction is right, no penalty is triggered. Otherwise, Passing Loop has the ability to re-execute affected instructions instead of flushing away everything after the misprediction. Passing Loop can recover mispredictions in any kind of complex control-flow structure so long as it is built by short forward branches.

Chapter 5

Dynamic Memory Dependence Predication

Store-queue-free architectures remove the store queue and use memory cloaking to communicate in-flight stores instead. In these architectures, frequent mispredictions may occur when the store to load dependencies are inconsistent. This work, Dynamic Memory Dependence Predication (DMDP), is implemented to mitigate these misprediction effects.

[0]The material contained in this chapter will be published in Proceedings of the 45th International Symposium on Computer Architecture (ISCA "18).

5.1 Overview

In superscalar processors, store instructions do not update the memory subsystem until they commit. Therefore, a mechanism to bridge in-flight stores to in-flight loads is necessary and critical. Without this mechanism, in-flight loads and their dependent instructions would have to wait until all preceding stores commit. Most processors implement an associatively searched, age-ordered store queue to deal with the store-load communication. When a load is executed, the store queue and the data cache are simultaneously accessed. If the store queue does not contain a store instruction with the same address, the data read from the cache is used. Otherwise, the youngest matching store data is selected. This search latency dramatically increases as the number of in-flight stores grows. Although processor manufacturers do not release the search latency of their mechanisms, it is unlikely to be one cycle [18].

Despite the increase in access latency, each processor generation incorporates a larger store queue than its predecessors in order to exploit more instruction-level parallelism (Sandy Bridge 36, Haswell 42, SkyLake 56). In an attempt to reduce the access latency and improve scalability, several mechanisms were proposed to simplify the associative search operations [95, 96, 97, 98, 99, 100, 101, 102]. Others were designed to completely remove the store queue [17, 103, 104].

NoSQ [17] is one of these mechanisms which completely eliminates the store queue. The store in **NoSQ** is executed at the commit stage and then updates the cache. Therefore, the store is never issued to the out-of-order core. The in-flight store-load communication is accomplished through a memory dependence predictor. A load which is predicted to be dependent on a prior store is renamed to the physical register of that store (memory cloaking [12]). This collapses the DEF-store-load-USE dependence chain into a DEF-USE dependence chain. In order to verify the memory dependence prediction, the load is re-executed at the retire stage if necessary. Given prior history, if the memory dependence prediction confidence is low, **NoSQ** delays the execution of the load until the store is committed and the cache is updated. Therefore, these delayed loads need to be kept in a reservation station like structure and woken up by committed stores.

This work, Dynamic Memory Dependence Predication (**DMDP**), completely eliminates the unnecessary delays and the overhead of keeping delayed loads. The basic idea is to create a new MicroOp to compare the addresses of the predicted load-store pair. If the addresses match, the load uses the store data directly. Otherwise, the data read from the cache is used. As a result, the false dependence between the load execution and the store commit is removed.

Removal of this false dependence is significant because the store commit latency increases drastically when memory consistency models [105] are considered. Store-queue-free architectures eliminate the store queue which holds speculative stores before they are retired, but a store buffer still has to be provided to hold the retired stores until they update the cache. This buffer is essential for overlapping the penalty of store misses and properly implementing consistency models.

5.2 Motivation

It is possible to classify load-store dependencies into three categories: 1. Never Colliding (NC); 2. Always Colliding (AC); 3. Occasionally Colliding (OC). In NC, loads can always read from the cache without touching the store queue. For example, sweeping accesses through an array without changing the array values will generate many NC loads. Store-queue-free mechanisms also work perfectly in AC scenario due to the high dependence prediction accuracy. Examples of AC include register spilling, global variable accesses, stack accesses, etc. In contrast, OC is hard to predict since a correct prediction can not be achieved by simply observing the history of the memory dependencies. Figure 5.1(a) shows a code example of OC cases. In each iteration, a new pointer is read from an array and the pointed content is incremented. Figure 5.1(b) shows two nearby iterations in which the increment instructions collide whenever the two pointers are the same.

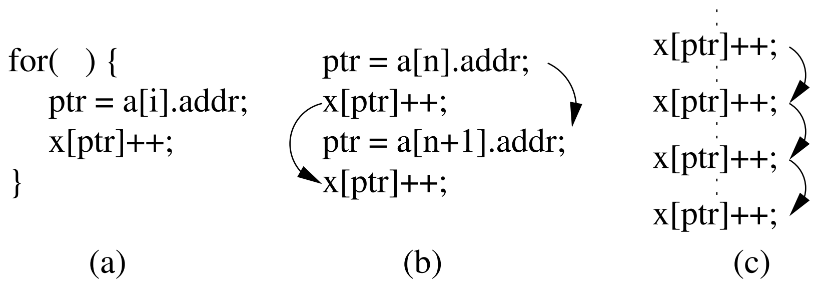


Figure 5.1: OC dependence caused strict ordering.

A common store queue free mechanism such as **NoSQ** initially would always read from the cache. When the first collision happens, the dependence is added and the increment instruction will be predicted to read the value from the previous iteration instead of the cache. However, if the pointers mismatch the next time, the forwarded data is probably wrong and the memory dependence is mispredicted as well. Frequent mispredictions on a certain load would impose a strict memory ordering. That is the load can not execute until the potentially aliasing store is committed. Figure 5.1(c) shows that the increment instruction will not execute until the previous one commits. This strict ordering makes sure the load reads the correct value regardless of whether or not the store and load addresses match. However, this strict ordering may significantly impede the program performance. First of all, if the store and the load addresses are different, the load and its dependent instructions suffer unnecessary latency. Even if the addresses are identical, the load has to wait until the store commits. Many unrelated events such as cache misses may delay the store committing and consequently affect the load execution time.

Figure 5.2 illustrates a breakdown of load instructions based on how they get their

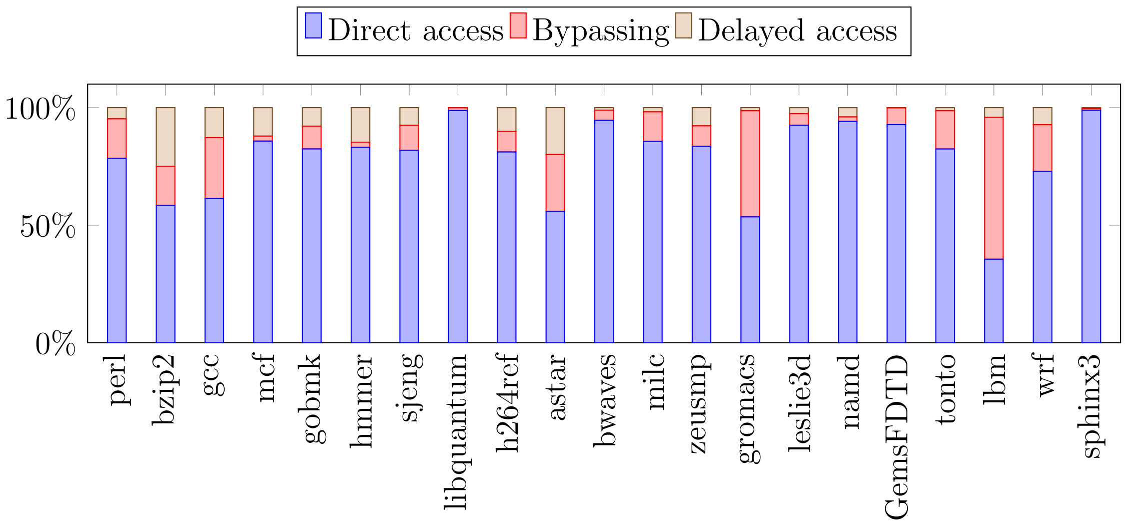


Figure 5.2: Load instruction distribution

values in **NoSQ**. The configuration of the evaluation is described in detail in Section 5.5. In the figure, **Direct access** means the load gets its value directly from the cache. **Bypassing** means the load gets its value through memory cloaking. **Delayed access** means the load cannot get its value from the cache until the conflicting store commits. Note that *bzip2*, *gcc*, *mcf*, *hmmer*, *h264ref* and *astar* have more than 10% **Delayed access** loads.

We also simulate the average execution time of **Bypassing** and **Delayed access** loads where the load execution time is defined to be the number of cycles spent between renaming of the load and the load result becoming available. In **Bypassing** cases, the execution time might be negative since the store data is available even before the load is renamed, on which case its execution time is set to zero. The comparison is illustrated in Figure 5.3.

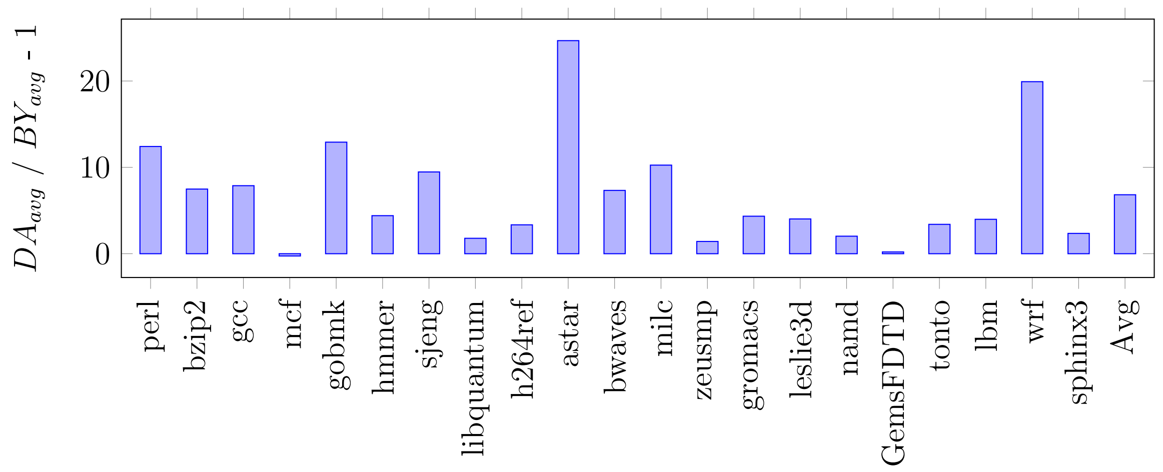


Figure 5.3: Delayed loads vs. bypassing loads

In this figure, if the ratio is greater than zero, **Delayed access** loads have longer average execution time. Otherwise, **Bypassing** loads are longer. The figure shows that **Delayed access** loads take significantly more cycles to execute in most benchmarks, except *mcf*. In *mcf*, the average execution time of **Delayed access** loads is 117.6 cycles and is 159.3 for **Bypassing** loads. This is because the colliding stores are always dependent on some other cache miss loads so that memory cloaking does not effectively work in these cases. Overall, the execution time of **Delayed access** loads is about 7 times longer than the **Bypassing** loads. If these **Delayed access** loads are on the critical path, the program will be forced to execute in an in-order manner.

5.3 The Concept of Memory Predication

Predication is a technique to convert control dependencies into data dependencies. It is widely applied in branch elimination where the branch result is computed as a predicate which can then be used to select the correct operand [106, 107]. When the store and load dependence is not consistent, **NoSQ** needs to conservatively wait until the data source becomes unambiguous or, on the other hand, risk frequent mispredictions. Clearly, this problem is analogous to branch prediction where a difficult to predict branch is encountered. **DMDP** therefore dynamically inserts a predicate to compare the store and load addresses. The comparison result can then be used to guide the load to obtain the correct operand from either the cache or the in-flight store in a manner similar to a conditional move instruction.

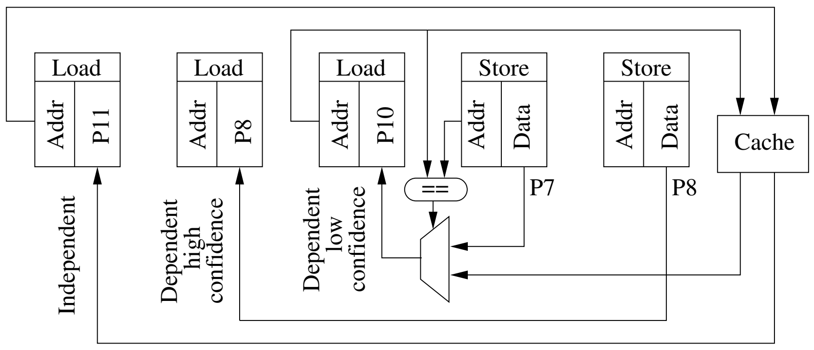


Figure 5.4: Three different ways to read data for loads.

Figure 5.4 shows how the load gets its data in three different ways. The first load does not find any dependent store so it gets its data from the cache. The second load has a colliding store and the predictor is confident. Therefore, memory cloaking is applied and the load reuses the same physical register (P8) as its own destination physical register. The third load also has a colliding store except its prediction is not confident. The store and the load addresses are compared to drive a multiplexer for selecting the correct data. Since the multiplexer's output is a new definition, the load is assigned a new physical register (P10) as usual.

Table 5.1
The difference between NoSQ and DMDP on different loads

	NoSQ	DMDP
No dependence or the store has committed	The data is read from the cache directly.	
Aliased store is predicted (high confidence)	The load reuses the physical register from the store. No cache read is necessary.	
Aliased store is predicted (low confidence)	The data is not read from the cache until the store is committed.	Predication is inserted so that both the store's data and the data from the cache are forwarded to the predicate instruction. The correct one is selected to bypass to the load's physical register.

Table 5.1 illustrates the difference between **NoSQ** and **DMDP**. The primary difference is how inconsistent store-load dependencies are handled. To be specific, the first row describes the situation when the load has never observed any dependence violation or the aliased store has committed and updated the cache when the load is renamed. **DMDP** converts all memory dependencies into register data dependencies so that loads do not need to check any store when it commits.

The use of the store data and address physical registers are not included in the original program semantics. These registers might have been released and re-allocated to other instructions before the predicated instruction is created. **DMDP** delays the release time of store registers until the store is committed and updates the cache so that any in-flight store can be utilized to create the predicate. For this purpose, **DMDP** assigns an extra physical register to each memory instruction to hold the computed address. This change simplifies the address comparison and the resulting microarchitecture as well, since the memory address can be directly read from the physical register file instead of doing a base register plus offset computation. The details of the implementation are described in the next section.

Note, when a load is predicted to be dependent on a store, there are two possible types of mispredictions. Either the load is independent of any in-flight store or the load is dependent on a different in-flight store. **DMDP** can only handle the former case, not the latter one. From the simulation, it is found that the first type of misprediction dominates the mispredictions. Therefore, applying predication can save most of the memory dependence mispredictions.

Figure 5.5 illustrates the memory dependence prediction results for low confidence loads. In this figure, **IndepStore** means the load is predicted to be dependent on a store but it is actually independent of any in-flight store; **DiffStore** means the load is dependent on a different in-flight store. **Correct** means the prediction is correct. It is apparent that **IndepStore** dominates every benchmark. In other words, if a load

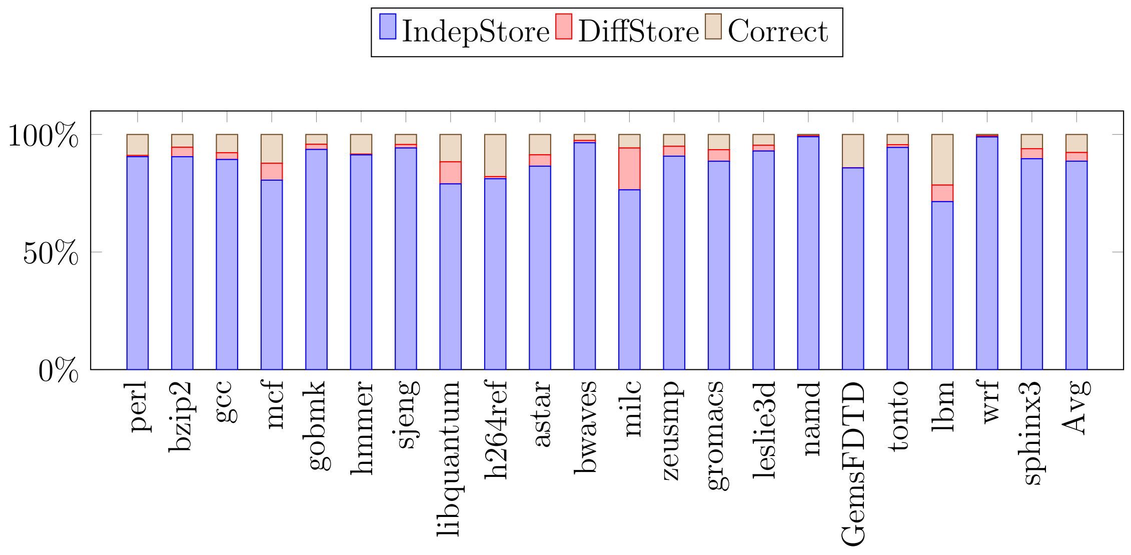


Figure 5.5: Memory dependence prediction results over low confidence loads

has a low confidence prediction, the load is most likely independent of any in-flight store. A naive solution would treat low confidence loads as independent loads and make them read directly from the cache. However, **DiffStore** and **Correct** would be mispredicted in this algorithm and the misprediction rate is considerable in some benchmarks such as *lbm*(28.6%) and *milc*(23.5%). The average misprediction rate is 11.4%. **DMDP** can correctly execute **IndepStore** and **Correct** which results in a misprediction rate of 3.7%. The delayed execution in **NoSQ** can also cover **IndepStore** and **Correct**. It can also cover some of **DiffStore** if the actual colliding store is older than the predicted store at the cost of drastically increased load latency.

5.4 Microarchitecture

The microarchitecture of **DMDP** is shown in Figure 5.6. It tracks each store using a unique *store sequence number* (SSN) [108] and three globally observable registers, SSN_{rename}, SSN_{retire} and SSN_{commit} are used to track the store instruction states.

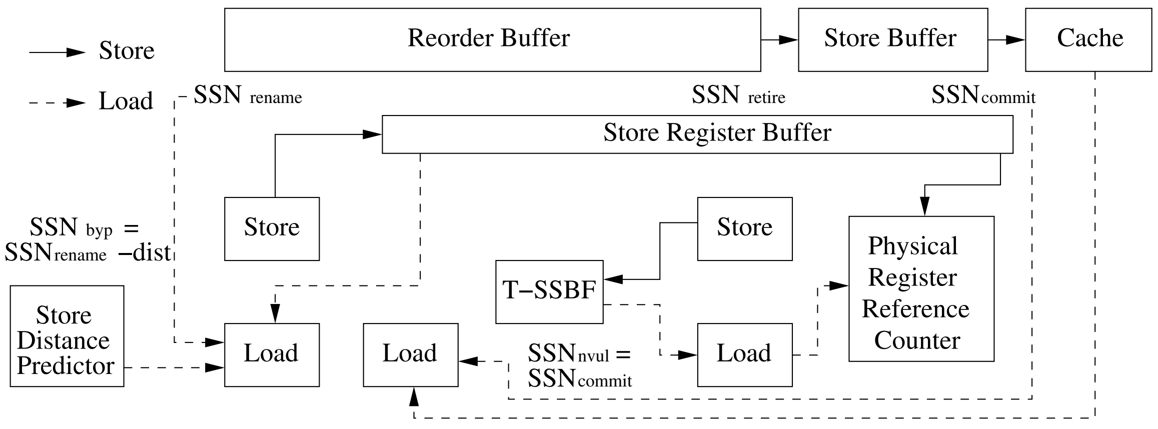


Figure 5.6: DMDP Microarchitecture

When a store is renamed, the SSN_{rename} is incremented and set as the store's SSN, hence a younger store has a larger SSN. When a store retires and commits, its SSN updates SSN_{retire} and SSN_{commit} respectively. The store buffer works like a queue to hold retired stores before they commit and loads never search the store buffer. *Store Register Buffer* holds the physical register numbers of every in-flight store instruction before they are committed. When a predicated MicroOp is created, the store's data and address physical register numbers are read from this buffer. In **DMDP**, store instructions have an extended physical register lifetime since the register might be read even after the store is retired. Therefore **DMDP** includes a *Physical Register Reference Counter* to manage the register release operations. For a load, its colliding store's SSN (SSN_{byp}) is predicted when the load is renamed. The relative store distance is predicted by the *Store Distance Predictor* and SSN_{byp} equals SSN_{rename} minus the store distance. When a load is executed and reads the data from the cache, the current SSN_{commit} is kept with the load as SSN_{nvul}. SSN_{nvul} indicates the youngest store to which the load is not vulnerable. At the retire stage, the speculative load needs to verify its value by re-execution. In order to minimize the number of re-executions, a *Tagged Store Sequence Bloom Filter* (T-SSBF) is added. In the rest of the section, we are going to elaborate on the microarchitecture details starting with the basic operations.

In order to minimize the overhead of speculative load verification, Store Vulnerability Window (SVW) [108] was adopted which only re-executes the load if necessary. The second part is a path-sensitive store distance predictor [17].

5.4.1 Store Vulnerability Window

When a load is speculative, its value has to be verified before the load is retired. A simple mechanism re-executes every load at the retire stage which doubles the bandwidth requirement for the cache. SVW substantially reduces the number of re-executions by only re-executing the load if the colliding store committed after the load was executed.

While a store commits, it writes the data to the cache and updates the SSN_{commit}. Therefore, any store whose SSN is smaller than or equal to SSN_{commit} has updated the cache. When a load reads the data from the cache, it also reads SSN_{commit} and keeps it as SSN_{nvul}. During the retire time, if the colliding store's SSN is greater than the load's SSN_{nvul}, that means the colliding store updated the cache after the load read from the cache. A potential RAW hazard is possible and the load needs to re-execute. If the re-execution provides a different value, a full penalty recovery is initiated. Otherwise, if the colliding store's SSN is smaller than or equal to the load's SSN_{nvul}, that means the colliding store has committed its change to the cache and the load read the correct value. Hence, no re-execution is required.

5.4.2 Tagged Store Sequence Bloom Filter

As it is described before, when a load is retired, the processor needs to identify its colliding store's SSN. Tagged Store Sequence Bloom Filter [109] (T-SSBF) is used to efficiently detect the store's SSN. T-SSBF is similar to an N-way set-associative cache indexed by the (hashed) memory address. The difference is each set in T-SSBF is organized like a FIFO, containing the last N store's SSNs which map to that set. When a store is retired (not committed), it writes its SSN into the T-SSBF (T-SSBF[st.addr] = st.SSN). When a load is retired, it reads the T-SSBF to find its colliding store's SSN. If multiple instances with the same address are found, the largest SSN (the youngest) is selected. On the other hand, if no matching address is found, the smallest SSN in the same set is selected.

In order to detect collisions on partial-word accesses, each memory access maintains a word address and a Byte Access Bits (BAB). BAB is used to indicate which bytes in that word are accessed. BAB is also written into the T-SSBF along with the SSN. When the word address matches and $store_{BAB}$ & $load_{BAB}$ is greater than zero, this load-store pair collides.

5.4.3 Load Re-execution

When the colliding store's SSN is larger than the load's SSN_{nvul}, a load re-execution is scheduled. Since the store buffer still holds some pending stores which have not yet updated the cache, the load re-execution is not issued until the store buffer is drained. This impact caused by the store buffer can significantly deteriorate the overall performance. Optimization in reducing the number of load re-executions is desirable and we will describe it later.

5.4.4 Memory Dependence Prediction

In **DMDP**, *Store Distance Predictor* [110] is used to predict the memory dependence. This structure is indexed using the load's PC and predicts how many stores there are between the aliased store and the load, assuming this distance is constant so the same load will always collide with the store at the same distance. With the predicted store distance, the colliding store's SSN is calculated as $ld.SSN_{byp} = SSN_{rename}$ - $ld.dist_{byp}$. At the retire stage, the store distance prediction needs to be verified. The actual store distance is computed as SSN_{retire} - T-SSBF[ld.addr]. If the prediction is wrong, the store distance predictor is updated with the actual distance.

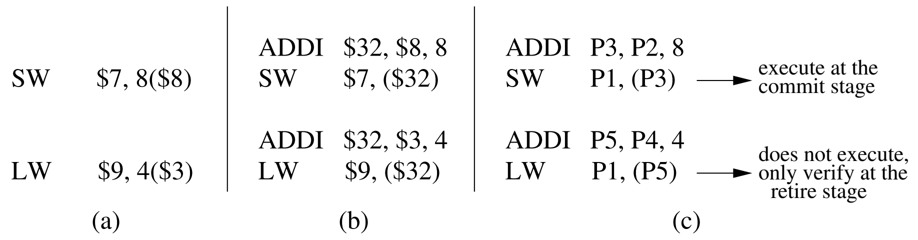
Multiple branches between the store and the dependent load may cause the store distance to vary if the branches lead to different paths. A path-sensitive memory dependence predictor [17] is used to handle the change in control flow which uses an identical structure except indexing its table by an XOR of load's PC and branch history bits. The path-sensitive predictor and the path-insensitive predictor are read simultaneously. Prediction from the path-sensitive predictor is selected if it is available. Otherwise, the path-insensitive prediction is used. If the load misses both predictors, the load is predicted to be independent and can directly read the cache when its address is available.

5.4.5 Memory Cloaking

Once a colliding store's SSN is predicted, the physical register which produces the store data is read from the *Store Register Buffer*. The load uses this physical register as its destination register. As a result, this load does not need to access the cache and only computes its address. In this approach, the DEF-store-load-USE dependence chain now is collapsed to DEF-USE and this is called memory cloaking [12]. Memory Cloaking is a very powerful method, because the data is forwarded to the load even without knowing the address.

DMDP splits memory operations at the decode stage into two MicroOps: an address computation and a memory access. In this design, the store queue and the load queue are completely eliminated. Figure 5.7 illustrates the MicroOps creation and the renaming procedure. Figure 5.7(a) shows the original assembly code of the store and the load. In Figure 5.7(b), an address generation MicroOp, ADDI, is added before each memory access, which eliminates the offset field in the memory MicroOps. Note that the logical destination register of the ADDI is \$32, which is not defined in MIPS ISA (\$0-\$31). This register is only visible in the hardware and facilitates physical register renaming and release in the same way as a normal superscalar processor.

Figure 5.7(c) shows the renamed MicroOps. The store does not write to the cache



```
(a)
SW      $7, 8($8)

LW      $9, 4($3)

(b)
ADDI    $32, $8, 8
SW      $7, ($32)

ADDI    $32, $3, 4
LW      $9, ($32)

(c)
ADDI    P3, P2, 8
SW      P1, (P3)

ADDI    P5, P4, 4
LW      P1, (P5)
```

execute at the commit stage
does not execute, only verify at the retire stage
(a)
(b)
(c)

Figure 5.7: Memory Cloaking

until it is committed, hence it is not dispatched to the out-of-order core. Both the data physical register identity (P1) and the address physical register identity (P3) are kept in the *Store Register Buffer* so that when this store is committed it can read these two values from the register file and update the cache. The store queue is removed at the cost of an additional address physical register (in a typical superscalar processor, the address does not require a dedicated physical register but instead an entry in the store queue).

Load instructions operate in a similar manner such that a dedicated address physical register is allocated. At the retire stage, both the data and the address physical registers are read to verify the memory dependence prediction (the data is required if a load re-execution verification is issued, and the address is required to read T-SSBF). The load queue is removed since the address is kept in the register file. Figure 5.7(c) shows a bypassing load which reuses the store data register (P1) as its own destination physical register. Hence, this load is not dispatched to the out-of-order core either. The processor only verifies its value at the retire stage.

DMDP is different from **NoSQ** in the memory address computation. In **NoSQ**, the address offset is kept in the ROB and the address is calculated at the retire stage (for non-bypassing loads, these are extra computations). In **DMDP**, address computation is finished in the out-of-order core and a register file read is required at the retire/commit stage in order to eliminate the store queue and the load queue. Furthermore, the address generation instruction (AGI) translates the virtual address to a physical address by setting a special MicroOp flag, making it different from a normal ADDI. When the AGI is computed, it searches the TLB to find the physical address and stores the physical address in the address register. Therefore, at the retire/commit stage, physical addresses are available for memory ordering violation detection and no extra translation is needed. The drawback of this approach is the non-bypassing loads have to wait for the address translation and an extra delay is imposed. In order to remove this side effect, the virtual address is used to read the data array and the tag array simultaneously with the address translation. In the next cycle, the translated physical address is compared with the tag array output and the correct data is selected. This approach takes advantage of a VIPT cache organization.

5.4.6 Predication Insertion

A predicate is inserted at the load if the memory dependence prediction is not confident. Figure 5.8 shows an example of predicate creation and renaming. Figure 5.8(a) shows the original assembly code of the store and the load. Figure 5.8(b) presents the decoded MicroOps alongside with address generation MicroOps. Figure 5.8(c) illustrates the predicate creation before renaming. Note that, in reality the predication insertion uses physical registers. the logical registers in Figure 5.8(c) are used to improve readability.

```
                 | ADDI  $32, $8, 8 | ADDI  $32, $8, 8    | ADDI  P3, P2, 8
SW   $7, 8($8)   | SW    $7, ($32)  | SW    $7, ($32)     | SW    P1, (P3)
                 |                  |                     |
                 | ADDI  $32, $3, 4 | ADDI  $32, $3, 4    | ADDI  P5, P4, 4
LW   $9, 4($3)   | LW    $9, ($32)  | LW    $33, ($32)    | LW    P6, (P5)
                 |                  | CMP   $34, $32, ?   | CMP   P7, P5, P3
                 |                  | CMOV  $9, $34, ?    | CMOV  P8, P7, P1
                 |                  | CMOV  $9, !$34, $33 | CMOV  P8, !P7, P6

      (a)                (b)                 (c)                  (d)
```

Figure 5.8: Memory predication insertion

In the figure, there are three new MicroOps inserted after the load: a comparison, CMP, which produces the predicate \$34 and two conditional moves, one of which would update the load destination register \$9. Logical register \$33 keeps the data read from the cache and \$34 serves as the predicate. The store address and the store data sections are marked with a question mark in the figure. Because logical register \$7 and \$32 may be modified after the store, only physical registers are available during the predication insertion process. The CMP instruction compares the store address with the load address and sets the predicate \$34 to one if the addresses match. If the predicate is set, the CMOV instruction forwards the store data to \$9. Otherwise, the loaded data \$33 is forwarded.

Figure 5.8(d) shows the renamed predication code. Note that the two CMOVs are mapped to the same physical register P8 since only one of them will write to the RF. Both CMOVs are dispatched to the out-of-order core. When the CMOV is woken up to execute, it first checks the predicate and only executes if the predicate is set. Otherwise, the CMOV is treated as a NOP and does not update the RF. The benefit of sharing one physical register is it reduces the data dependence and the number of the operands is two instead of three (a predicate and two operands selected by the predicate). The physical register P8 is defined twice, similar to a memory cloaking destination register. Its release algorithm is the same and is elaborated on the details next.

5.4.7 Physical Register Reference Counter

In superscalar processors, a physical register is defined once in its lifetime and is never read again after it is released. Since neither of these conditions are valid in **DMDP**, the mechanism, *Physical Register Reference Counters*, is incorporated to guide the physical register allocation and release policy.

In **DMDP**, a given physical register might be defined multiple times in its lifetime. For example, a load might reuse the colliding store's data register as its own destination register in memory cloaking. A predication insertion creates two conditional moves which have the same destination register. A counter based algorithm [111] is implemented in **DMDP** to track the number of definitions through the register's lifetime. This producer counter is incremented when the register is defined and decremented when the register is virtually released. Figure 5.9 demonstrates a simple example. P7 is defined twice so the counter number is two at the beginning. When the second instruction is retired, it virtually releases the previous definition, P7. Hence, the counter value is decremented to one. When the last instruction is retired, it also decrements the producer count of P7 and the register is released as the counter is zero.

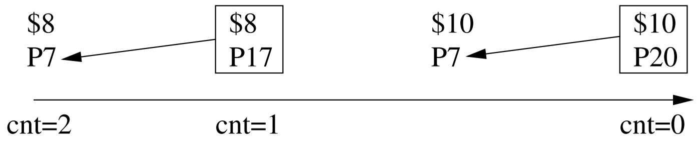


Figure 5.9: The producer counter

On the other hand, a physical register might be read even after it is released. For example, in Figure 5.8(d), P1 contains the store data and could be released before it is read by the conditional move. Another case happens in the write buffer. When a store is retired and transferred to the store buffer, its data and address physical registers might be released before the store is committed. We have to extend the lifetime of these physical registers in order to guarantee correct memory forwarding. A consumer counter is added for each physical register. When a consumer operand is renamed, the renamed physical register's counter is incremented. When the instruction which has that operand executes, the counter is decremented. The store executes when it is committed. The consumer counter was first proposed in [112] which is used to make early physical register release. But in our scenario, it is used to delay the register release.

Overall, when a physical register has a producer counter of zero and a consumer counter of zero, it is released. During a misprediction recovery, the walking mechanism described in [18] is implemented to restore the counters.

5.4.8 Load Re-execution Filter

When a speculative load is about to retire, its loaded data needs to be verified by a re-execution. The cost of load re-execution is too high as loads can not be issued until the write buffer is drained. As a result, a filter algorithm is desirable for minimizing the number of load re-executions. All of the loads are classified into two categories: i) the loads which read their data from the cache; ii) the loads which get their data from an in-flight store.

Table 5.2
Load re-execution policy for different loads.

	re-execution condition
Data from a cache	T-SSBF[ld.addr] $> ld.SSN_{nvul}$
Data from a store	T-SSBF[ld.addr] $!= ld.SSN_{byp}$

Table 5.2 lists the re-execution policy for these two kinds of loads. If the data is coming from the cache, $ld.SSN_{nvul}$ indicates the SSN_{commit} when the load is executed. Therefore, if the actual colliding store's SSN is larger than $ld.SSN_{nvul}$, a re-execution

verification is required. If the data is forwarded by a predicted colliding store, then the actual colliding store's SSN must match with the predicted one. Otherwise, a re-execution verification is issued.

5.4.9 Silent Store Effect

Silent stores [113] can be detected in many programs, in which multiple stores write the same value into a particular memory location. This is mainly caused by the program redundancy. In **DMDP**, silent stores impose a lot of unnecessary load re-executions. Figure 5.10 shows a simple example, in which the two stores write the same value to the same address. The load also reads this address. This figure displays the status when the load is executed. Since store1 has committed (st1.SSN $< SSN_{commit}$), the load reads the data updated by store1 from the cache. When the load is about to retire, it finds itself colliding with store2 (st2.SSN $> ld.SSN_{nvul}$). Thus, a load re-execution is issued and no exception is set since the reloaded data is the same. The original **NoSQ** design does not update the *Store Distance Predictor* unless an exception is reached. Consequently, this load will incur re-execution many times without creating any exception.

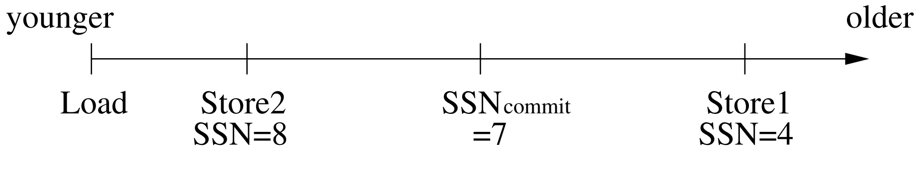


Figure 5.10: Load re-execution incurred by silent store

In order to solve this problem, the memory dependence should be created even when no exception is observed. Whenever a load re-execution occurs, the *Store Distance Predictor* is updated. In Figure 5.10, the store distance between store2 and the load is kept in the predictor. Thus, store2 will forward the data to the load and no load re-execution is required based on the policy in Table 5.2.

5.4.10 Partial-Word Forwarding

For memory predication to work, it must handle any kind of store-load forwarding, including partial-word forwarding. **DMDP** uses word address plus byte access bits (BAB) to detect partial-word forwarding violation. On a 32-bit machine, a word is composed of 4 bytes so 4 bits (BAB) are used to indicate which bytes are accessed. This BAB can be expanded to 8 bits for a 64-bit machine. When a store retires, its BAB, $store_{BAB}$ is written into the T-SSBF with other information. When a load retires, not only the word address but also the BAB is compared. Figure 5.11 shows how a partial-word forwarding is verified.

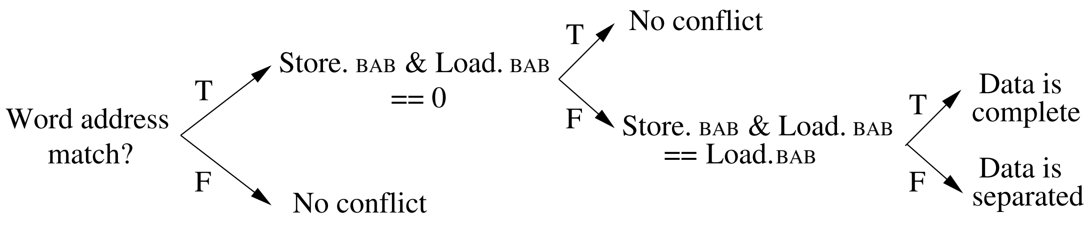


Figure 5.11: The decision tree of partial-word forwarding detection

If $store_{BAB}$ & $load_{BAB}$ is equivalent to $load_{BAB}$, that means the prior store contains the whole data section required by the load. The data forwarding is correct. On the other hand, if the store only modifies part of the data section, which means the needed data is separated by multiple stores, then a load re-execution is triggered when the load is retired.

A partial-word forwarding may cause the forwarded data to be shifted. For example, a store writes a whole word into the cache and the dependent load only reads the upper half word. In which case, the store data is right shifted 16 bits before the forwarding. In MIPS, the shift amount is decided by the least significant two bits of the memory address. The value of these two bits times eight is the shift amount. The store shifts left and the load shifts right. **DMDP** has a dedicated MicroOp, CMP, to compare the store and load address. When a CMP is executed, **DMDP** also puts the shift amount and direction information into the predicate (the predicate is a word-wide register, only one bit is used to guard the predicated instruction, other bits can be used). Therefore, the CMOV knows how to shift the operand before it is forwarded.

Other than address alignment, the forwarded data may be masked and sign/zero extended based on the load type. **DMDP** keeps the load type in the CMOV when it is created. So when CMOV is executed, the operand is trimmed properly. Moreover, partial-word loads, such as half word load or byte load, are prohibited from doing memory cloaking due to the alignment or sign/zero extension. Thus, these loads are forced to use predication for store-load communications. **NoSQ** inserts special "shift & mask instruction" for partial-word communication. However, the store and the load addresses are unknown at the rename stage, thus the shift amount has to be predicted. This partial-word mechanism can be easily adopted to other ISAs, since the store address, load address, store type and load type can be integrated into the predicate.

5.4.11 Confidence Predictor

When a load is predicted to be dependent on a store, **DMDP** consults the confidence predictor to decide whether memory cloaking or memory predication should be used. The confidence predictor is embedded in the *Store Distance Predictor* and updated at the retire stage. The loads which cause load re-executions or are predicted to be dependent on a store can update the confidence predictor. When the prediction is correct, the corresponding confidence counter is increased. Otherwise, the confidence counter is decreased. A common confidence predictor modifies the counter with a balanced strategy. In other words, it increases and decreases the counter by the same amount, for example, one. In **DMDP**, pushing instructions to predication only causes a trivial data dependence increase, but a dependence misprediction results in a full recovery penalty. Because the cost is biased, the confidence counter update should be biased as well. **DMDP** right shifts the counter by one bit (divided by two) whenever a misprediction is detected and only increments the counter by one in other cases. By applying this strategy, fewer mispredictions are experienced at the cost of more predications.

5.4.12 Memory Consistency

In a multi-core system, the cache lines might be invalidated by other cores. Therefore, even if the load satisfies the local memory dependence check, it may still need to be re-executed due to the stores from other cores [109]. This makes two changes to the design: i) When a cache line is invalidated by another core, all the words in that cache line should update the T-SSBF; ii) The word invalidated by another core should write $SSN_{commit}+1$ as its SSN in the T-SSBF. In other words, all of the in-flight loads which were executed before the invalidation should re-execute if their addresses match.

Different memory consistency models were considered in the design of **DMDP**, such as total store order (TSO) and relaxed memory order (RMO). In TSO, the stores in the store buffer are committed following the program order. When the store buffer is full, stores are not allowed to retire from the ROB. RMO mitigates the pressure of the store buffer and permits the stores to commit in any order. In either model, the load in **NoSQ** has to wait for the colliding store and its preceding stores to commit if the memory dependence prediction is not confident. **DMDP** is not constrained by the committing of the stores.

5.5 Evaluation Methodology

This work was simulated by using MIPS-I ISA without delayed branching. This ISA is very similar to PISA ISA, used by SimpleScalar [55]. GCC 4.9.2 tailored to this ISA was used to compile the benchmarks and generate binary code with the highest optimization ("-O3") set. Spec 2006 was selected as the benchmark suite. All simulation models were designed with Architectural Description Language (ADL) [46]. The ADL compiler can automatically generate the assembler, the disassembler and a cycle-accurate simulator which respects timing at the register transfer level from the description of the microarchitecture and its ISA specified in ADL language.

In order to efficiently simulate these mechanisms, Simpoint 3.2 [56, 57] was incorporated to minimize the simulation time. For each benchmark, a set of checkpoint images were generated and each checkpoint image contained the complete memory data segment, the register file and the program counter (PC). Other architecture related structures were not included, such as cache, branch predictor, memory dependence predictor, etc. Hence, the simulation of each interval had a cold start. In order to compensate for this effect, a large size, 100 million retired instructions, was selected to simulate each interval. Since each interval simulation is independent of others, all of the intervals were simultaneously simulated to further shorten the total simulation time. Currently, the file descriptors are not kept in the checkpoint. Thus, there is no way to simulate an interval if it had file operations and the file descriptor was created before the checkpoint. When this happened, that checkpoint interval was replaced with the dominant checkpoint in that benchmark. In *h264ref*, one checkpoint was substituted (0.92% weight) for the dominant checkpoint (18.14% weight) and in *hmmer*, two checkpoints (0.22%, 0.43% weight) were substituted for the dominant checkpoint (98.9% weight). As the replaced intervals have very limited weights (<1.0 %), the impact of this substitution is minimal. McPAT 1.4 [50] was also modified to evaluate the dynamic energy consumption. T-SSBF and the memory dependence predictor which replace the load queue and the store queue were both modeled. DRAMSim2 [58] was also embedded to evaluate the memory subsystem behavior.

The benchmarks simulated in this work are:

Integer: *perl*, *bzip*, *gcc*, *mcf*, *gobmk*, *hmmer*, *sjeng*, *libquantum*, *h264ref*, *astar*.

Float: *bwaves*, *milc*, *zeusmp*, *gromacs*, *leslie3d*, *namd*, *GemsFDTD*, *tonto*, *lbm*, *wrf*, *sphinx3*.

These benchmarks were simulated with the "ref" input. The remaining missing benchmarks were not included due to the linker's inability to link them. The processor configuration of the baseline architecture is listed in Table 5.3 which is used as a reference for comparing with other models. The baseline architecture has a store queue and a load queue with unlimited entries. A store buffer is also implemented obeying TSO model. Store coalescing was implemented to alleviate the write port pressure. Since TSO is considered, only consecutive stores are coalesced.

Table 5.3
Baseline Processor Configuration

ROB / RS / PRF	256 / 64 / 320
Fetch / Decode / Issue	8 / 8 / 8
Store Queue	unlimited entries, 4 cycles search latency
Store Buffer	16 entries, store coalesce
Memory Dependence Prediction	Store Sets [16]
Cache	32KB 8-way set associative iL1; 32KB 8-way set associative dL1, 4 cycles hit latency, 2 read ports, 1 write port; 512KB 8-way set associative L2, 10 cycles hit latency;
Memory	16GB DDR3L-1600, 2 channels, 2 ranks, 8 banks, open page, up to 64 pending requests [59]
Recovery Penalty	minimum 15 cycles
Int ALU / Int Mul	1 cycle / 3 cycles
Int Div, FP ALU	7 cycles
Branch Predictor	8 kB TAGE [14]
Tech node	22nm
Clock frequency	3.2GHz

A similar configuration from the baseline was adopted to the following models except their store-load communication mechanism.

1. NoSQ: This architecture has no store and load queue which are replaced with a 4-way set associative, 128 entry T-SSBF. Each entry contains a 20 bit SSN, a 4 bit BAB and a 25 bit tag. The total size of T-SSBF is 6.125 Kbits. The *Store Distance Predictor* uses two 4-way associative, 1K entry tables. One of the tables is for path-insensitive predictions and the other is for path-sensitive predictions. The path-insensitive table is indexed by the load PC. The path-sensitive table is indexed by the XOR of the load PC and an 8 bit branch history. Each entry of the tables contains a 7 bit confidence counter, a 22 bit tag and a 6 bit distance part. The predictor's total size is 8.75KB. The confidence counter is set to 64 by default. If the value is greater than 63, memory cloaking is used, or the load has to wait for the colliding store to commit. The number of delayed loads which can be kept in the core is unlimited. Silent-store-aware predictor update policy is used to match with **DMDP**.

2. DMDP: DMDP has the same T-SSBF and dependence predictor as **NoSQ**. The only difference is that **NoSQ** will decrease the confidence counter by one if a misprediction is detected. But **DMDP** will divide the counter by two in the same cases. Predicate is added when a low confidence prediction is made.

3. Perfect: This model has a perfect memory dependence predictor so that every load gets its data from either a colliding store or the cache. No delayed executions or mispredictions are experienced.

5.6 Experimental Results

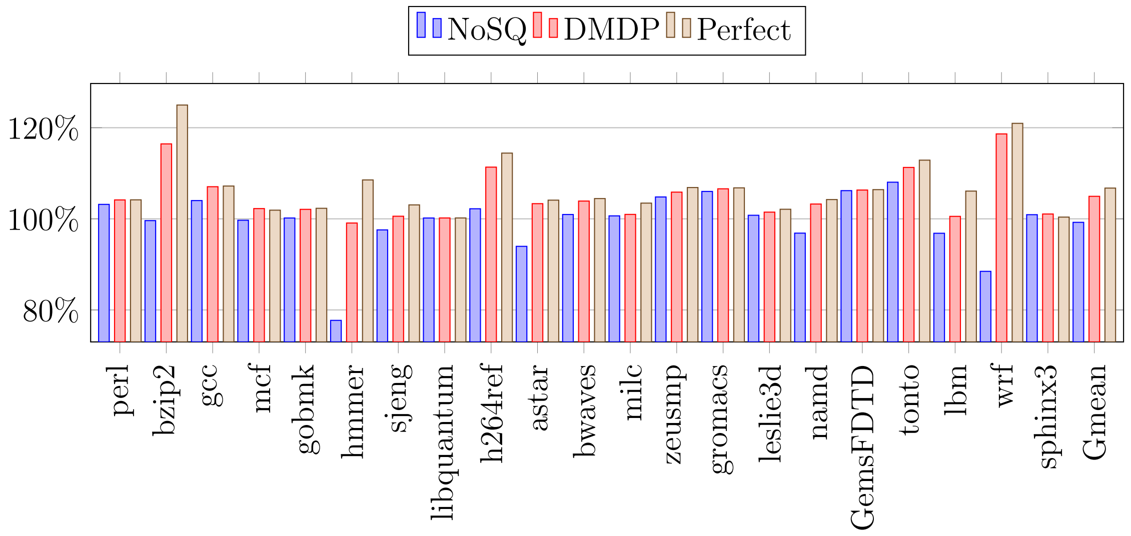


Figure 5.12: Spec 2006 Speedup over the baseline

Figure 5.12 illustrates the speed-up of **NoSQ**, **DMDP** and **Perfect** models over the baseline superscalar model. The geometric means of the Integer benchmarks are 97.5%, 104.5% and 106.8%, for **NoSQ**, **DMDP** and **Perfect** respectively, and the corresponding floating point benchmark performances are 100.8%, 105.3% and 106.6%. Clearly, **DMDP** is much closer to **Perfect** in terms of IPC performance.

5.6.1 NoSQ VS. Baseline

NoSQ can forward store data to the load by memory cloaking. That is the reason why it outperforms the baseline in some of the benchmarks. On the other hand,

NoSQ has to stall the retire stage when a load re-execution is issued when the store buffer is not drained. Moreover, **NoSQ** may experience more memory dependence mispredictions due to its aggressive prediction strategy. Figure 5.12 shows **NoSQ** works more than 20% worse in *hmmer*. Analyzing the result, we found that **NoSQ** has a significant amount of memory dependence mispredictions (3.06 Mispredictions Per 1k Instructions). Further analysis showed that the silent store predictor update policy, which was described before, had a substantial impact in this benchmark. This policy would update the **Store Distance Predictor** whenever a load re-execution was triggered. If the original mechanism was used, which only updates the predictor if the re-execution leads to an exception (the reloaded data is different), **NoSQ** has fewer mispredictions and higher performance in *hmmer*. However, it reduces the performance of other benchmarks.

The silent-store-aware predictor update policy is a double-edged sword. It reduces the number of load re-executions immensely but also might cause the increase of mispredictions. **DMDP** could compensate this shortcoming by using predication. From our evaluation, **DMDP** had much fewer mispredictions (1.03MPKI VS. 3.06MPKI) and it only had 1% lower performance than the baseline in *hmmer*.

5.6.2 DMDP VS. Baseline

DMDP surpasses the baseline in every benchmark except *hmmer* which is caused by the nontrivial memory dependence mispredictions as mentioned before. In the baseline model, loads have to read the data from the cache, store queue or store buffer. All of these structures have a constant access latency (4 cycles in the simulation). **DMDP** can use memory cloaking to bridge stores with loads if the predictions are confident. Even a low confidence load can obtain its data quicker if the data is from an in-flight store.

Table 5.4
Average execution time of all loads

	baseline (Cycles)	DMDP (Cycles)		baseline (Cycles)	DMDP (Cycles)
perl	15.86	12.45	bzip2	36.67	19.48
gcc	44.98	35.04	mcf	112.44	104.00
gobmk	13.51	11.52	hmmer	11.20	7.47
sjeng	12.60	10.62	libquantum	125.23	124.73
h264ref	22.68	17.32	astar	21.18	13.77
bwaves	42.56	36.76	milc	73.40	61.18
zeusmp	26.97	21.21	gromacs	32.13	11.41
leslie3d	36.55	32.91	namd	20.22	18.94
GemsFDTD	14.78	11.62	tonto	20.31	12.89
lbm	72.17	31.15	wrf	18.17	9.19
sphinx3	51.95	50.47	Average	39.31	31.15

Table 5.4 lists the average execution time of the loads in the baseline and **DMDP**. **DMDP** has a shorter execution time in every single benchmark and on average, saves more than 20% of the execution time for loads. Figure 5.12 shows that **DMDP** has

the most improvements in *wrf* and *bzip2* and in these two benchmarks, **DMDP** saves about half of the execution time of the loads.

5.6.3 DMDP VS. NoSQ

DMDP outperforms **NoSQ** in every single benchmark. The geometric mean of the speed-up is 7.17% (Int) and 4.48% (FP). If a low confidence load is renamed, **NoSQ** has to wait until the predicted colliding store commits. **DMDP** can steer the load to find its correct data by predication, disregarding the store commit states. Table 5.5 lists the average execution time of the low confidence loads in **NoSQ** and **DMDP**. **DMDP** saves up to 79.25% execution time and the average is 54.48%. *Libquantum* is the only benchmark in which **DMDP** has a longer execution time. This data is not representative due to the fact *libquantum* has so few low confidence loads.

In Figure 5.12, **DMDP** surpasses **NoSQ** in *wrf* by 34.1%, which is the highest improvement. **NoSQ** works worse than not only **DMDP** but also the baseline. From our evaluation, the average execution time of all loads is 18.17 cycles, 13.85 cycles and 9.19 cycles for the baseline, **NoSQ** and **DMDP** respectively. **NoSQ** has a shorter execution time, but still works worse than the baseline. One possible reason is the delayed loads in **NoSQ** are on the critical path of the program and slow down the rest of the instructions. Therefore, even the average load execution time is saved,

Table 5.5
Average execution time of low confidence loads

	NoSQ (Cycles)	DMDP (Cycles)		NoSQ (Cycles)	DMDP (Cycles)
perl	63.79	14.54	bzip2	65.29	22.34
gcc	60.69	18.14	mcf	111.40	52.39
gobmk	23.73	11.58	hmmer	37.19	8.91
sjeng	24.54	12.36	libquantum	9.11	13.38
h264ref	41.29	19.17	astar	85.47	25.16
bwaves	103.90	36.17	milc	141.28	85.72
zeusmp	118.72	24.66	gromacs	68.10	65.94
leslie3d	53.76	20.59	namd	22.53	16.33
GemsFDTD	11.11	9.98	tonto	35.85	10.75
lbm	129.69	100.76	wrf	61.59	12.78
sphinx3	49.93	18.68	Average	62.81	28.59

the whole program runs slower. The average execution time of all instructions is 19.53 cycles, 21.47 cycles and 12.74 cycles for the baseline, **NoSQ** and **DMDP** respectively.

5.6.4 DMDP VS. Perfect

On geometric mean, **DMDP** loses 2.19% (Int) and 1.25% (FP) IPC compared to **Perfect**. There are three reasons why **Perfect** outperforms **DMDP**: i) **DMDP** has a large amount of memory dependence mispredictions in some benchmarks which **Perfect** does not have; ii) When a load verification triggers a load re-execution, **DMDP** has to wait for the store buffer to be drained whereas **Perfect** never re-executes any loads; iii) For low confidence loads, **DMDP** still needs to wait for the addresses being computed even if the addresses match. **Perfect** only has high confidence loads which receive their data by memory cloaking.

Table 5.6
Memory dependence misprediction rate

	NoSQ (MPKI)	DMDP (MPKI)		NoSQ (MPKI)	DMDP (MPKI)
perl	0.144	0.141	bzip2	0.784	1.409
gcc	0.301	0.250	mcf	0.232	0.147
gobmk	0.305	0.198	hmmer	3.061	1.029
sjeng	0.420	0.357	libquantum	0.000	0.000
h264ref	0.226	0.118	astar	0.110	0.086
bwaves	0.039	0.002	milc	0.363	0.363
zeusmp	0.021	0.024	gromacs	0.070	0.057
leslie3d	0.063	0.056	namd	0.004	0.003
GemsFDTD	0.007	0.001	tonto	0.160	0.130
lbm	0.101	0.089	wrf	0.057	0.066
sphinx3	0.020	0.046	Average	0.309	0.218

Table 5.6 lists the memory dependence misprediction rate in **NoSQ** and **DMDP** measured in terms of Mispredictions Per 1k Instructions (MPKI). On one hand, **DMDP** would have more low confidence loads since the confidence predictor has a biased update policy (Section 5.4.11). As a result, **DMDP** should have fewer mispredictions. On the other hand, **NoSQ** would delay the execution of low confidence loads. As a result, some mispredictions which can not be covered by **DMDP** can be covered by **NoSQ**. For example, if the load depends on a different in-flight store and that store is older than the predicted one, **NoSQ** can read the correct data through the cache. In the same case, **DMDP** might read the cache earlier but potentially the wrong value.

When a load is re-executed, the retire stage is stalled until the store buffer drains. Table 5.7 lists the number of stalled cycles per 1000 committed instructions in **NoSQ**

Table 5.7
Load re-execution related stalls per 1k committed instructions

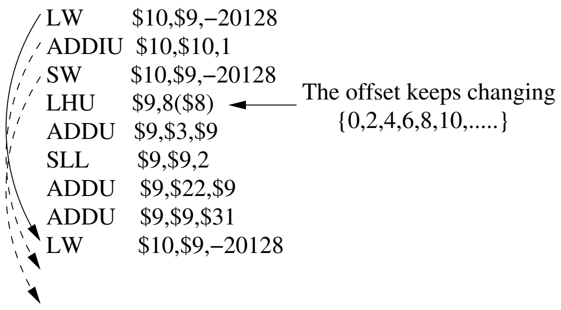
	NoSQ (cycles)	DMDP (cycles)		NoSQ (cycles)	DMDP (cycles)
perl	4.532	11.258	bzip2	3.999	30.235
gcc	5.031	12.182	mcf	4.205	9.173
gobmk	0.679	0.951	hmmer	32.675	101.631
sjeng	1.118	1.485	lib	0.001	0.001
h264ref	1.002	15.038	astar	0.881	54.548
bwaves	0.720	6.231	milc	16.997	32.113
zeusmp	5.629	18.389	gromacs	0.262	0.526
leslie3d	2.430	3.925	namd	0.040	0.338
Gems	0.004	0.028	tonto	0.419	0.882
lbm	154.956	155.260	wrf	0.575	5.027
sphinx3	0.161	0.200			

and **DMDP**. **DMDP** has more stalled cycles in every benchmark due to its early load execution. Since **NoSQ** delays the low confidence load execution, it has a much narrower vulnerable window and fewer load re-executions are issued.

DMDP loses the most performance in these three benchmarks: *hmmer* (91.29%), *bzip2* (93.17%) and *lbm* (94.77%) when compared with **Perfect**. The first two benchmarks have the most memory dependence mispredictions (Table 5.6) and *lbm* has the most re-execution related stalls (Table 5.7). These are the two major obstructions impeding **DMDP** to reach a higher performance.

5.6.5 Case Study in *bzip2*

Note that **DMDP** has more dependence mispredictions than **NoSQ** in *bzip2*. A snapshot of the code which causes the most mispredictions is demonstrated in Figure 5.13. In this figure, **LHU** sequentially reads an array which contains a pointer ($9). After a series of computation, the pointed value is incremented by one. If the array has two identical values, they point to the same memory location and the increment operations collide with each other. During the execution, the distance between the colliding store and the load kept changing. Therefore, a lot of mispredictions were observed in *bzip2*.



```
LW      $10,$9,–20128
ADDIU   $10,$10,1
SW      $10,$9,–20128
LHU     $9,8($8)
ADDU    $9,$3,$9
SLL     $9,$9,2
ADDU    $9,$22,$9
ADDU    $9,$9,$31
LW      $10,$9,–20128
```

Figure 5.13: A *bzip2* code snapshot

Let us assume the colliding store is randomly distributed. Thus, when a misprediction happens, half of the time the actual colliding store is older than the predicted colliding store and the other half is younger. **NoSQ** can cover the former cases and **DMDP** mispredicts both cases. Table 5.6 shows **NoSQ** has about half of the mispredictions of **DMDP**.

5.6.6 Store Buffer Size Effect

The loads in **DMDP** and **NoSQ** do not associatively search the store buffer which simplifies the design significantly. As a result, a much larger store buffer is implemented with the same cost and hide more cache misses imposed by stores. The store buffer size has a substantial impact on performance, especially for a multiprocessor [105, 114]. Even **DMDP** is designed and evaluated for single-thread processors, it can easily be adopted to a multiprocessor and boost the performance for multi-threaded workloads.

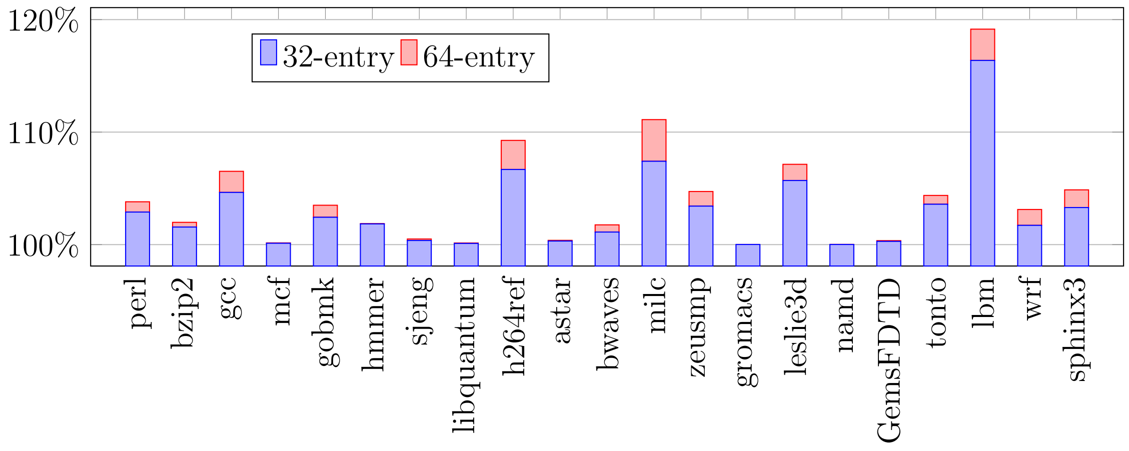


Figure 5.14: 32,64-entry SB VS. 16-entry SB

Figure 5.14 depicts the speed-up of **DMDP** with a 32-entry store buffer and a 64-entry store buffer over another one with a 16-entry store buffer. Using a geometric mean, the 32-entry model outperforms the 16-entry model by 2.07% (Int) and 3.81%(FP), the 64-entry model outperforms the 16-entry model by 2.77% (Int) and

5.01% (FP). Through all the benchmarks, *lbm* has the highest performance improvement with a larger store buffer. Moreover, the number of stalled cycles incurred by a full store buffer is 503.1 cycles, 220.5 cycles and 75.0 cycles per 1000 committed instructions for a 16-entry, 32-entry and 64-entry store buffer accordingly. Apparently, a larger store buffer can immensely help for single-thread performance and it is more fruitful in multi-threaded workloads.

5.6.7 Alternative Configurations

A 4-issue width configuration was also simulated. The IPC improvement over **NoSQ** shrunk to 4.56% (Int) and 2.41% (FP). This is because with a smaller issue width, the vulnerable window gets narrower and the in-flight store-load communication reduces. The number of the low confidence loads drops as well (23.4% is removed). As a result, it is less likely for **DMDP** to save delayed load executions.

A 512-entry ROB is also simulated, which yields more IPC improvement (7.56% Int, 6.35%). A larger ROB would help **DMDP** to bridge long distance store-load communications. This long distance store-load dependence is more difficult to predict and **DMDP** has slightly fewer mispredictions.

Consistency model RMO was simulated as well. Stores were allowed to commit in an out-of-order manner. SSN_{commit} is set to the one preceding the oldest store in store buffer. When a store is committed, its corresponding entry in *Store Register Buffer* is invalidated and forwarding is prohibited. The results shows **DMDP** surpasses **NoSQ** by 7.67% (Int) and 4.08% (FP).

5.6.8 Energy Efficiency

Figure 5.15 illustrates the EDP (Energy Delay Product) of **DMDP** normalized to **NoSQ**.

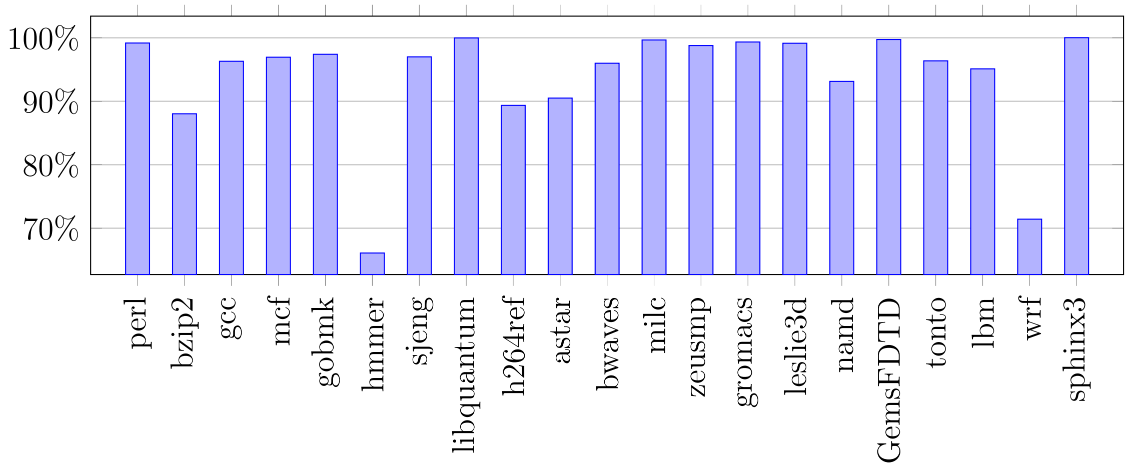


Figure 5.15: The EDP of **DMDP**, normalized to **NoSQ**

As is observed, **DMDP** reduces the total execution time in every benchmark (Figure 5.12) and slightly consumes more energy due to the extra predicated instructions. Overall, **DMDP** is more power efficient than **NoSQ** and saves 8.5% (Int) and 5.1% EDP on average. The EDP result is correlated to the memory dependence misprediction rate (Table 5.6), more mispredictions result in more energy consumption.

DMDP has more mispredictions in *bzip2* and that costs **DMDP** to consume about 3% more energy. On the other hand, **DMDP** saves around 2 MPKI in *hmmer* which leads to a 15% energy saving.

5.7 Related Work

Instruction predication was initially used to convert control dependencies to data dependencies [115], so that vectorization could be applied even if a loop contained conditional branches. With the mechanism, qualified branches are converted to predicate computation instructions statically during the compilation. The control dependent instructions are guarded by these predicates to eliminate conditional branches. No branches means no mispredictions and no recovery penalties at all, especially if the eliminated branches are hard to predict. Dynamic predication is also proposed to insert predicates during the run time [107] for processors which do not have a predicated Instruction Set Architecture (ISA).

Store-queue-free architectures were proposed when memory dependence prediction accuracy became acceptable. Sha et al. designed **NoSQ** [17] to completely remove the store queue by using memory cloaking to communicate the in-flight stores and loads. When the dependence is inconsistent, **NoSQ** delays the load execution, therefore, **NoSQ** still needs a structure to hold all the delayed loads until they are ready to execute. In contrast, **DMDP** inserts predication which converts load-store dependencies into simple data dependencies, therefore it does not need this extra storage at the expense of executing additional operations.

Subramaniam et al. proposed Fire-and-Forget (FnF) [103] to eliminate the store queue in a different way. Instead of predicting the load, FnF predicts the store so that when a store is decoded, its consumer load is predicted and the store forwards its data to that load. We selected **NoSQ** as our reference design since some memory dependencies are path sensitive. When FnF is predicting a store, the branches between the store and the dependent load are not considered.

Several related works attempt to reduce the size of store queue or simplify it. The mechanism proposed by Sha et al. predicted the index of the colliding store in the store queue [97]. This approach eliminates the need to search the store queue when a load is executed. Their mechanism is similar to **DMDP** in that both techniques require address comparison. The load retrieves its data from either the store queue or the data cache. The difference is that the address comparison is not triggered until the store is executed in their mechanism, whereas **DMDP** only waits for the store address computation.

Stone et al. separated the function of the store queue into three different structures [96]: store forwarding cache (SFC) for memory forwarding; memory disambiguation table (MDT) for memory ordering violation detection; and a store FIFO for in-order store retirement. CAM structures are completely replaced by RAM structures since no associative search is needed. Garg et al. had a different view of the store queue and proposed SMDE (Slackened Memory Dependence Enforcement) [100]. Their mechanism uses an L0 cache to bridge the in-flight loads and stores similar to a store queue. This simple design does not need any associative search so the size scales well. Sethumadhavan et al. proposed *Late-Binding* [116] to allocate LSQ at the issue stage instead of the dispatch stage which reduces the demand for LSQ entries. Since the LSQ allocation is completely out-of-order, an age tag is explicitly integrated into each LSQ entry. The current commercial processors are still using store queue to disambiguate memory ordering. Kim et al. designed a mechanism, store dependence prediction (SDP) [117], to indicate which store is not likely to collide with any younger load so that the load does not need to wait for this store address computation.

Perais et al. expanded the store-load bypassing to load-load bypassing with a new register sharing mechanism [18]. They introduced an Instruction Distance Predictor to predict the instruction which produces the load result. A TAGE-like [14] predictor was designed and could also be tuned as a Store Distance Predictor and adopted to **DMDP**.

5.8 Summary

Although predication was proposed a long time ago to mitigate the branch mispredictions, it has never been employed to convert data dependencies through memory to register dependencies.

DMDP is the first mechanism which does not require any special buffer or queue to keep memory instructions. Therefore, the operation of memory instructions only needs to consider data dependence, similar to ALU instructions. **DMDP** converts a load to i) direct access to cache; ii) reuse of the colliding store data; iii) predication between cache and the colliding store data. Therefore, the only hardware change is predication insertion at the rename stage and a consumer counter for expanding the lifetime of store's physical registers.

Our evaluation of the mechanism shows that **DMDP** surpasses **NoSQ** in all benchmarks (7.17% Int, 4.48% FP). Meanwhile, **DMDP** works more power efficiently as well and saves EDP (8.5% Int, 5.1% FP).

Chapter 6

Conclusion

In this work, we designed and implemented four innovative techniques to effectively mitigate the misspeculation recovery penalties. The current superscalar processors highly rely on speculative execution and we expect more speculative mechanisms will be involved in the future. Instead of working on a more accurate prediction, how to efficiently recover from a misspeculation have become an increasingly important issue. This dissertation includes all of our efforts to address this concern and excellent results have been achieved and presented.

Mower (Chapter 2) was first proposed to mitigate the state restoration delay. Two dependence matrices are designed to track the branch and register mapping dependencies. With these matrices, F-RAT (Front-end Register Alias Table) do not need to be treated as a single entity. Instead, each register mapping has its own valid bit to indicate if this mapping is correct after a misprediction is detected. Therefore, the rename process continues as long as the operands of the newly fetched instructions still have a correct mapping.

In addition to the dependence mechanism, Mower also has a broadcast network to selectively eliminate invalid instructions left in the back-end of the pipeline. A special walker is used to walk from the misprediction to the ROB tail to eliminate invalid instructions. Simultaneously, F-RAT is recovered by undoing the mapping updates. Because the walking direction in Mower is reversed, the mapping recovery is much faster than a conventional walking mechanism.

Mower is a general recovery mechanism which can effectively shorten the state restoration delay. Its drawback is, it requires extra hardware overhead and many design modifications. In order to simplify the design, we proposed a two-phase recovery algorithm in Chapter 3.

In two-phase recovery mechanism, the first phase is exactly the same as a basic recovery mechanism which can cover most of the recovery scenarios. The second phase is launched only when the retire stage is stalled by a long latency execution such as an LLC (Last Level Cache) miss. In the second phase, all of the instructions left in the pipeline are treated as invalid instructions and will be eliminated. Therefore, the penalty to recover the misspeculation is overlapped with cache miss penalty. Two-phase reuses many structures in the basic recovery mechanism and is very simple and efficient.

Other than state restoration delay, pipeline fill delay is also covered. We designed and described Passing Loop in Chapter 4. In order to exploit control independence, the convergence point is usually predicted. In Passing Loop, only short forward branches are considered. Therefore, the convergence point is always the branch target. Before the misprediction is detected, the CD (Control Dependent) instructions on both paths are fetched and only predicted path instructions are executed. During the recovery, the predicted path (wrong) instructions are eliminated and the alternative path (correct) instructions are inserted to the reservation station and executed. Passing Loop is much simpler than other techniques which exploit control independence. It focuses on the short forward branches to simplify the necessary hardware.

Memory dependence mispredictions account for another part of misspeculations. Dynamic Memory Dependence Predication (DMDP) is proposed to minimize the number of memory dependence mispredictions (Chapter 5). In a store-queue-free architecture, when the dependence of a store-load pair is inconsistent, the load instruction has to wait for the related store to commit. This extra delay is detrimental to the overall performance.

DMDP inserts a predicate to compare the address of the load and the store. If the addresses match, the store data is forwarded to the load instruction. Otherwise, the data read from the cache is forwarded to the load instruction. As a result, the load is executed at its earliest time. DMDP works even better in a multi-core system where store instructions have to commit in program order.

These four innovative mechanisms cover most of the design space of misspeculation recoveries, including branch mispredictions and memory dependence mispredictions. This work provides a quantitative view over the concerned problems and have a thorough quantitative analysis for the proposed mechanisms. A technique such as the two-phase recovery mechanism is very practical and we believe it can be implemented in a real hardware and improve the computation performance.

References

[1] Held, J.; Bautista, J.; Koehl, S. *white paper, Intel* **2006**.

[2] Asanovic, K.; Bodik, R.; Demmel, J.; Keaveny, T.; Keutzer, K.; Kubiatowicz, J.; Morgan, N.; Patterson, D.; Sen, K.; Wawrzynek, J.; Wessel, D.; Yelick, K. Vol. 52, pages 56–67, New York, NY, USA, 2009. ACM.

[3] Amdahl, G. M. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[4] Hill, M. D.; Marty, M. R. *Computer* **2008**, *41*(7), 33–38.

[5] Woo, D. H.; Lee, H.-H. S. *Computer* **2008**, *41*(12), 24–31.

[6] Lee, J. K. F.; Smith, A. J. *Computer* **1984**, *17*(1), 6–22.

[7] Perleberg, C. H. Branch target buffer design Technical report, Berkeley, CA, USA, **1989**.

[8] Yeh, T.-Y.; Patt, Y. N. In *Proceedings of the 24th Annual International Symposium on Microarchitecture*, MICRO 24, pages 51–61, New York, NY, USA, 1991. ACM.

[9] Lipasti, M. H.; Shen, J. P. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29, pages 226–237, Washington, DC, USA, 1996. IEEE Computer Society.

[10] Sazeides, Y.; Smith, J. E. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 30, pages 248–258, Washington, DC, USA, 1997. IEEE Computer Society.

[11] Wang, K.; Franklin, M. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 30, pages 281–290, Washington, DC, USA, 1997. IEEE Computer Society.

[12] Moshovos, A.; Sohi, G. S. *Int. J. Parallel Program.* **1999**, *27*(6), 427–456.

[13] Moshovos, A. I. *Memory Dependence Prediction* PhD thesis, The University of Wisconsin - Madison, **1998**.

[14] Seznec, A. In *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ISCA '05, pages 394–405, Washington, DC, USA, 2005. IEEE Computer Society.

[15] Seznec, A.; Michaud, P. *Journal of Instruction Level Parallelism* **2006**, *8*, 1–23.

[16] Chrysos, G. Z.; Emer, J. S. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, ISCA '98, pages 142–153, Washington, DC, USA, 1998. IEEE Computer Society.

[17] Sha, T.; Martin, M. M. K.; Roth, A. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 285–296, Washington, DC, USA, 2006. IEEE Computer Society.

[18] Perais, A.; Seznec, A. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 694–706, 2016.

[19] Seznec, A. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 117–127, New York, NY, USA, 2011. ACM.

[20] Seznec, A. In *JILP - Championship Branch Prediction*, Minneapolis, United States, 2014.

[21] Shen, J. P.; Lipasti, M. H. *Modern processor design: fundamentals of superscalar processors;* Waveland Press, 2013.

[22] Jourdan, S.; Stark, J.; Hsing, T.-H.; Patt, Y. N. *Int. J. Parallel Program.* **1997**, *25*(5), 363–383.

[23] Hinton, G.; Sager, D.; Upton, M.; Boggs, D.; others. In *Intel Technology Journal.* Citeseer, 2001.

[24] Armstrong, D. N.; Kim, H.; Mutlu, O.; Patt, Y. N. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pages 119–128, Washington, DC, USA, 2004. IEEE Computer Society.

[25] Method and apparatus for managing instruction flushing in a microprocessor's instruction pipeline. McIlvaine, M. S.; Dieffenderfer, J. N.; Sartorius, T. A. **2011**.

[26] Akkary, H.; Rajwar, R.; Srinivasan, S. T. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 423–, Washington, DC, USA, 2003. IEEE Computer Society.

[27] Akkary, H.; Rajwar, R.; Srinivasan, S. T. *IEEE Micro* **2003**, *23*(6), 11–19.

[28] Rotenberg, E.; Jacobson, Q.; Smith, J. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, HPCA '99, pages 115–, Washington, DC, USA, 1999. IEEE Computer Society.

[29] Jin, Z.; Aşilioğlu, G.; Önder, S. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 285–294, New York, NY, USA, 2015. ACM.

[30] Moudgill, M.; Pingali, K.; Vassiliadis, S. In *Proceedings of the 26th Annual International Symposium on Microarchitecture*, MICRO 26, pages 202–213, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[31] Hwu, W. W.; Patt, Y. N. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, ISCA '87, pages 18–26, New York, NY, USA, 1987. ACM.

[32] Yeager, K. C. *IEEE Micro* **1996**, *16*(2), 28–40.

[33] Kessler, R. E. *IEEE Micro* **1999**, *19*(2), 24–36.

[34] Cristal, A.; Ortega, D.; Llosa, J.; Valero, M. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, HPCA '04, pages 48–, Washington, DC, USA, 2004. IEEE Computer Society.

[35] Cristal, A.; Santana, O. J.; Valero, M.; Martínez, J. F. *ACM Trans. Archit. Code Optim.* **2004**, *1*(4), 389–417.

[36] Zhou, P.; Önder, S.; Carr, S. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, pages 41–50, New York, NY, USA, 2005. ACM.

[37] Akl, P.; Moshovos, A. In *Proceedings of the 20th Annual International Conference on Supercomputing*, ICS '06, pages 36–45, New York, NY, USA, 2006. ACM.

[38] Akl, P.; Moshovos, A. In *Proceedings of the 3rd International Conference on High Performance Embedded Architectures and Compilers*, HiPEAC'08, pages 258–272, Berlin, Heidelberg, 2008. Springer-Verlag.

[39] Latorre, F.; Magklis, G.; González, J.; Chaparro, P.; González, A.; Springer-Verlag: Berlin, Heidelberg, 2011; chapter CROB: Implementing a Large Instruction Window Through Compression, pages 115–134.

[40] Golander, A.; Weiss, S.; Springer-Verlag: Berlin, Heidelberg, 2009; chapter Reexecution and Selective Reuse in Checkpoint Processors, pages 242–268.

[41] Golander, A.; Weiss, S. *ACM Trans. Archit. Code Optim.* **2009**, *6*(3), 10:1–10:27.

[42] Hilton, A.; Eswaran, N.; Roth, A. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, pages 159–168, Washington, DC, USA, 2009. IEEE Computer Society.

[43] Dual prediction branch system having two step of branch recovery process which activated only when mispredicted branch is the oldest instruction in the out-of-order unit. Hoyt, B. D.; Hinton, G. J.; Papworth, D. B.; Gupta, A. K.; Fetterman, M. A.; Natarajan, S.; Shenoy, S.; D'sa, R. V. **1998**.

[44] Goshima, M.; Nishino, K.; Kitamura, T.; Nakashima, Y.; Tomita, S.; Mori, S.-i. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 34, pages 225–236, Washington, DC, USA, 2001. IEEE Computer Society.

[45] Sassone, P. G.; Rupley, II, J.; Brekelbaum, E.; Loh, G. H.; Black, B. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 335–346, New York, NY, USA, 2007. ACM.

[46] Önder, S.; Gupta, R. In *Proceedings of the 1998 International Conference on Computer Languages*, ICCL '98, pages 80–, Washington, DC, USA, 1998. IEEE Computer Society.

[47] Patterson, D. A.; Hennessy, J. L. *Computer Organization and Design MIPS Edition: The Hardware/Software Interface;* Newnes, 2013.

[48] Henning, J. L. *SIGARCH Comput. Archit. News* **2006**, *34*(4), 1–17.

[49] Brooks, D.; Tiwari, V.; Martonosi, M. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, pages 83–94, New York, NY, USA, 2000. ACM.

[50] Li, S.; Ahn, J. H.; Strong, R. D.; Brockman, J. B.; Tullsen, D. M.; Jouppi, N. P. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 469–480, New York, NY, USA, 2009. ACM.

[51] Cruz, J.-L.; González, A.; Valero, M.; Topham, N. P. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, pages 316–325, New York, NY, USA, 2000. ACM.

[52] Golander, A.; Weiss, S. *ACM Trans. Archit. Code Optim.* **2008**, *4*(4), 6:1–6:32.

[53] Branch recovery mechanism to reduce processor front end stall time by providing path information for both correct and incorrect instructions mixed in the instruction pool. Kyker, A. B.; Boggs, D. D. **2000**.

[54] Dixon, M.; Hammarlund, P.; Jourdan, S.; Singhal, R. *Intel Technology Journal* **2010**, *14*(3).

[55] Burger, D.; Austin, T. M. *SIGARCH Comput. Archit. News* **1997**, *25*(3), 13–25.

[56] Perelman, E.; Hamerly, G.; Van Biesbrouck, M.; Sherwood, T.; Calder, B. *SIGMETRICS Perform. Eval. Rev.* **2003**, *31*(1), 318–319.

[57] Sherwood, T.; Perelman, E.; Calder, B. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, PACT '01, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[58] Rosenfeld, P.; Cooper-Balis, E.; Jacob, B. *IEEE Comput. Archit. Lett.* **2011**, *10*(1), 16–19.

[59] Micron. `https://www.micron.com/resource-details/e570d65b-2664-4037-a141-620a6f2e58e7`.

[60] Branch prediction research. INRIA, T. `https://team.inria.fr/alf/members/andre-seznec/branch-prediction-research/`.

[61] Seznec, A. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, pages 443–454, Washington, DC, USA, 2011. IEEE Computer Society.

[62] Rotenberg, E.; Smith, J. In *Proceedings of the 32Nd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 32, pages 4–15, Washington, DC, USA, 1999. IEEE Computer Society.

[63] Riseman, E. M.; Foster, C. C. *IEEE Trans. Comput.* **1972**, *21*(12), 1405–1411.

[64] Lam, M. S.; Wilson, R. P. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ISCA '92, pages 46–57, New York, NY, USA, 1992. ACM.

[65] Austin, T. M.; Sohi, G. S. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ISCA '92, pages 342–351, New York, NY, USA, 1992. ACM.

[66] Uht, A. K. *Hardware Extraction of Low-level Concurrency from Sequential Instruction Streams (Parallelism, Implementation, Architecture, Dependencies, Semantics)* PhD thesis, Pittsburgh, PA, USA, **1985**.

[67] Wang, S. S.; Uht, A. K. In *Proceedings of the 23rd Annual Workshop and Symposium on Microprogramming and Microarchitecture*, MICRO 23, pages 125–134, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[68] Uht, A. K. *IEEE Trans. Parallel Distrib. Syst.* **1992**, *3*(5), 573–581.

[69] Uht, A. K. *SIGARCH Comput. Archit. News* **1993**, *21*(3), 5–12.

[70] Uht, A. K.; Sindagi, V.; Hall, K. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, MICRO 28, pages 313–325, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press.

[71] Chen, T.-F. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, HPCA '98, pages 185–, Washington, DC, USA, 1998. IEEE Computer Society.

[72] Heil, T. H.; Smith, J. E. Selective dual path execution Technical report, Technical report, University of Wisconsin-Madison, **1996**.

[73] Tyson, G.; Lick, K.; Farrens, M. Limited dual path execution Technical report, Technical Report CSE-TR 346-97, University of Michigan, **1997**.

[74] Ahuja, P. S.; Skadron, K.; Martonosi, M.; Clark, D. W. In *Proceedings of the 12th International Conference on Supercomputing*, ICS '98, pages 101–108, New York, NY, USA, 1998. ACM.

[75] Klauser, A.; Paithankar, A.; Grunwald, D. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, ISCA '98, pages 250–259, Washington, DC, USA, 1998. IEEE Computer Society.

[76] Klauser, A.; Grunwald, D. In *Proceedings of the 32Nd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 32, pages 38–47, Washington, DC, USA, 1999. IEEE Computer Society.

[77] Aragón, J. L.; González, J.; González, A.; Smith, J. E. In *Proceedings of the 16th International Conference on Supercomputing*, ICS '02, pages 220–229, New York, NY, USA, 2002. ACM.

[78] Mahlke, S. A.; Hank, R. E.; McCormick, J. E.; August, D. I.; Hwu, W.-M. W. In *Proceedings of the 22Nd Annual International Symposium on Computer Architecture*, ISCA '95, pages 138–150, New York, NY, USA, 1995. ACM.

[79] Wallace, S.; Calder, B.; Tullsen, D. M. *SIGARCH Comput. Archit. News* **1998**, *26*(3), 238–249.

[80] Mantripragada, S.; Nicolau, A. In *Proceedings of the 14th International Conference on Supercomputing*, ICS '00, pages 206–214, New York, NY, USA, 2000. ACM.

[81] Chuang, W.; Calder, B. In *Proceedings of the 17th Annual International Conference on Supercomputing*, ICS '03, pages 183–192, New York, NY, USA, 2003. ACM.

[82] Kim, H.; Mutlu, O.; Stark, J.; Patt, Y. N. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, pages 43–54, Washington, DC, USA, 2005. IEEE Computer Society.

[83] Quiñones, E.; Parcerisa, J.-M.; Gonzalez, A. In *Proceedings of the 20th Annual International Conference on Supercomputing*, ICS '06, pages 46–54, New York, NY, USA, 2006. ACM.

[84] Sodani, A.; Sohi, G. S. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, pages 194–205, New York, NY, USA, 1997. ACM.

[85] Chou, Y.; Fung, J.; Shen, J. P. In *Proceedings of the 13th International Conference on Supercomputing*, ICS '99, pages 109–118, New York, NY, USA, 1999. ACM.

[86] Cher, C.-Y.; Vijaykumar, T. N. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 34, pages 4–15, Washington, DC, USA, 2001. IEEE Computer Society.

[87] Kane, G. *Chapter* **1989**, *2*, 2–6.

[88] Collins, J. D.; Tullsen, D. M.; Wang, H. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pages 129–140, Washington, DC, USA, 2004. IEEE Computer Society.

[89] Gandhi, A.; Akkary, H.; Srinivasan, S. T. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, HPCA '04, pages 254–, Washington, DC, USA, 2004. IEEE Computer Society.

[90] Al-Zawawi, A. S.; Reddy, V. K.; Rotenberg, E.; Akkary, H. H. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 448–459, New York, NY, USA, 2007. ACM.

[91] Hilton, A. D.; Roth, A. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 436–447, New York, NY, USA, 2007. ACM.

[92] Premillieu, N.; Seznec, A. *ACM Trans. Archit. Code Optim.* **2012**, *8*(4), 43:1–43:20.

[93] Palacharla, S.; Jouppi, N. P.; Smith, J. E. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, pages 206–218, New York, NY, USA, 1997. ACM.

[94] Onder, S.; Gupta, R. In *Proceedings of the 32Nd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 32, pages 170–176, Washington, DC, USA, 1999. IEEE Computer Society.

[95] Torres, E. F.; Ibanez, P.; Vinals, V.; Llaberia, J. M. In *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ISCA '05, pages 469–480, Washington, DC, USA, 2005. IEEE Computer Society.

[96] Stone, S. S.; Woley, K. M.; Frank, M. I. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, pages 171–182, Washington, DC, USA, 2005. IEEE Computer Society.

[97] Sha, T.; Martin, M. M. K.; Roth, A. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, pages 159–170, Washington, DC, USA, 2005. IEEE Computer Society.

[98] Sethumadhavan, S.; Desikan, R.; Burger, D.; Moore, C. R.; Keckler, S. W. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 399–, Washington, DC, USA, 2003. IEEE Computer Society.

[99] Park, I.; Ooi, C. L.; Vijaykumar, T. N. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 411–, Washington, DC, USA, 2003. IEEE Computer Society.

[100] Garg, A.; Rashid, M. W.; Huang, M. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ISCA '06, pages 142–154, Washington, DC, USA, 2006. IEEE Computer Society.

[101] Gandhi, A.; Akkary, H.; Rajwar, R.; Srinivasan, S. T.; Lai, K. In *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ISCA '05, pages 446–457, Washington, DC, USA, 2005. IEEE Computer Society.

[102] Baugh, L.; Zilles, C. *IBM J. Res. Dev.* **2006**, *50*(2/3), 287–297.

[103] Subramaniam, S.; Loh, G. H. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 273–284, Washington, DC, USA, 2006. IEEE Computer Society.

[104] Hilton, A.; Roth, A. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 245–254, New York, NY, USA, 2009. ACM.

[105] Wenisch, T. F.; Ailamaki, A.; Falsafi, B.; Moshovos, A. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 266–277, New York, NY, USA, 2007. ACM.

[106] Mahlke, S. A.; Lin, D. C.; Chen, W. Y.; Hank, R. E.; Bringmann, R. A. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, MICRO 25, pages 45–54, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[107] Klauser, A.; Austin, T.; Grunwald, D.; Calder, B. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, PACT '98, pages 278–, Washington, DC, USA, 1998. IEEE Computer Society.

[108] Roth, A. In *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ISCA '05, pages 458–468, Washington, DC, USA, 2005. IEEE Computer Society.

[109] Roth, A. *Journal of Instruction Level Parallelism* **2006**, *8*, 22.

[110] Yoaz, A.; Erez, M.; Ronen, R.; Jourdan, S. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, ISCA '99, pages 42–53, Washington, DC, USA, 1999. IEEE Computer Society.

[111] Roth, A. *IEEE Comput. Archit. Lett.* **2008**, *7*(1), 9–12.

[112] Ergin, O.; Balkan, D.; Ponomarev, D.; Ghose, K. In *Proceedings of the IEEE International Conference on Computer Design*, ICCD '04, pages 480–487, Washington, DC, USA, 2004. IEEE Computer Society.

[113] Lepak, K. M.; Lipasti, M. H. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 33, pages 22–31, New York, NY, USA, 2000. ACM.

[114] Chou, Y.; Spracklen, L.; Abraham, S. G. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, pages 183–196, Washington, DC, USA, 2005. IEEE Computer Society.

[115] Allen, J. R.; Kennedy, K.; Porterfield, C.; Warren, J. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '83, pages 177–189, New York, NY, USA, 1983. ACM.

[116] Sethumadhavan, S.; Roesner, F.; Emer, J. S.; Burger, D.; Keckler, S. W. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 347–357, New York, NY, USA, 2007. ACM.

[117] Method and apparatus for store dependence prediction. Kim, H.-S.; Chappell, R. S.; Soo, C. Y. **2017**.

Appendix A

Copyright Permission

ASSOCIATION FOR COMPUTING MACHINERY, INC. LICENSE TERMS AND CONDITIONS

Mar 15, 2018

This Agreement between Michigan Technological University -- Zhaoxiang Jin ("You") and Association for Computing Machinery, Inc. ("Association for Computing Machinery, Inc.") consists of your license details and the terms and conditions provided by Association for Computing Machinery, Inc. and Copyright Clearance Center.

All payments must be made in full to CCC. For payment instructions, please see information listed at the bottom of this form.

License Number	4310301417821
License date	Mar 15, 2018
Licensed Content Publisher	Association for Computing Machinery, Inc.
Licensed Content Publication	Proceedings
Licensed Content Title	Mower: A New Design for Non-blocking Misprediction Recovery
Licensed Content Author	Zhaoxiang Jin, et al
Licensed Content Date	Jun 8, 2015
Type of Use	Thesis/Dissertation
Requestor type	Author of this ACM article
Is reuse in the author's own new work?	Yes
Format	Print and electronic
Portion	Full article
Will you be translating?	No
Order reference number	
Title of your thesis/dissertation	Mitigating the effect of Misspeculations in Superscalar Processors
Expected completion date	May 2018
Estimated size (pages)	180
Requestor Location	Michigan Technological University 2001F, Woodmar Dr HOUGHTON, MI 49931 United States Attn: Zhaoxiang Jin
Billing Type	Credit Card
Credit card info	Master Card ending in 7794
Credit card expiration	10/2018
Total	**8.00 USD**