



Michigan Technological University
Create the Future Digital Commons @ Michigan Tech

Dissertations, Master's Theses and Master's Reports - Open

Dissertations, Master's Theses and Master's Reports

2015

MOWER : A NEW DESIGN FOR NON-BLOCKING MIS PREDICTION RECOVERY

Zhaoxiang Jin
Michigan Technological University

Follow this and additional works at: <https://digitalcommons.mtu.edu/etds>


 Part of the [Computer Sciences Commons](#)

Copyright 2015 Zhaoxiang Jin

Recommended Citation

Jin, Zhaoxiang, "MOWER : A NEW DESIGN FOR NON-BLOCKING MIS PREDICTION RECOVERY", Master's Thesis, Michigan Technological University, 2015.
<https://digitalcommons.mtu.edu/etds/918>

Follow this and additional works at: <https://digitalcommons.mtu.edu/etds>

 Part of the [Computer Sciences Commons](#)

MOWER : A NEW DESIGN FOR NON-BLOCKING MIS PREDICTION
RECOVERY

By
Zhaoxiang Jin

A THESIS

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

In Computer Science

MICHIGAN TECHNOLOGICAL UNIVERSITY

2015

© 2015 Zhaoxiang Jin

This thesis has been approved in partial fulfillment of the requirements for the Degree of MASTER OF SCIENCE in Computer Science.

Department of Computer Science

Thesis Advisor: *Dr. Soner Onder #1*

Committee Member: *Dr. Zhenlin Wang #1*

Committee Member: *Dr. Saeid Nooshabadi #2*

Committee Member: *Dr. Nilufer Onder #3*

Committee Member: *Dr. Zhuo Feng #4*

Department Chair: *Dr. Min Song*

Contents

List of Figures	vii
List of Tables	ix
Acknowledgments	xi
Abstract	xiii
1 Introduction	1
2 Background	7
2.1 Instruction Level Parallelism	8
2.2 Register Renaming	10
2.3 Speculation	12
2.3.1 Branch Prediction	13
2.3.2 Reorder Buffer	15
2.4 Recovery	17
2.5 Summary	18
3 Mower	19

3.1	General Overview	19
3.2	Explicit Dependency Tracking and Out-of-order Branch Resolution	23
3.3	Tracking Rename Map Validity	25
3.4	Tracking Branch Dependencies	27
3.5	Recovery Timing and Details	28
4	Hardware Design	33
4.1	Renaming Branches	33
4.2	Bit Matrices	34
4.3	Fixing The RAT	36
5	Evaluation	39
5.1	Methodology	39
5.2	Performance Results	41
5.3	Power Estimation	46
6	Related Work	51
7	Conclusion	57
7.1	Contribution of the thesis	57
7.2	Future work	58
	References	61

List of Figures

2.1	The bubbles in the pipeline	9
2.2	Instruction Level Parallelism	9
2.3	Register Renaming	11
2.4	History-Based Branch Prediction	14
2.5	Branch Target Buffer	15
2.6	Reorder Buffer	16
2.7	Misprediction Recovery	18
3.1	Mower Block Diagram	20
3.2	Mow all the invalid instructions through ROB, release all the relative resources.	24
3.3	Explicit Dependency Tracking	25
3.4	Interleaved reclamation and dispatching on ROB	29
3.5	Fix the RAT by the reclamation	30
4.1	Branch Dependency Matrix	35
4.2	Recovering RAT from MDM	37

5.1	IPC vs In-flight Branches	42
5.2	The Damaged F-RAT through the recovery process	42
5.3	The average number of invalid instructions left in the pipeline when misprediction is detected	43
5.4	Spec2006 Integer Speedup	44
5.5	Spec2006 Float Speedup	44
5.6	Power Evaluation Spec2006 Integer	46
5.7	Power Evaluation Spec2006 Float	46
5.8	EDP normalized to baseline configuration	49

List of Tables

5.1	The configuration of the simulation	40
5.2	Misprediction Rates for Spec2006INT	45
5.3	Power distribution in Spec2006	47

Acknowledgments

I would like to thank all those who have helped me learn, understand and appreciate this subject as well as those who helped me with \LaTeX .

I started to work on my thesis one year ago and it is my first time to do research work at an academic level. Find out the problem, abstract it and come with the solution and test it. During this time, Dr Soner helped me, guided me on the right way to accomplish this mission. Your top-down strategy in writing is very efficient and make the writing not boring any more. I appreciate the help Gorkem gave in the survey of the related work. With your thorough help I could get a clear sense of what has been done and what is the contribution of my work. Murat, thank you for your hard work on the cache implementation. We all know that's not easy and you figure that out.

At this moment, how can I forget my lovely parents and my best friends. Without your support and encouragement, it is impossible for me to finish my MSc degree and make contributions to the field of Computer Architecture. Thank you to my parents, Jianmin Jin and Huizhu Xu, I miss your delicious traditional Chinese food so much. Thanks to all my labmates for your kindness and friendship.

Abstract

Mower is a micro-architecture technique which targets branch misprediction penalties in superscalar processors. It speeds-up the misprediction recovery process by dynamically evicting stale instructions and fixing the RAT (Register Alias Table) using explicit branch dependency tracking. Tracking branch dependencies is accomplished by using simple bit matrices. This low-overhead technique allows overlapping of the recovery process with instruction fetching, renaming and scheduling from the correct path. Our evaluation of the mechanism indicates that it yields performance very close to ideal recovery and provides up to 5% speed-up and 2% reduction in power consumption compared to a traditional recovery mechanism using a reorder buffer and a walker. The simplicity of the mechanism should permit easy implementation of Mower in an actual processor.

Chapter 1

Introduction

Branch prediction is a vital component in any contemporary processor. Today, most superscalar processors are organized in the form of decoupled architectures where an in-program-order front-end chases the instruction stream by relying on accurate branch prediction and an out-of-program-order back-end schedules and executes these instructions based on operand availability. The performance of such an architecture is dependent on not only the accuracy of branch prediction, but also the cost of branch misprediction. A branch misprediction leaves the two halves of the processor in an incorrect state which needs to be corrected before instruction fetch and execution can resume.

When a particular branch is detected to be mispredicted, the processor needs to do

the following: (1) Eliminate all the instructions which follow the mispredicted branch; (2) Restore the processor structures, such as rename map tables which are frequently referred to as *Register Alias Table (RAT)*, to their correct values; (3) Resume fetching from the correct branch target.

Each of these tasks contribute to the branch misprediction penalty and each must be targeted individually to reduce its impact on processor performance. In other words, to be effective, any technique which targets branch misprediction penalty must reduce both the duration of each task and permit overlapping of individual tasks to the maximum extent possible. Eliminating instructions following the mispredicted branch requires knowledge of where in the processor these instructions are and a mechanism to nullify them. Restoring the processor structures requires knowledge of what constitutes the correct processor state [1]. Resuming the fetching from the correct target requires knowledge of the correct target and the resources necessary to store the new instructions. Provision of resources is particularly important if fetching is to resume before the invalidation is complete as newly fetched instructions cannot be mixed with those already in the pipeline. As a result, either a separate set of locations must be provided to store the correct instructions or a mechanism should exist to distinguish the two. As we will discuss shortly, Mower accomplishes a reduction in the duration of all three tasks and maximises their overlap, in essence reducing the overall branch misprediction penalty.

In order to understand the issues involved, revisiting a commonly used basic recovery mechanism [2] will be helpful. This simple recovery scheme incorporates a *reorder buffer* (ROB) into which instructions are fed in program-order as they are fetched. A *front-end* register alias table (F-RAT) is used to speculatively rename the incoming instruction stream and a *retirement* register alias table (R-RAT) is used to record the in-order state as the instructions complete and retire. The F-RAT is updated when instructions are being renamed and the R-RAT is updated when an instruction *commits*, i.e., it is determined to be part of the in-order state. Once a branch misprediction is detected, instruction fetching is halted and the execution of existing instructions continues until the mispredicted branch arrives at the head of the reorder buffer. At this point, it is known that all the instructions still in the pipeline are invalid and the R-RAT has the correct in-order state. Waiting for the completion of prior instructions leaves only invalid instructions in the pipeline and hence promotes the development of simple and fast mechanisms to squash invalid instructions by making the identification of invalid instructions trivial and restoring the correct state easy. When the branch reaches the head of ROB the retirement map table R-RAT contains the correct state which can simply be copied over F-RAT. On the other hand, this simple mechanism does not permit overlapping of subtasks involved in the recovery process and accomodates a highly variable misprediction penalty that is strongly correlated with the position of the mispredicted branch in the ROB. When the branch is close to the ROB head, it takes less time for the rear-end to retire the instructions preceding the

mispredicted branch. However, if the mispredicted branch is far from the ROB head, or, if the rear-end is waiting for a long latency operation ahead of the branch (such as a missed load), the time to recover from the misprediction may increase significantly.

Branch misprediction penalty has been the target of several techniques which can be broadly classified into two groups: those which target the pipeline fill delay after a misprediction and those concerned mainly with state maintenance. Techniques which target the pipeline fill delay are orthogonal to Mower and we do not discuss them further except in related work. Existing state maintenance techniques can be grouped into *checkpointing* and (ROB) *walking*. Checkpointing is a technique where a copy of the RAT is made for each potential misprediction. When a misprediction is detected, the correct copy of the RAT is restored immediately from one of the available checkpoints. The technique also permits fast invalidation of instructions within the processor as the required information is readily available and part of the checkpoint. Since checkpointing requires a copy of the RAT per checkpoint, each checkpoint may require a significant amount of space. As a result, taking a checkpoint at every branch may be prohibitively expensive. Several common modifications to the base checkpointing technique are available to address this. An example of these modifications is to take checkpoints only on low confidence branches and using the basic recovery technique outlined above to recover from branches that do not have checkpoints [3].

Walking is a technique where the RAT is corrected by walking over the ROB. Walking over the ROB from the ROB head up to the mispredicted branch while pseudo-retiring each instruction in between will construct the correct R-RAT, which can then be used to restore the front-end state as well. Since the processor does not actually have to wait for instructions to complete to construct the alias table, this technique fetches instructions from the correct path faster than waiting. Alternatively, we can forego the use of a retirement alias table and store the previous mapping values in ROB entries. Then, we can walk over the ROB starting from the tail up to the mispredicted branch and undo each change that was made in error [2].

Mower attempts to combine several concepts from the basic recovery techniques such as checkpointing and walking, and provides complete overlapping of the recovery tasks. It accomplishes rapid elimination of invalid instructions by explicitly keeping track of the dependencies by means of simple bit matrices. It also permits immediate fetching of instructions from the correct path upon a misprediction detection by (1) selectively blocking entries in the front-end map table RAT while the recovery process is continuing; (2) by relying on a novel walking mechanism which rapidly restores the state and *mows* the invalid instructions to provide the space where new instructions can be stored.

The rest of the thesis is organized as follows: In Chapter 2, the necessary background is provided. Chapter 3 gives an overview of the technique we propose, Mower.

Chapter 4 provides the detailed design. In Chapter 5 we present our experimental evaluation and analysis. In Chapter 6, we review the recent techniques and make comparisons with Mower. Finally, we summarize our critical findings in Chapter 7.

Chapter 2

Background

This chapter is written to help the readers to fully understand the concepts used and contributions made by this thesis. It contains the background information to understand the functioning of a contemporary superscalar processor. We will start with the explanation of the fundamental speedup factor in modern superscalar processor design, instruction level parallelism. Then we will discuss more about the rename strategy and how it solves the false data dependence problem. Following, the difference between speculative state and in-order state is illustrated. Finally, the recovery process will be discussed when speculation is wrong and the in-order state needs to be restored.

2.1 Instruction Level Parallelism

When a simple in-order pipelined processor is running, instructions are flowing through the pipeline sequentially. In this configuration younger instructions can not pass the old ones. If an old instruction is stalled, all the younger instructions have to stall in their previous position in the pipeline and no new instructions are fetched in that cycle. An example is shown in figure 2.1 which depicts a classic 5 stages MIPS processor, IF(Instruction Fetch), ID(Decode), EX(Execute), MEM(Memory Access) and WB(Writeback). The result of the instruction is not available until it is written back into the register file in WB stage. In this case, I_2 is supposed to read the result of I_1 so it has to wait for I_1 get to the end of the pipeline. During this time it can not do anything except stay in the ID stage in which case a *bubble* is inserted into the pipeline. Because it's an in-order processor in which all the following instructions have to wait, significant performance degradation may occur if back to back dependencies are observed frequently in the program.

In fact it is unnecessary to block the pipeline in terms of the correct execution. We can continue the instruction flow as long as it reads the correct operands, disregarding the program order. When a block of code is fetched as shown in figure 2.2, not all the instructions are dependent to each other back to back. We can always find some

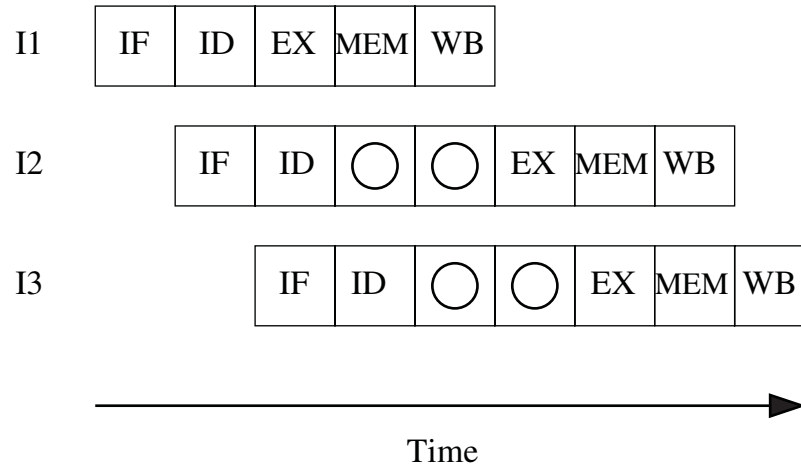


Figure 2.1: The bubbles in the pipeline

instructions which are able to execute during this period. In figure 2.2 $I1$ and $I3$ are executed in the 1st cycle since all the operands are available. $I2$ and $I4$ are scheduled to issue in the following cycle as both of them require the result from $I1$. When the constraint of sequential program order is removed, more instructions can run ahead depending on the data dependences. Substantial improvement has been achieved by processing more instructions in the same cycle which is measured as IPC (Instructions Per Cycle).

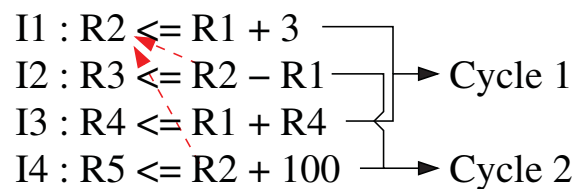


Figure 2.2: Instruction Level Parallelism

Tomasulo's algorithm [4] is the first algorithm devised to explore $ILLP$ through renaming. Instead of executing instructions in program order, Tomasulo's algorithm

will select those which have their operands ready and execute them first, disregarding their original order. Those instructions which do not have their operands ready wait and rescheduled into the execution units when their operands become available. Tomasulo's algorithm uses a *CDB*(Common Data Bus) to broadcast the ALU results to all the pending instructions. If the tag matches with the pending ones, the data will be stored locally. When both operands are collected, the pending instruction is signalled to execute. By doing this instructions are simultaneously executed by observing data dependencies, simultaneous execution in this manner is called instruction level parallelism, *ILP*. *ILP* processors aim to aggressively push as many ready instructions as possible to execute following the data dependencies, this is a common design employed in today's superscalar processors.

2.2 Register Renaming

In order to maximize the *ILP*, the processor has to somehow track down the true data dependencies as well as break down the false dependencies. A true dependence exists between two instructions if an instruction needs the output of a prior instruction. There are two different types of false dependencies, namely anti-dependence and output dependence. Anti-dependence is the reverse condition of true dependence. The register is read first then updated by the following instruction. An output dependence occurs when two or more instructions need to write to the same location. True or

false, all data dependencies must be observed for correct execution. However, false data dependencies can be eliminated by "renaming" the operand locations.

We can classify the register renaming techniques into two categories, implicit and explicit. The implicit renaming temporarily renames the registers using tags. The register is renamed to the output of its corresponding execution unit(EU), so all the following instructions can collect their operands when the EUs broadcast them. Meanwhile the results are captured by the register file if they are the most recent definitions of that value. As the result all the future instructions can read their operands from the register file directly.

The explicit renaming algorithm is renaming the logical registers to physical registers permanently during its life time. When the result is calculated it will write back to its assigned physical register, not the logical register. The operands are also read from the physical register file during the execution phase. In this manner, all the communications are unified via the physical register file.

I1 : R1(P0) <= 1
I2 : R2(P1) <= R1(P0) + 1
I3 : R1(P2) <= 2

Figure 2.3: Register Renaming

For example, in figure 2.3 *I1* and *I2* are truly dependent on each other, *I2* and *I3* are anti-dependence, *I1* and *I3* have output dependence. By renaming all these registers

to the physical registers $P0, P1$ and $P2$, we can execute $I3$ in any order, Since now it is not depend on $I1$ or $I2$.

Given the fact that all the instructions have distinct new names, only true data dependencies are left in the program which empower the superscalar processor to maximize the *ILP*.

2.3 Speculation

The processor may encounter exceptions like branch mispredictions, interrupts, traps, etc. To exploit the maximum amount of ILP, the superscalar processor continues to process incoming instructions until an exception is confirmed. During this time these instructions are in the speculative state. When an instruction is confirmed to be correct, we change it to so-called in-order state and permanently update the processor state. On the other hand, if it is confirmed as an exception, a recovery process is initiated to restore the processor back to its correct state. As a side effect the processor needs to discard some of the work finished previously belonging to the wrong state. Among all the exceptions, branch misprediction is the most common one. We discuss branch prediction and misprediction in detail in the following sections.

2.3.1 Branch Prediction

In addition to data dependencies described in section 2.2, we also have control dependencies in the program. When a branch is seen in the pipeline, the processor needs to know where to fetch the next instruction immediately or stall until the branch result is calculated. These dependencies are called control dependencies because all the future instructions are depend on the result of this branch. Modern processors incorporate branch predictors to predict the result of the branch so that the processors can keep fetching instruction.

Branch predictors rely on the past history to predict whether or not the branch will be taken. As a static branch is executed repeatedly at run time, it is possible to make an accurate prediction by observing the dynamic behavior of each branch. The most common technique is a history-based branch predictor. A history-based branch predictor predicts a given branch by asking for the history profile of this branch. Figure 2.4 demonstrates how it works. It's a FSM(Finite State Machine) with 4 different states. In each state the most significant bit represents the prediction, 1 means taken and 0 means not taken. Therefore state $\{11,10\}$ will predict branch taken and state $\{00,01\}$ will predict branch not taken. When the predictor is in state $\{11\}$, it will predict taken until two consecutive not taken are observed and vice versa for state $\{00\}$. The added hypothesis helps the predictor by changing its decision

slowly and ignoring rapid fluctuations.

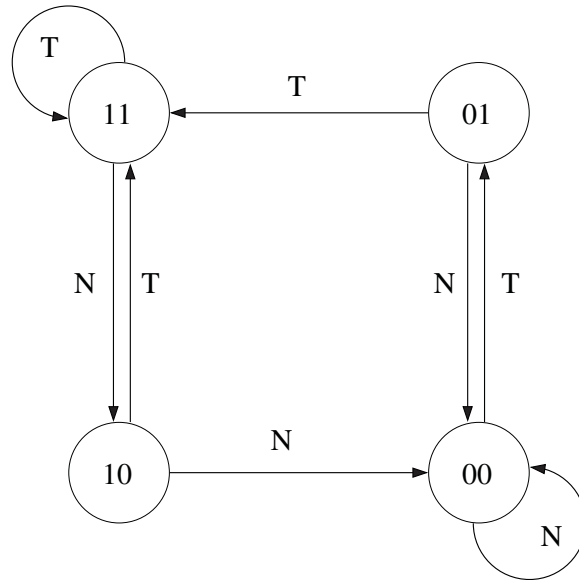


Figure 2.4: History-Based Branch Prediction

Knowing the direction of a given branch instruction is not sufficient. When it is predicted taken, branch predictor should also provide the correct target address used in the next cycle. Branch Target Buffer(BTB) is dedicated to supply the next PC address. The input of the BTB is the current PC address as shown in figure 2.5. For each entry in the BTB, it has BIA(Branch Instruction Address), BTA(Branch Target Address) and branch history fields. BIA and BTA are correlated, BIA is the current branch pc and BTA is the associative target PC if the branch is taken. Finding a given BIA in the BTB is called a BTB hit in this scenario. The corresponding line will be read to make a prediction. The branch history field contains 2 bits of the current FSM status which is used to predict the direction. BTA is fetched as the

target address if it predicts the branch is taken.

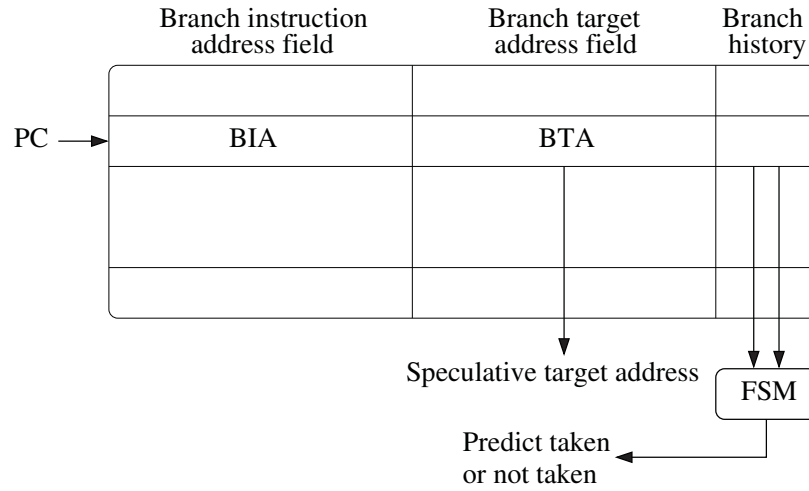


Figure 2.5: Branch Target Buffer

2.3.2 Reorder Buffer

As discussed before, when a branch is predicted, but not confirmed, all the following instructions fetched into the pipeline are in the speculative state. They are not permitted to change the processor state until the branch is confirmed to be correct. A Reorder Buffer (*ROB*) is used to keep speculative instructions until they are committed.

The block diagram of the *ROB* is illustrated in figure 2.6. The instructions are dispatched into the *ROB* following the program order, *I1* to *I10*. Then the instructions are executed based on their data dependencies irrespective of the program order. All

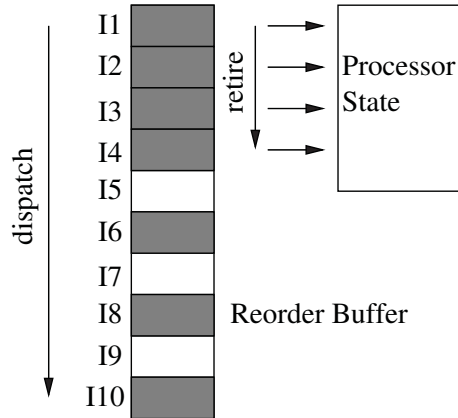


Figure 2.6: Reorder Buffer

the executed instructions are illustrated in shadow in the figure. On the other hand, the processor state is updated in the original program order. Only the completed instructions can update the processor state. Every time the instruction at the head of *ROB* updates the processor state, we call this instruction is retired from *ROB*. So *I1* will be the first one to retire, then *I2*, *I3* and *I4*. *I5* is uncompleted so it will not retire at this moment and all the following instructions can not retire either.

To an outside observer, the mechanism gives the illusion that instructions are being executed in program order, similar to an in-order processor. The only difference is that superscalar processor is running ahead by speculatively executing instructions and keeping the speculative states in the *ROB*. When the speculation is correct, a significant amount of performance is obtained compared to an in-order processor.

2.4 Recovery

When an exception like misprediction happens, recovery process is triggered. All the instructions before the exception should commit into the processor state. All the remaining instructions need to be evicted out, only the correct instructions can be fed into to pipeline and everything move on.

ROB plays an important roles here for recovery. All the instructions allocated in *ROB* obey the program order. Once a misprediction happens, *ROB* keeps retiring the instructions until it hits the mispredicting instruction. At this point all the instructions left in the *ROB* can be flushed away. The operation of the *ROB* therefore follows the FIFO(First Input First Output) scheme. Instructions flow into it in program order and retire out of it in the same order. Each time an exception happens with some instructions, it is easy to delete the incorrect ones as they are consecutive and always follow the mispredicted instruction.

Figure 2.7 demonstrates the recovery process. I_4 is a mispredicted branch. Instructions are retired to processor state starting with I_1 until I_4 is reached. All the following instructions from I_5 to I_{10} will be flushed out of the *ROB*. New instructions come from the correct path of I_4 are allocated into the *ROB* replacing I_5 towards the tail.

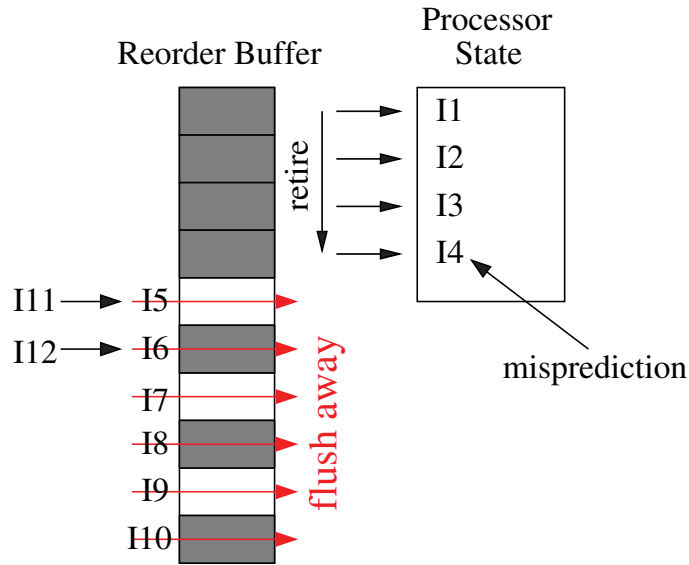


Figure 2.7: Misprediction Recovery

2.5 Summary

In this chapter, we described the basic concepts of superscalar processor, how to eliminate the false dependencies, how to execute speculatively and how to recover from exceptions. All these basic operations are critical for the reader to fully understand the proposed mechanism, Mower. With the conventional recovery algorithm in mind, We start to describe Mower in detail, an innovative misprediction recovery mechanism.

Chapter 3

Mower

3.1 General Overview

Mower is a novel technique which aims for complete overlapping of new instruction fetching with state restoration after a branch misprediction. This approach implies that the front-end will be fetching and processing instructions even if the back-end has not yet been drained of incorrect instructions. Mixing invalid instructions with valid instructions in the pipeline requires several concepts to be realized in the processor. In the front-end, renaming must be delayed for new valid instructions if they are referencing invalid mappings that have not yet been restored. Mower must therefore know if each RAT mapping is valid or not. In the back-end, Mower must release resources

occupied by invalid instructions precisely to restore the state without damaging the current state created by valid instructions. More specifically, the RAT mappings must be restored to their previous values and invalid instructions must be evicted or otherwise made ineffectual in reservation stations, execution units as well as the load/store queue.

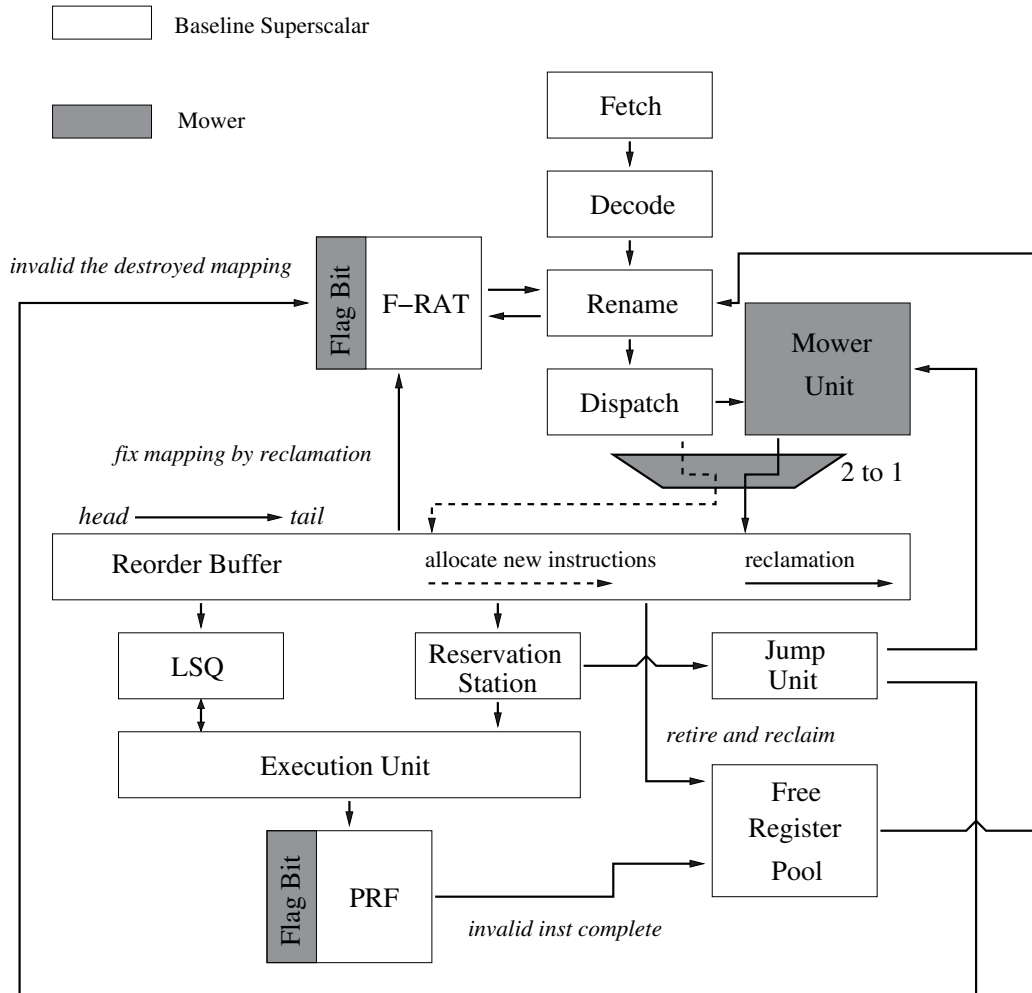


Figure 3.1: Mower Block Diagram

Figure 3.1 shows the overall additions required by Mower on a typical out-of-order superscalar pipeline. In the figure, darker colored sections show Mower components.

Overall, the pipeline stages for the traditional superscalar are kept the same. Branches and jumps are resolved in the jump unit (separated from execution units for clarity).

Examining the figure by following the pipeline flow order, the first change we notice is the additional bits which are attached to RAT. Mower uses these bits to identify whether a mapping has been damaged by a previous misprediction or not. All mapping bits start out valid. They are selectively invalidated when a branch instruction mispredicts. Note that there is no R-RAT under Mower, as the front RAT is incrementally updated to reflect the correct mapping after a misprediction.

Next, the *Mover Unit* (Figure 3.1, top-right) is responsible for maintaining and generating the mapping bits for RAT, walking over the ROB entries for rapid state recovery and releasing the resources kept by invalid instructions as well as making sure that invalid instructions still located in deep execution pipelines do not modify the state upon completion. Mower uses a single bit attached to each register in the Physical Register File (PRF) (Figure 3.1, bottom-left) to selectively disable writes for a particular register so that invalid instructions in deep pipelines can be allowed to complete without changing the state.

The operation of the Mower Unit involves its interaction with the jump unit, the reorder buffer and the renamer. The jump unit consults the Mower Unit, which provides a list of all mappings dependent on the mispredicting branch. These valid bits therefore allow the front-end to delay the renaming of those instructions which

reference to invalid mappings until the correct mapping is restored. In addition, during rename each jump or branch instruction is assigned a unique tag used by the Mower Unit. Since each branch is allocated a tag, the total number of in-flight branch instructions is equal to the number of available tags.

Although the Mower algorithm uses *walking* to reclaim resources, it is quite different. Mower walks sequentially from the mispredicted branch towards the ROB tail as opposed to existing algorithms, such as the walking technique which scans from the head of ROB towards the mispredicted branch and the history-buffer which starts from the tail of the buffer towards the mispredicted branch. Both of these techniques have to complete the entire scan before a usable, correct state is restored. In contrast, in case of Mower, due to its ability to identify damaged set of registers through a separate mechanism, scanning starts at the mispredicted branch and each register need to be corrected only once and at each step it is clearly known which part of the state is valid and which part of the state is still damaged. During Mower's walking process, each visited ROB entry undoes any changes it made to the processor structures. Visited invalid instructions are evicted from the reservation stations, load/store queues; release physical registers and restore their previous mappings in the RAT. Instructions which are still in deep pipelines are permitted to continue their execution as discussed above. When such an instruction completes, it frees the register instead of writing its result in that register. The walker then *mows* away all invalid instructions until the position of the ROB tail at the time of misprediction is

reached. Newly fetched instructions are stored in the freshly *mowed* area. Figure 3.2 illustrates resources released from the back-end by Mower.

3.2 Explicit Dependency Tracking and Out-of-order Branch Resolution

In order to accomplish full overlap of instruction fetching and state restoration as outlined, there are two problems which needs to be addressed. First, we need to keep track of branch dependency information for each instruction so that the required information is readily available as soon as a branch is mispredicted. In this respect, we are particularly interested to know which registers had been redefined after a mispredicted branch so that the front-end map table validity information can be immediately updated. Second, due to the out-of-order nature of branch completion, branch mispredictions may be detected in an out-of-order manner, which means additional mispredictions may come from those branches which follow as well as precede the branch which is in the process of state recovery. Obviously, mispredictions which come from later branches need to be ignored and mispredictions from prior branches should take over.

We address these issues by relying on two simple bit matrices, called the Mapping

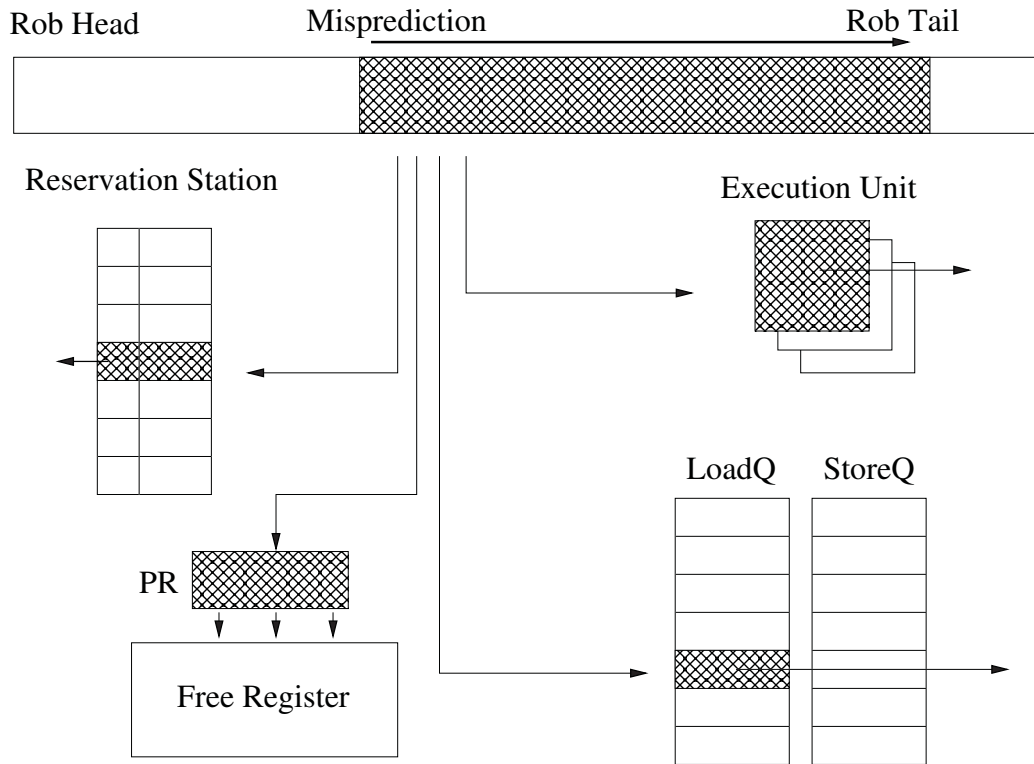


Figure 3.2: Mow all the invalid instructions through ROB, release all the relative resources.

Dependency Matrix (MDM) and the Branch Dependency Matrix (BDM). The operation of these matrices are identical; if a given resource (branch for BDM or register mapping for MDM) assigned to row i is dependent on a branch assigned to column j , the location $[i, j]$ is set to one. This organization permits easy extraction of the set of branches a particular resource is dependent on as well as the set of branches which are dependent on a particular resource.

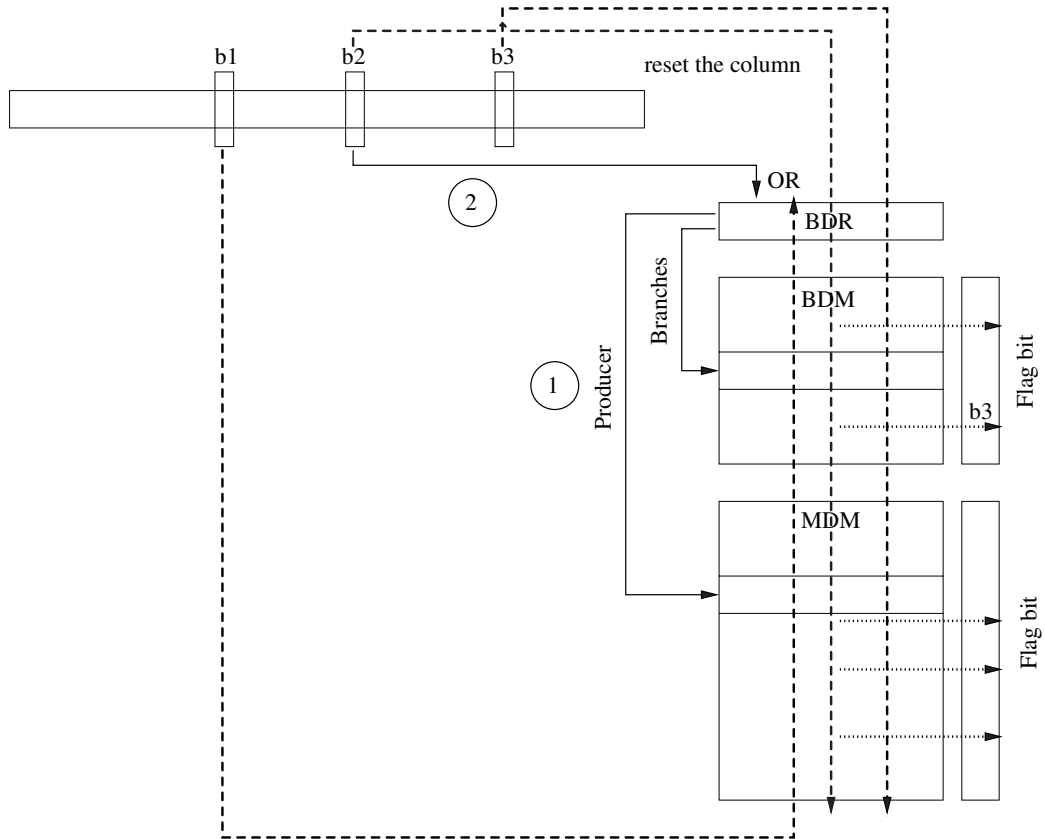


Figure 3.3: Explicit Dependency Tracking

3.3 Tracking Rename Map Validity

The organization of the MDM, which tracks rename map validity bits is shown together with the BDM which tracks in-flight branches in Figure 3.3. Each line in the MDM corresponds to a logical register number. Each column represents a dependency to a given branch tag. Therefore the MDM has a size of $L \times B$ where L is the number of logical registers and B is the number of in-flight branches. When Mower recovery starts, the column corresponding to the mispredicted branch is checked. If a given

logical register has a dependency on the mispredicted branch, i.e., it was redefined after the branch, it will have a 1 in the mispredicted branch id's column. Consequently, it is flagged as invalid. An invalid mapping can't be used to rename new instructions until corrected.

To populate the MDM, the Mower Unit must keep track of what branches a register should be dependent on at rename-time. A register called the Branch Dependency Register (BDR) of size B is maintained in the unit. Each time a branch is assigned a tag (decoded), the bit corresponding to the assigned tag is set in the register. BDR therefore provides an accurate representation of active dependencies at rename time. Each time an instruction is renamed, the BDR is inserted into the corresponding mapping line in the MDM, as shown in the figure.

When a branch instruction completes, there are two possible outcomes. If the branch has been correctly predicted, we should release the branch tag so new branches can use it. Since the branch is now complete, new instructions shouldn't be dependent on it either. Therefore we reset the corresponding bit in the BDR. Pre-existing physical registers should also clear their dependencies on the recently completed branch since the tag could now be re-used for a later branch instruction. Dependency clearing is accomplished by resetting the column corresponding to the branch tag in the MDM. In the case where the branch is mispredicted, the BDR and the MDM are handled in the same manner. Before clearing the MDM column, each register's dependency on

the mispredicted branch is checked. If it is dependent on the mispredicted branch, the register is marked as invalid.

The MDM covers invalidation of instructions that have a physical register mapping. Store and branch instructions do not have such a mapping. Handling invalid store instructions is simple. Stores do not affect processor state until they commit but they will not be allowed to write to memory since they are behind the mispredicted branch. All that needs to be done is to wait for the walker. Branch instructions are handled as discussed below.

3.4 Tracking Branch Dependencies

Branch exception ordering requires identification of which branch instructions are dependent on one another. Clearly, a branch instruction which was fetched after another branch instruction is dependent on that instruction. In order to track B in-flight branches, a $B \times B$ BDM matrix is used. When a branch is decoded, the BDR is copied into the BDM before being modified. When a branch is completed, the corresponding columns in the BDM, MDM, and BDR are cleared regardless of prediction accuracy. It is easy to see that the BDM is functionally identical to the MDM and provides a similar information with respect to branch instructions.

Let's now see an example of these matrices in action. Considering Figure 3.3, let us assume the branches $b1$, $b2$ and $b3$ are in program order, but they are resolved in the order $b2$, $b3$ and $b1$. $b2$ and $b3$ are mispredicted. When $b2$ resolves, the Mower Unit will check the BDM and MDM to discover branch instructions and mappings affected by $b2$'s misprediction, and supply these flags to external structures if necessary. Following that, the bit columns corresponding to $b2$'s tag from the BDM and MDM will be cleared, as well as the bit corresponding to the tag in the BDR. When $b3$ resolves, the BDM's flag register will inform the Mower Unit that it is an invalid branch. Thus, all columns corresponding to $b3$'s tag will still be cleared but no recovery will be triggered even though $b3$ is a mispredicted branch. Finally, when $b1$ resolves, the columns and the bit corresponding to $b1$'s tag will be cleared, but no additional action will be taken since $b1$ has been predicted correctly.

3.5 Recovery Timing and Details

Mower precisely recovers resources by walking over the ROB starting from the mispredicted instruction while allowing new instructions to be fetched and inserted into the ROB as soon as possible. Considering the corrections required in ROB, the correct location for the first instruction from the correct path in the ROB should be at the tail pointer, which should be reset to the mispredicting branch for recovery. In Mower, at the time of misprediction detection, these ROB slots are occupied by

invalid instructions and may contain data crucial to precise state recovery. Therefore Mower must walk over several ROB entries before these entries could be allocated.

Walking over the ROB requires entries to be read and written to the ROB. If Mower had to accomplish this in the same cycle as it is fetching instructions, we would be required to double the number of ports on the ROB. We clearly can't start fetching new instructions before doing at least some recovery since the slots in the ROB are occupied.

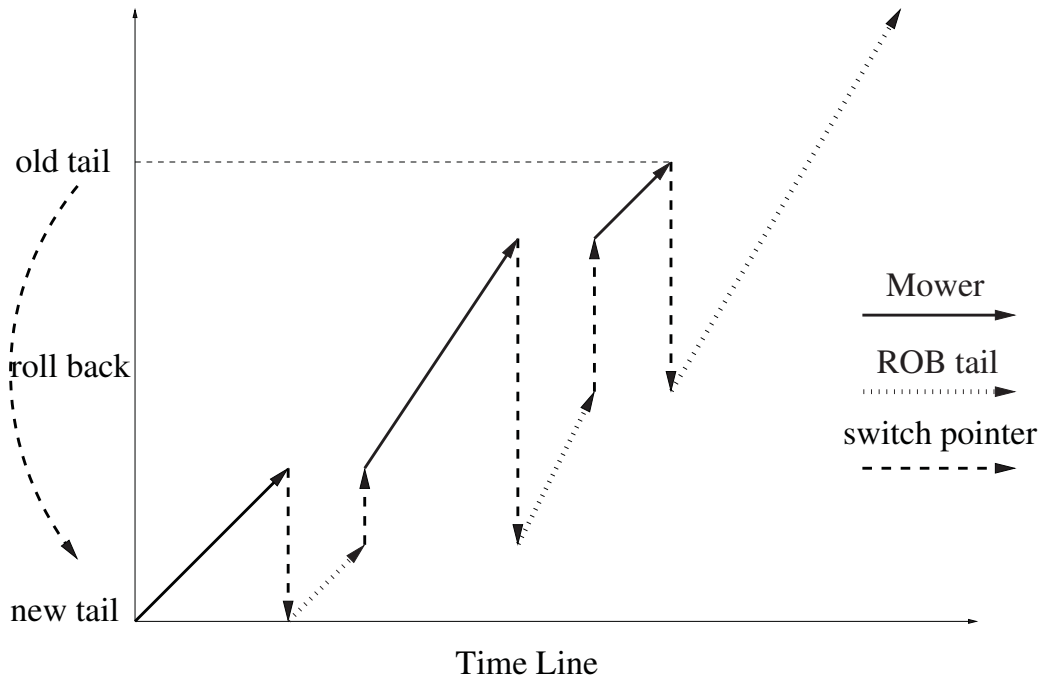


Figure 3.4: Interleaved reclamation and dispatching on ROB

We observe that it will take several cycles, based on pipeline depth, for newly fetched instructions to make it to the rename stage in the pipeline. Mower would like to

overlap this time with recovery time. We use an interleaved reclamation scheme where the ROB ports are split in half. The first half are used to walk over and recover some ROB entries. The next half is used to insert new instructions into ROB. Note that instructions are not inserted into the ROB until several cycles into the pipeline. During these cycles, Mower uses the full ROB port bandwidth available to recover resources. An example timeline can be seen in Figure 3.4. Using this interleaved technique, Mower can walk over the ROB as it is being populated by new instructions without adding much extra hardware. Most notably, Mower does not require extra ports for the ROB. While new instructions dependent on damaged mapping will need to be stalled at the rename stage, we observe that by the time instructions make it to the rename stage, most mappings will have been recovered.

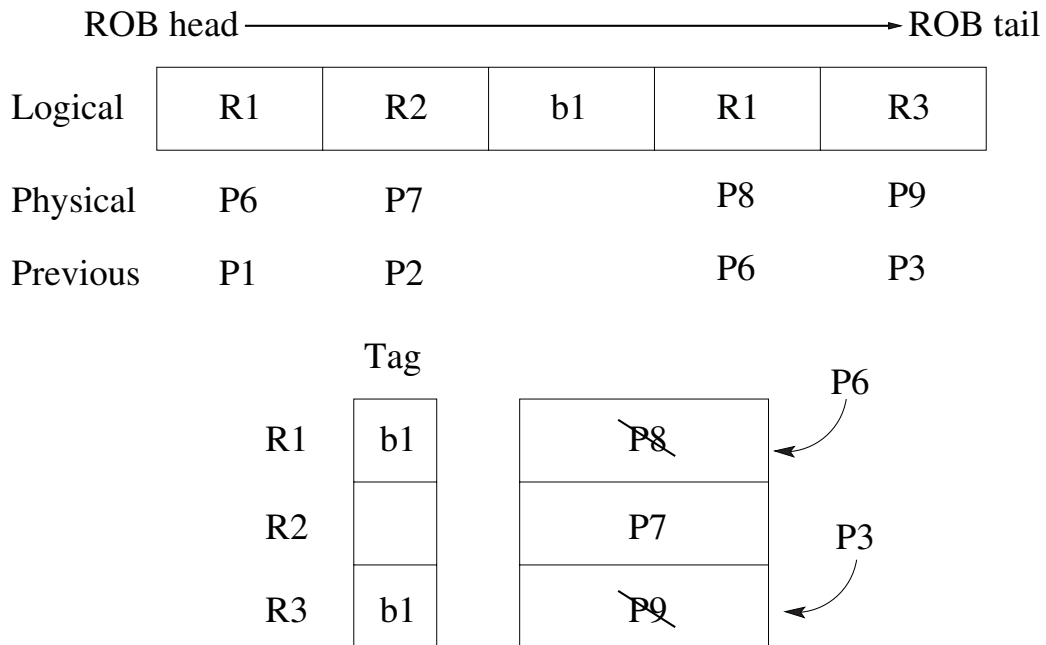


Figure 3.5: Fix the RAT by the reclamation

Figure 3.5 shows an example of RAT recovery in Mower. *b1* is a mispredicted branch. When *b1* is confirmed to be mispredicted, R1 and R3 mappings are marked as invalid. As Mower walks over these entries, the previous entry in the RAT is replaced with the previous mapping entry stored in the ROB. The register mapped to the invalid instruction is returned to the free pool (P8, in R1's case), and the register re-written into the table is marked as valid (P6, in R1's case). Note that walking in this direction, only one fix per RAT entry is applied. Once a RAT entry is fixed, further mappings to that logical register during walking is ignored and only physical register release is performed.

Chapter 4

Hardware Design

In this section we discuss the hardware design of the structures required to implement Mower. In Figure 3.1 the darker structures show the additions by Mower.

4.1 Renaming Branches

As discussed in Section 3.1, Mower assigns unique physical tags to each branch from a free pool of tags when a branch is decoded. If there are no free tags available (i.e. processor has reached the limit of in-flight branches), that branch is stalled until more tags become available. In-use branch tags are released when the related branch instruction is completed or reclaimed for invalid branch instructions during recovery.

Branch tags are encoded differently compared to regular rename identifiers. If there are B in-flight branches allowed in the processor, each tag will be B bits long. The tag contents will be the same as a $2^{\log(B)}$ -to- B decoder. For example, for $B = 4$, the branch tags will be 0001, 0010, 0100 and 1000.

4.2 Bit Matrices

To implement Mower as explained in Section 3.2, we need to build a $B \times B$ BDM as well as a $L \times B$ MDM. One possible hardware implementation of these matrices can be found in [5].

Both matrices require updates on certain events. When a relevant instruction (a branch for the BDM, a result producing instruction for the MDM) is decoded, a new line is inserted into the matrices. When a branch completes, it clears the corresponding columns it was renamed to in both matrices. Each line in the matrix indicates if it has a dependency on a certain branch.

To facilitate insertions into the matrices, a Branch Dependency Register is located near the matrices. This branch dependency register contains the logical OR of all current in flight branch tags. This operation constructs a B bit register which has the value 1 for branches that are currently in-flight and 0 for unassigned branches.

Each time a new entry is inserted into either matrix at the correct position, the BDR is copied into the line instead to simplify acquiring new entries.

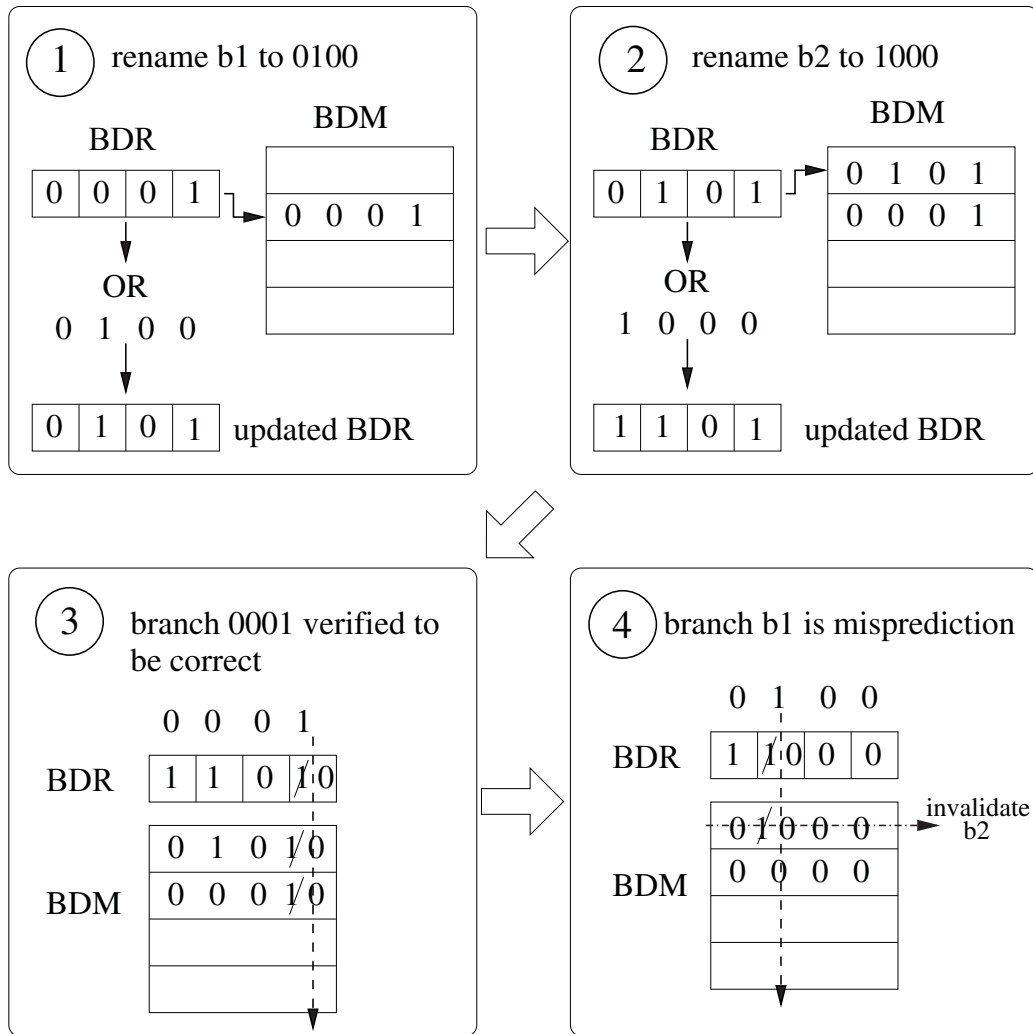


Figure 4.1: Branch Dependency Matrix

Figure 4.1 illustrates all the possible operations on a BDM in a processor which only allows 4 in-flight branches. In the figure ① *b1* is renamed to 0100 and the current BDR value is copied to the 2nd row of the BDM. The BDR is updated to 0101 by setting the

2nd most significant bit. Then in figure ② we rename another branch $b2$ to 1000, the BDR is copied to the 1st row of BDM and we can see that the 2nd most significant bit is set which means $b2$ is dependent on $b1$. Both $b1$ and $b2$ are dependent on branch 0001. In figure ③, branch 0001 is resolved and has been correctly predicted. Thus we reset its corresponding column in both the BDR and the BDM. All future branches will be independent from this cleared branch until a new branch is renamed to that value. Finally in figure ④ $b1$ is resolved and is mispredicted. We still reset its column as before. However, we also invalidate all younger branches in flight by invalidating all the entries which have a dependency on the physical tag of $b1$ (which is 0100 in this example). Updates to the MDM are handled similarly.

4.3 Fixing The RAT

As with the BDM, the MDM is updated by copying the BDR into the line which corresponds to the register that was just renamed. On a misprediction on a given branch b , entries which have a 1 in column b correspond to damaged mapping entries.

Figure 4.2 demonstrates the procedure to fix the alias table as well as correctly updating the MDM.

The figure on the left shows I2 being renamed, where I1 is an instruction older than

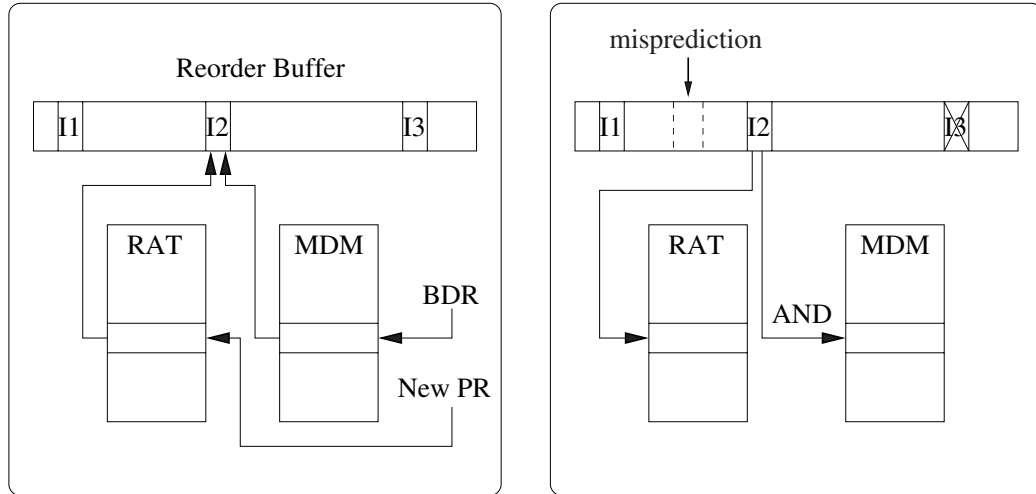


Figure 4.2: Recovering RAT from MDM

I2 and I3 is younger than I2. All three instructions write to the same logical register. When I2 is renamed, its previous mapping and MDM information are stored in the ROB. At this point, we update both the RAT and the MDM with current information. The figure on the left shows the state of the structures after a misprediction is detected between I1 and I2. When walking back over the ROB, if the RAT entry corresponding to the instruction's logical destination is damaged, the previous mapping stored in ROB is restored and the RAT entry is marked clean. In the example, we can see that when walking over I2, we will correct the RAT to I1's mapping (since I2 stored that as it was being renamed), but not when we walk over I3 since the mapping table entry is marked as clean at that time. In the MDM, we unfortunately can't simply restore the entry from the ROB. A branch the mapping was depended on may have completed, making the archived entry invalid. In essence, we want the intersection of the archived branch dependencies and the current branch dependencies. We can achieve this by ANDing the archived entry with the entry currently in the MDM. This

concludes everything necessary to restore the RAT mappings and maintain MDM state while walking back over the ROB after a misprediction.

Chapter 5

Evaluation

5.1 Methodology

We use MIPS I ISA to evaluate Mower. Mower is modeled in ADL [6], an architecture description language used to generate cycle-accurate simulators. We incorporated power models and estimated the power consumption for both Mower and the baseline. Power values have been obtained by adapting Wattch[7] to the ADL simulator framework. The power results have been validated against the McPAT[8] tool tested with a very similar superscalar pipeline to ensure correctness. Additionally, the non-ideal clock gating option is enabled in Wattch (causes only 10% power use when a particular port is not in use) as well as the dynamic activity factors (only precharges memory

lines if they previously contained 0). The BDM and MDM matrices are implemented as CAM-like structures in Wattch for power evaluation purposes.

Table 5.1
The configuration of the simulation

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/sutractor	7 cycles
Float multiplier	7 cycles
Load Speculation	Store Set
Branch Predictor	4KB GShare 14 bits global branch history shift register

We use the Spec2006 benchmark suite for performance and power evaluations. The suite was compiled using gcc version 4.3 with the highest optimization setting (-O3). Binutils 2.22 was used as the software environment. uClibC 0.9.33 was used to link the benchmarks and was simulated along with benchmark code. O/S kernel was not simulated. The ref inputs for the given benchmarks were ran for 500 million instructions to warm up the branch predictor and the cache, and an additional 1 billion instructions were simulated to gather the data.

The baseline implementation is an 8-wide conventional superscalar processor. A

GShare branch predictor is used in the front end.

Aside from the baseline superscalar implementation, we also compared Mower with EMR (Eager Misprediction Recovery)[9] and CPR (Checkpoint Recovery) [10] algorithms. For sanity checking purposes, a perfect recovery technique is also tested against Mower.

The tested EMR implementation is implemented with 4 in-flight checkpoints which are used per misprediction recovery. Invalid instructions are assumed to be eliminated immediately once detected. The CPR utilizes 8 checkpoints. Each decoded branch will occupy one checkpoint until all instructions dependent on that branch complete. Checkpoints are released out-of-order. 8 checkpoints were chosen since it is expensive to implement a large number of checkpoints. The perfect recovery technique is an ROB based technique which recovers from a misprediction in the same cycle with no delays and resources being occupied.

5.2 Performance Results

We first explored the Mower design space to figure out what the in-flight branch limit should be. Figure 5.1 shows the average IPC value of Mower with different in-flight branch numbers. Even with only 4 in-flight branches, Mower can extract 98.2% of

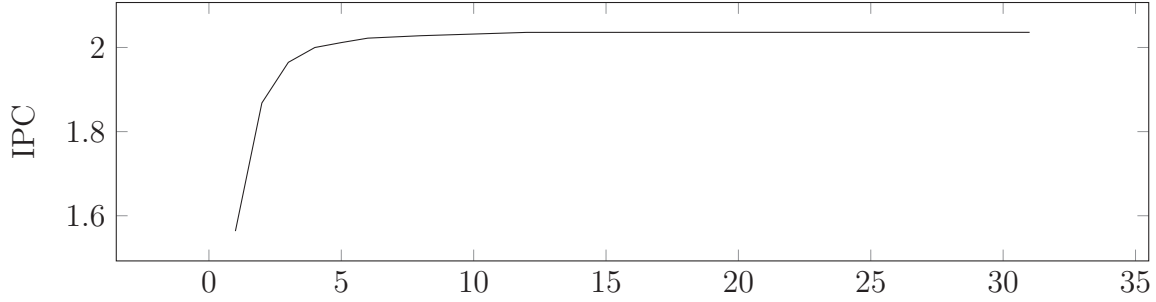


Figure 5.1: IPC vs In-flight Branches

performance available to a processor with access to unlimited in-flight branches. In our experiments, we set the in-flight branch number to 12 which puts our Mower implementation at 99.99% performance of a Mower processor with unlimited in-flight branches.

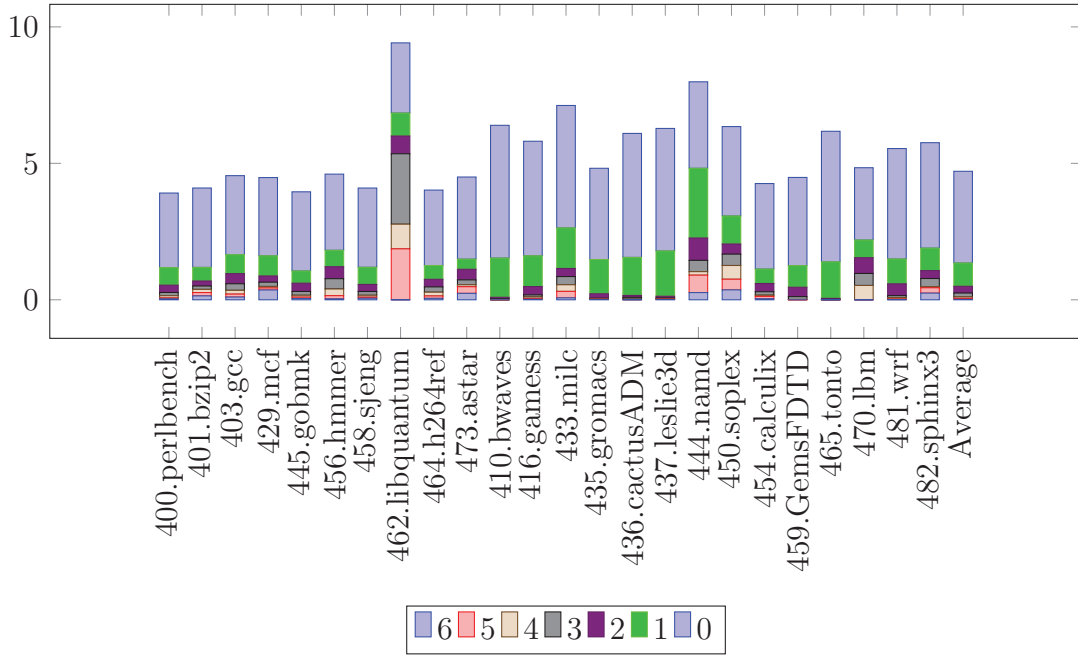


Figure 5.2: The Damaged F-RAT through the recovery process

Next, we show the number of damaged mappings throughout the recovery procedure.

In figure 5.2, the values identified by 0 show the number of damaged mappings when misprediction is first indicated. Index 1 shows damaged mappings after one cycle of recovery. Index 2 shows damaged mappings after two cycles of recovery and so on. Our data shows that after 6 cycles, Mower recovers nearly as much as the perfect recovery baseline, and no delays will be incurred by instructions arriving at the rename stage at this point.

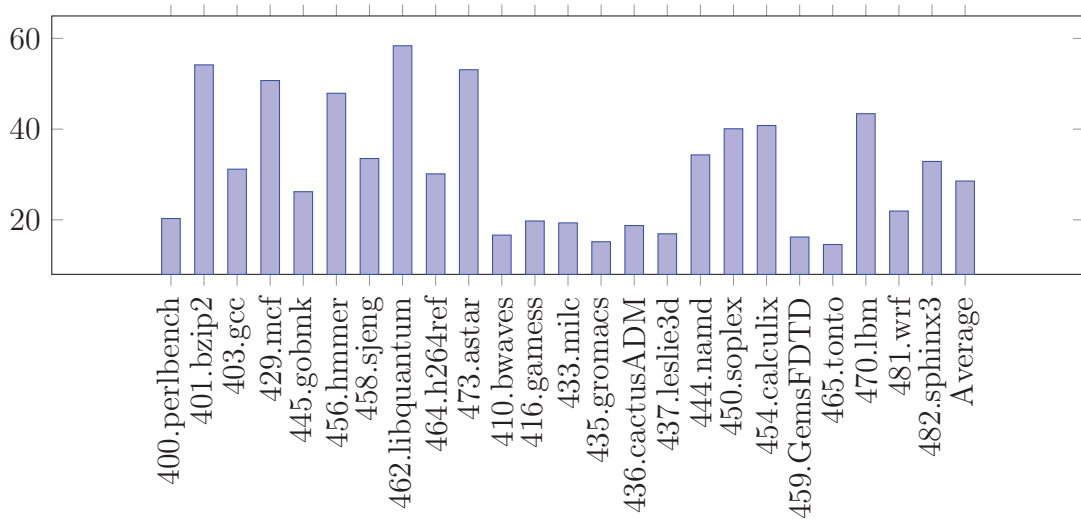


Figure 5.3: The average number of invalid instructions left in the pipeline when misprediction is detected

We also measure the number of invalid instructions left in the processor when a misprediction occurs. This data is presented in Figure 5.3. Looking at Figure 5.3 and Figure 5.2 together, we see invalid instructions almost up to 60 after a misprediction yet the number of invalid mappings never exceeds 8. This is partly due to store and branch instructions within the invalid instructions which do not have mappings, and partly due to register overrides. Having very few invalid mappings makes the recovery

job easier.

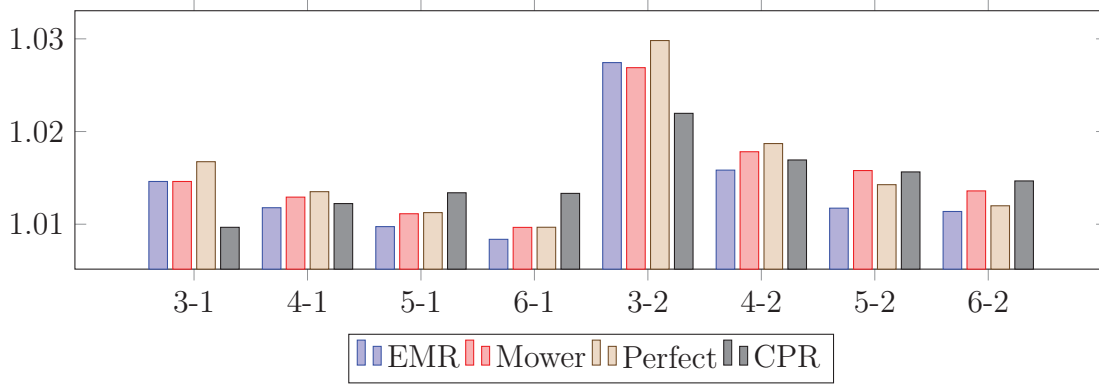


Figure 5.4: Spec2006 Integer Speedup

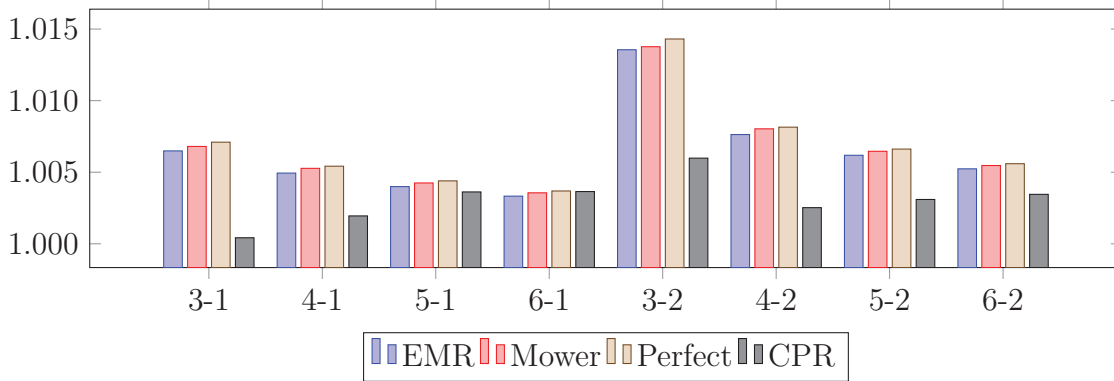


Figure 5.5: Spec2006 Float Speedup

In Figure 5.4 and 5.5 we show the geometric mean of the speed up factor over the baseline system in all Spec2006 Integer and Float benchmarks, respectively. Different shades of color indicate different configurations. The first number in the configuration is the number of front-end pipeline stages (up to where an instruction would be inserted into the reservation stations) and the second number indicates the number of stages before retirement but after execution. These graphs show that having a shorter front-end pipeline is better for faster recovery. This is expected since a shorter pipeline fill delay will benefit all techniques implemented here. Another observation is

that performance increase jumps up when retiring an instruction takes more than one cycle after completion. The baseline implementation has to wait for the misprediction to retire, but Mower moves ahead with the fetching during the same time frame. Most notably, we can see that Mower performs very close to perfect misprediction recovery in every case.

Table 5.2
Misprediction Rates for Spec2006INT

Spec2006	Mispred. Per 10k	Mower Speedup
401.bzip2	30.0	1%
403.gcc	66.9	3%
429.mcf	59.9	2%
445.gobmk	89.4	5%
458.sjeng	52.4	4%
462.libquantum	11.9	0%
464.h264ref	31.5	1%
473.astar	88.8	4%

Looking at Table 5.2 lets us examine the speedups provided by Mower. In general, we can see a correlation between the misprediction rate and performance increase in the integer benchmarks. There are some outliers, for instance the 429.mcf benchmark shows a smaller speedup while having a greater misprediction rate than 458.sjeng. A more telling example is the comparison of the gcc benchmarks for Spec2000 and Spec2006 suites. 176.gcc from Spec2000 shows 132.9 mispredictions per 10k instructions, which is roughly double the number of mispredictions from the Spec2006 version. The performance is similarly impacted: 176.gcc shows close to 6% performance increase using the same parameters. Since Mower is a technique that targets branch misprediction recovery delays, if many mispredictions do not occur in the base case,

not much performance will be extracted since there are no delays in the first place.

5.3 Power Estimation

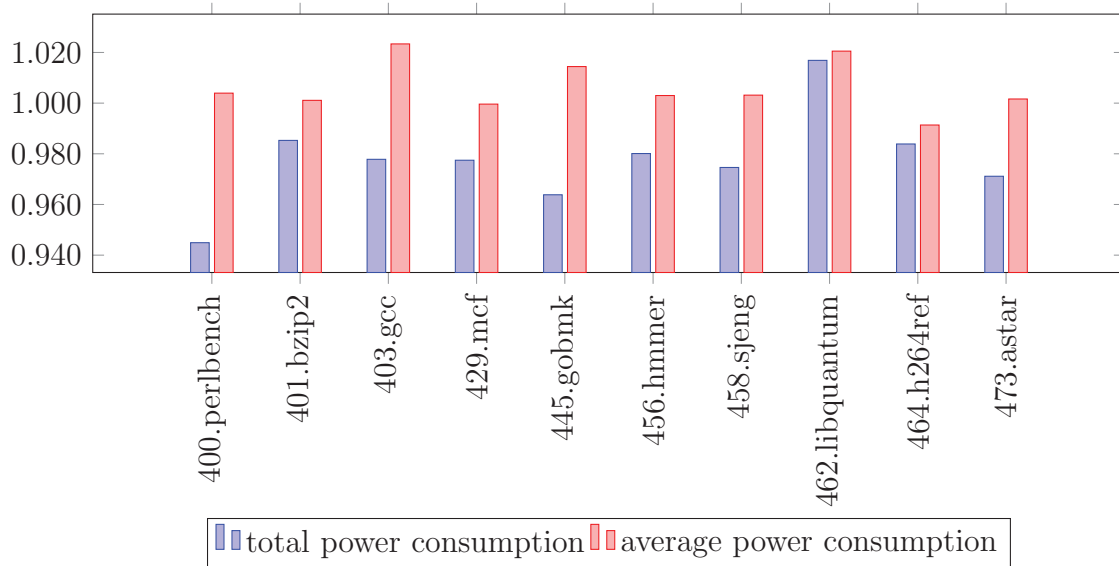


Figure 5.6: Power Evaluation Spec2006 Integer

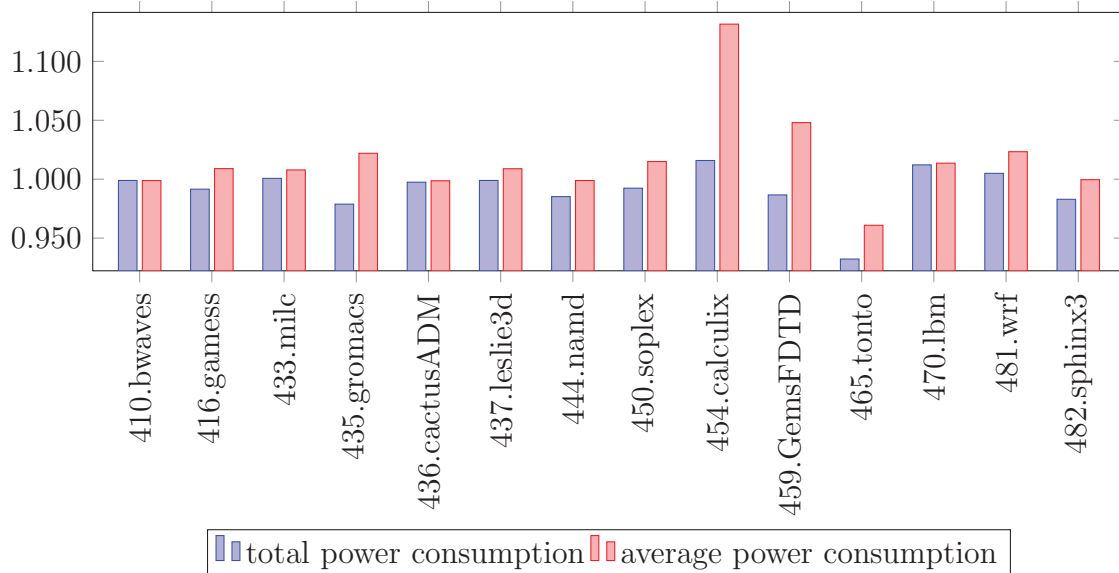


Figure 5.7: Power Evaluation Spec2006 Float

Table 5.3
Power distribution in Spec2006

	Mower				Baseline	
	total power	average power	average BDM	average MDM	total power	average power
400.perlbench	39414282.82	23.0847	0.0178	0.3418	41711982.94	22.9941
401.bzip2	19912545881	36.5107	0.0099	0.4674	20209901773	36.4707
403.gcc	20577678552	31.064	0.0177	0.4205	21044400878	30.3547
429.mcf	24399551282	23.9638	0.0125	0.2964	24962176205	23.9732
445.gobmk	21246144709	30.1403	0.0116	0.4082	22043497993	29.7116
456.hmmer	20169386181	52.1055	0.0108	0.7009	20578874932	51.9498
458.sjeng	20952348479	45.881	0.0243	0.5581	21498326343	45.7369
462.libquantum	18367126463	56.8546	0.014	0.7518	18062344589	55.7114
464.h264ref	19009792993	44.2755	0.015	0.5757	19321361564	44.6614
473.astar	25288238076	47.5951	0.0157	0.6439	26039881396	47.5177
410.bwaves	23545019645	49.1315	0.0012	0.5251	23570556909	49.1887
416.gamess	20683416431	53.3552	0.0097	0.6136	20859995597	52.8809
433.milc	22033327175	40.0966	0.004	0.4567	22017599284	39.7845
435.gromacs	20251109581	41.9003	0.0329	0.5177	20689221337	40.9992
436.cactusADM	24119700357	28.5811	0.0018	0.3049	24181298377	28.62
437.leslie3d	22490620504	44.733	0.0043	0.5086	22515068357	44.3398
444.namd	19011620317	47.6229	0.0348	0.5776	19297024881	47.6767
450.soplex	3531370375	42.2901	0.0134	0.5432	3558493095	41.6645
454.calculix	10508899197	51.8637	0.0172	0.6256	10344855160	45.8328
459.GemsFDTD	18450836352	42.359	0.0428	0.5504	18700506380	40.4183
465.tonto	21579146650	45.8823	0.022	0.605	23147948645	47.7491
470.lbm	21052658006	34.911	0.003	0.3709	20799850538	34.4426
481.wrf	23438584264	56.3007	0.015	0.5877	23322660370	55.017
482.sphinx3	20007152673	45.5453	0.0387	0.5723	20354623969	45.564

In Figure 5.6, we illustrate the power consumption of Mower for all integer benchmarks. The data is normalized to the baseline superscalar. The lighter bar is the average power consumption and the darker bar is the total power consumption for a given benchmark. Average power consumption is the energy dissipated per second and the total power consumption is the energy dissipated by the whole benchmark. Mower saves close to 2% of total power in most integer benchmarks (462.libquantum

is an exception). All power savings come from early reclamation. Looking at performance results, we can also see an association between performance increase and power savings. Reclaiming more invalid instructions for mispredictions translates to power savings.

Unfortunately, Figure 5.7 shows a poorer picture for total power dissipation. This is due to the smaller number of branches in floating point benchmarks. Fewer branches mean even fewer mispredictions, and fewer mispredictions mean the additional structures employed by Mower are being used without providing a major advantage. 465.tonto has more branches than any other floating point benchmark, explaining the power difference.

It should be noted that the average power consumption is always worse for Mower since additional structures are involved. However, since Mower completes workloads faster, total power per workload is lower when Mower structures are being taken advantage of.

Detailed power numbers can be seen in Table 5.3. MDM dominates the additional power use of Mower due to its large size compared to the BDM. Additionally, all instructions which produce a value update the MDM while this is only true for branch instructions in the case of BDM. A possible solution for this issue would be to bank the MDM as described in [11].

In order to evaluate the power efficiency of a given technique an appropriate metric is needed. Traditionally, energy per instruction metric was widely used to measure the power efficiency. This metric is proportional to CV^2 where C is the capacitance of the transistor and the V is the operational voltage of the processor. A processor can be designed for power efficiency by using smaller transistors or decreasing the voltage. However this technique accomplishes energy efficiency at the expense of performance. Therefore EDP[12] makes more sense to evaluate the power efficiency. Generally speaking, EDP compares the energy consumption under the same performance. For example, if the processor saves energy with the same execution time or consumes the same energy but reduces the execution time, both of them are regarded as power efficient in terms of EDP measurement.

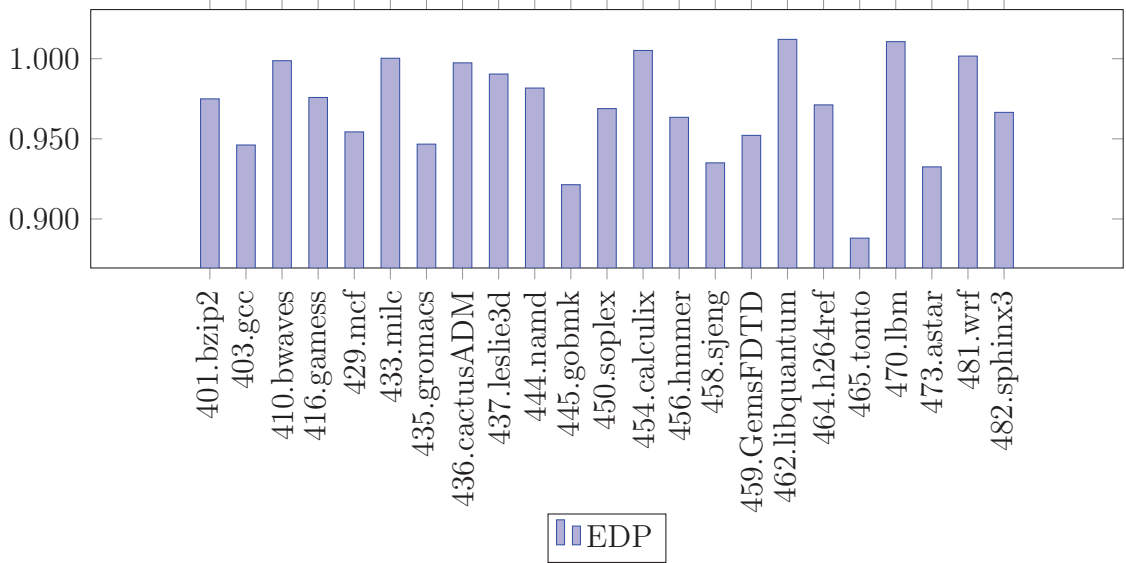


Figure 5.8: EDP normalized to baseline configuration

Figure 5.8 displays the EDP of Mower normalized to the baseline configuration. It is not surprising that a benchmark which saves more energy will be more power efficient since it reduces the total execution time at the same time. Mower helps to reclaim the invalid instructions early by which more energy is saved and misprediction penalty is reduced. In this manner, Mower is more power efficient in most benchmarks except 462.libquantum and 470.lbm. Even in these two benchmarks Mower is only 1% worse than the baseline. In other benchmarks, Mower performs much better than our baseline configuration in power efficiency. For example in 465.tonto Mower is 13% more power efficient.

Chapter 6

Related Work

Many misprediction recovery techniques have been proposed over the years. We will show an overview of existing work in this field as well as discuss a few techniques that are orthogonal to Mower.

Smith et. al. [2] show recovery mechanisms using ROB in their study to set the stage for the majority of non-checkpointing based recovery techniques. Hwu et. al. [10] propose a checkpointed recovery system to repair processor state. A checkpoint of the processor state is taken at every branch and recovery happens from these checkpoints on a misprediction. This is the technique Mower was tested against in this paper. Variations and improvements using checkpoints have been proposed by other authors. Akkary et. al. [3] propose Checkpoint Processing and Recovery (CPR). The major

relevant contribution of CPR is selective checkpointing (only low confidence branches are checkpointed). Both the original checkpointing algorithm as well as CPR do not use an ROB.

Zhou et. al. [9] proposed Eager Misprediction Recovery (EMR), which is the second technique we compared with Mower. EMR creates checkpoints of the front-end mapping tables. However, when a misprediction occurs, instructions from the new path are immediately fetched and decoded. Misprediction recovery allows each new instruction to reach up to the reservation station but incorrect front-end mappings are marked as damaged. Unlike Mower, which restores the correct mappings before instructions are renamed, EMR instead restores the correct values of damaged mappings into the damaged physical registers by utilizing functional units.

Amit et. al. [13] proposes Selective Branch Recovery which is another technique which does not necessarily discard all instructions from the incorrect path immediately. SBR attempts to detect if the mispredicted path will converge back on to the correct path. If such a convergence is detected, some results from the incorrect path may still be useful.

Akl et. al. [14] propose Turbo-ROB, which is an ROB based recovery design that only allocates ROB entries at certain points determined to be repair points. Turbo-ROB allows for the illusion of a larger ROB while using fewer entries at the cost of not being able to restore the state to any instruction. Latorre et. al. [15] provide a

similar improvement with CROB but handle misprediction recovery by copying over a checkpoint of relevant processor structures per branch.

Hilton et. al. [16] propose a checkpointing scheme with an ROB called CPROB. CPROB switches to an ROB based recovery scheme if mispredictions are frequent and uses a smaller window. If predictions are going well, the window size is dynamically expanded and the misprediction recovery scheme switches to using checkpoints. Golander et. al. [17] propose a similar scheme without an ROB utilizing checkpointing. They take checkpoints more frequently when mispredictions are frequent, and vice versa.

BranchTap [18] by Akl et. al. is a technique that allows a processor to stop fetching instructions when a threshold of low confidence branches are in the processor. This threshold is dynamically calculated during execution. BranchTap can work with or without checkpointing available. BranchTap is orthogonal to Mower - not fetching many incorrect instructions would speed up Mower's recovery.

Another technique orthogonal to Mower is WPE [19] by Armstrong et. al. WPE detects so called Wrong Path Events which include unusual or illegal behaviour such as a NULL pointer access or a divide by zero. These events are then used to speculate that we are on the wrong path of execution. WPE would allow Mower to start recovery early.

Several techniques have been proposed that fetch instructions from both paths on low confidence branches. Such techniques target the pipeline fill delay, which is not targeted by Mower. Use of these techniques alongside Mower could further improve Mower’s performance. Disjoint Eager Execution (DEE) by Uht et. al. [20] is one of the earlier examples. Code paths that are likely to be executed are taken, meaning both paths on a branch that is likely to mispredict will be executed. DEE is implemented in a custom architecture called LEVO. Heil et. al. [21] propose Selective Dual Path Execution (SPDE) which selectively executes two (and only two) branch paths at the same time based on prediction confidence. SPDE is implemented in a regular superscalar processor with the addition of confidence predictors and an additional register alias table. Another work in this area is Selective Eager Execution (SEE) by Klauser et. al. [22] SEE is similar to SPDE but uses a novel instruction tagging and register renaming to avoid including a second RAT. The architecture is called PolyPath. Wallace et. al. [23] apply a similar technique to simultaneous multi-threading processors to allow the SMT processor to execute two paths from the same process called Threaded Multiple Path Execution (TMPE). TMPE introduces the Mapping Synchronization Bus which allows a new thread to be forked when a low confidence branch is encountered. TMPE also deals with resource use between the multiple paths. Dual Path Instruction Processing (DPIP) is proposed by Aragon et. al. [24]. DPIP specifically targets the window fill penalty where it fetches, decodes and renames instructions from the alternate path but does not execute them. An

extension to DPIP gets pre-fetched instructions in an estimated schedule for rapid issue into execution units.

Chapter 7

Conclusion

7.1 Contribution of the thesis

Mower is an innovative mechanism to provide better branch misprediction recovery for superscalar processors. we believe there are several important contributions made by the Mower algorithm. These are 1) ability to dynamically release the stale resources while keeping correct dependencies; 2) Fixing the frontend RAT at the same time in order to make an expedited channel to the backend. 3) sharing ROB access ports during reclaim process to maximize the utilization.

In most recovery algorithms, the recovery process is limited by hardware resources which are predetermined. We believe Mower is the first approach to do it in a dynamic

way so that there are no predetermined limitations in the sense that it can deal with any number of mispredictions which can happen. Mower is also a low-cost solution. Added structures are small in size, providing reasonable power consumption while improving performance. Our evaluation indicates that Mower eliminates nearly all of state restoration penalty component of branch mispredictions. We believe the proposed technique is an effective technique, particularly for integer programs which contain a large number of high penalty branch mispredictions.

7.2 Future work

Mower is dedicated to dynamically fixing the pipeline before the *ROB* is drowned when misprediction happens. It accelerates the renaming procedure by fixing the *RAT* on-line. Meanwhile it will nullify the stale instructions through the *ROB* by the shared port. One of the biggest issues here is the large number of invalid instructions left in the pipeline, for example, figure 5.3. It takes time and wastes energy to do so.

Chou et. al. [25] devise a mechanism to detect CI(Control Independence) instructions. Generally speaking, when a program diverges at the branch, most of the time it will converge at a later point. All the instructions after the convergence point are control independent of the branch. The CI detection technique can be integrated into Mower seamlessly to minimize the trashed instructions when we clear the *ROB*. Not all the

instructions after the misprediction are useless. If we can find a way to identify the useful ones, it will further mitigate the penalty of misprediction.

References

- [1] Mike Johnson, *Superscalar Microprocessor Design*, Prentice Hall, 1991.
- [2] James E. Smith and Andrew R. Pleszkun, “Implementing precise interrupts in pipelined processors”, *Computers, IEEE Transactions on*, vol. 37, no. 5, pp. 562–573, 1988.
- [3] Haitham Akkary, Ravi Rajwar, and Srikanth T Srinivasan, “Checkpoint processing and recovery: Towards scalable large instruction window processors”, in *MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2003, pp. 423–434.
- [4] R.M. Tomasulo, “An efficient algorithm for exploiting multiple arithmetic units”, *IBM Journal of Research and Development*, vol. 11, no. 1, pp. 25–33, Jan 1967.
- [5] Peter G. Sassone, Jeff Rupley, II, Edward Brekelbaum, Gabriel H. Loh, and Bryan Black, “Matrix scheduler reloaded”, in *Proceedings of the 34th Annual*

- International Symposium on Computer Architecture*, New York, NY, USA, 2007, ISCA '07, pp. 335–346, ACM.
- [6] Soner Önder and Rajiv Gupta, “Automatic generation of microarchitecture simulators”, in *IEEE International Conference on Computer Languages*, Chicago, May 1998, pp. 80–89.
- [7] David Brooks, Vivek Tiwari, and Margaret Martonosi, “Wattch: A framework for architectural-level power analysis and optimizations”, in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, New York, NY, USA, 2000, ISCA '00, pp. 83–94, ACM.
- [8] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi, “Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures”, in *42nd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2009, pp. 469–480.
- [9] Peng Zhou, Soner Önder, and Steve Carr, “Fast branch misprediction recovery in out-of-order superscalar processors”, in *Proceedings of the 19th Annual International Conference on Supercomputing*, New York, NY, USA, 2005, ICS '05, pp. 41–50, ACM.

- [10] Wen-mei W. Hwu and Yale N Patt, “Checkpoint repair for out-of-order execution machines”, in *Proceedings of the 14th annual international symposium on Computer architecture*. ACM, 1987, pp. 18–26.
- [11] J.-L. Cruz, A. Gonzalez, M. Valero, and N.P. Topham, “Multiple-banked register file architectures”, in *Proceedings of the 27th International Symposium on Computer Architecture*, June 2000, pp. 316–325.
- [12] R. Gonzalez and M. Horowitz, “Energy dissipation in general purpose microprocessors”, *Solid-State Circuits, IEEE Journal of*, vol. 31, no. 9, pp. 1277–1284, Sep 1996.
- [13] Amit Gandhi, H. Akkary, and S.T. Srinivasan, “Reducing branch misprediction penalty via selective branch recovery”, in *10th International Symposium on High Performance Computer Architecture*, Feb 2004, pp. 254–264.
- [14] Patrick Akl and Andreas Moshovos, “Turbo-rob: a low cost checkpoint/restore accelerator”, in *High Performance Embedded Architectures and Compilers*, pp. 258–272. Springer, 2008.
- [15] Fernando Latorre, Grigorios Magklis, Jose González, Pedro Chaparro, and Antonio González, “Crob: implementing a large instruction window through compression”, in *Transactions on high-performance embedded architectures and compilers III*, pp. 115–134. Springer, 2011.

- [16] Andrew Hilton, Neeraj Eswaran, and Amir Roth, “Cprob: Checkpoint processing with opportunistic minimal recovery”, in *18th International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 2009, pp. 159–168.
- [17] Amit Golander and Shlomo Weiss, “Checkpoint allocation and release”, *ACM Trans. Archit. Code Optim.*, vol. 6, no. 3, pp. 10:1–10:27, Oct. 2009.
- [18] Patrick Akl and Andreas Moshovos, “Branchtap: Improving performance with very few checkpoints through adaptive speculation control”, in *Proceedings of the 20th Annual International Conference on Supercomputing*, New York, NY, USA, 2006, ICS '06, pp. 36–45, ACM.
- [19] David N Armstrong, Hyesoon Kim, Onur Mutlu, and Yale N Patt, “Wrong path events: Exploiting unusual and illegal program behavior for early misprediction detection and recovery”, in *37th International Symposium on Microarchitecture*. IEEE, 2004, pp. 119–128.
- [20] Augustus K Uht, Vijay Sindagi, and Kelley Hall, “Disjoint eager execution: An optimal form of speculative execution”, in *Proceedings of the 28th annual international symposium on Microarchitecture*. IEEE Computer Society Press, 1995, pp. 313–325.
- [21] Timothy H Heil and James E Smith, “Selective dual path execution”, Tech. Rep., Technical report, University of Wisconsin-Madison, 1996.

- [22] Artur Klauser, Abhijit Paithankar, and Dirk Grunwald, “Selective eager execution on the polypath architecture”, in *ACM SIGARCH Computer Architecture News*. IEEE Computer Society, 1998, vol. 26, pp. 250–259.
- [23] Steven Wallace, Brad Calder, and Dean M Tullsen, “Threaded multiple path execution”, in *ACM SIGARCH Computer Architecture News*. IEEE Computer Society, 1998, vol. 26, pp. 238–249.
- [24] Juan L Aragón, José González, Antonio González, and James E Smith, “Dual path instruction processing”, in *Proceedings of the 16th international conference on Supercomputing*. ACM, 2002, pp. 220–229.
- [25] Yuan Chou, Jason Fung, and John Paul Shen, “Reducing branch misprediction penalties via dynamic control independence detection”, in *Proceedings of the 13th International Conference on Supercomputing*, New York, NY, USA, 1999, ICS '99, pp. 109–118, ACM.
- [26] O. Ergin, D. Balkan, D. Ponomarev, and K. Ghose, “Increasing processor performance through early register release”, in *Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings. IEEE International Conference on*, oct. 2004, pp. 480 – 487.
- [27] Timothy M. Jones, Michael F. P. O’Boyle, Jaume Abella, Antonio González, and Oğuz Ergin, “Energy-efficient register caching with compiler assistance”, *ACM Trans. Archit. Code Optim.*, vol. 6, no. 4, pp. 13:1–13:23, oct 2009.

- [28] T.M. Jones, M.F.R. O’Boyle, J. Abella, A. Gonzalez, and O. Ergin, “Compiler directed early register release”, in *14th International Conference on Parallel Architectures and Compilation Techniques*, sept. 2005, pp. 110 – 119.
- [29] J. Alastruey, T. Monreal, V. Vinals, and M. Valero, “Microarchitectural support for speculative register renaming”, in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, march 2007, pp. 1 –10.
- [30] J.-L. Cruz, A. Gonzalez, M. Valero, and N.P. Topham, “Multiple-banked register file architectures”, in *Proceedings of the 27th International Symposium on Computer Architecture*, june 2000, pp. 316 –325.
- [31] R. Balasubramonian, S. Dwarkadas, and D.H. Albonesi, “Reducing the complexity of the register file in dynamic superscalar processors”, in *Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture*, dec. 2001, pp. 237 – 248.
- [32] D.M. Brooks, P. Bose, S.E. Schuster, H. Jacobson, P.N. Kudva, A. Buyuktosunoglu, J. Wellman, V. Zyuban, M. Gupta, and P.W. Cook, “Power-aware microarchitecture: design and modeling challenges for next-generation microprocessors”, *Micro, IEEE*, vol. 20, no. 6, pp. 26 –44, nov/dec 2000.
- [33] John Paul Shen and Mikko H Lipasti, *Modern processor design: fundamentals of superscalar processors*, Waveland Press, 2013.

- [34] Doug Burger and Todd M. Austin, “The simplescalar tool set, version 2.0”, *SIGARCH Comput. Archit. News*, vol. 25, no. 3, pp. 13–25, June 1997.
- [35] J. Adam Butts and Gurindar S. Sohi, “Use-based register caching with decoupled indexing”, in *Proceedings of the 31st annual international symposium on Computer architecture*, Washington, DC, USA, 2004, ISCA '04, pp. 302–, IEEE Computer Society.
- [36] M.M. Martin, A. Roth, and Charles N. Fischer, “Exploiting dead value information”, in *Proceedings., Thirtieth Annual IEEE/ACM International Symposium on Microarchitecture*, Dec, pp. 125–135.
- [37] S. Onder and R. Gupta, “Automatic generation of microarchitecture simulators”, in *Proceedings. 1998 International Conference on Computer Languages*, 1998, pp. 80–89.
- [38] Jeff Bastian and Soner Onder, “Specification of intel ia-32 using an architecture description language”, in *Architecture Description Languages*, 2005, pp. 151–166.
- [39] E. Quiones, J. Parcerisa, and A. Gonzalez, “Early register release for out-of-order processors with registerwindows”, in *16th International Conference on Parallel Architecture and Compilation Techniques*, 2007, pp. 225–234.
- [40] José F Martínez, Jose Renau, Michael C Huang, and Milos Prvulovic, “Cherry:

checkpointed early resource recycling in out-of-order microprocessors”, in *Proceedings. 35th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2002, pp. 3–14.