FINE-GRAIN STATE PROCESSORS

By

PENG ZHOU

A DISSERTATION

Submitted in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

(Computer Science)

MICHIGAN TECHNOLOGICAL UNIVERSITY

2009

Copyright © Peng Zhou 2009

This dissertation, “Fine-grain State Processors,” is hereby approved in partial fulfillment of the requirements for the degree of DOCTOR OF PHILOSOPHY in the field of Computer Science.

DEPARTMENT:
Computer Science

Signatures:

Dissertation Advisor _____

Dr. Soner Önder

Committee _____

Dr. Steven M. Carr

_____

Dr. David A. Poplawski

_____

Dr. Brian T. Davis

Department Chair _____

Dr. Linda M. Ott

Date _____

To My Parents and My Wife

Acknowledgement

First and foremost, I would like to thank my advisor Dr. Soner Önder, for his support, guidance, and encouragement in the Ph.D program. He showed me the way to approach research problems and taught me how to express my ideas. I am deeply impressed by his inspiring and encouraging way to guide me to a deeper understanding of knowledge. I am grateful for his advise and comments in this dissertation.

I also would like to thank the rest of my thesis committee: Dr. Steve Carr, Dr. David Poplawski, and Dr. Brian Davis, for their kind help and valuable comments on this work. Thanks also go to Dr. Zhenlin Wang from our research group, who gave me the suggestion and helped me on the career selection.

I really appreciate my friends and colleagues in school, who had fun with me and gave me the help in a place far away from home. I really enjoyed the life with them in the beautiful Keweenaw Peninsula.

Last but not least, I wish to thank my parents and my sister, for their continued support and encouragement. I am also very grateful to my wife Xiaodi, for her love and patience during my Ph.D period.

Abstract

Proper manipulation of processor state is crucial for high performance speculative superscalar processors. This dissertation presents a new state paradigm. In this paradigm, the processor is aware of the in-order, speculative and architectural states on an individual data location basis, rather than with respect to a particular point in the program's execution. We refer to the traditional processors which adopt a lump-sum approach with respect to the processor state as *Coarse-grain State Processors (CSP)*, and those which can classify individual data locations belonging to a particular state as *Fine-grain State Processors (FSP)*.

Fine-grain State Processors break the atomic state set into finer granularity at the individual value level. As a result, they can utilize correct values upon a mis-speculation. Furthermore, they can continue execution with a partially correct state and still maintain correct program semantics. Performing the state recovery without stopping the execution of future instructions potentially can hide the latency of the recovery process, resulting in zero-penalty speculation under ideal conditions.

This dissertation also presents a taxonomy of FSP. The taxonomy categorizes existing fine-grain state handling techniques and outlines the design space of future FSP designs. Based on the developed general framework, the dissertation explores applications of FSP on sophisticated uni-processor as well as multi-core/multi-threaded organizations. Two detailed FSP models are evaluated, *EMR* and *FSG-RA*, regarding control speculation and value speculation, respectively. In both models, the FSP technique handles processor states more efficiently and obtains much higher performance than traditional mechanisms. For example, EMR achieves an average of 9.0% and up to 19.9% better performance than traditional course grain state handling on the SPEC CINT2000 benchmark suite, while FSG-RA obtains an average of 38.9% and up to 160.0% better performance than a comparably equipped CSP processor on the SPEC CFP2000 benchmark suite.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Out-of-order and speculative executions have been the hallmark of high performance superscalar processors which dominated a large variety of systems from laptop computers to workstations and servers during the last decade. Processors utilizing these mechanisms need to handle processor states properly. For instance, if an exception occurs or a speculation misses, the processor state must be restored to a previous correct point to maintain the correct semantics. Proper manipulation of processor states is crucial for the successful implementation of speculation in contemporary processors [21, 47].

In this dissertation, we present a new state paradigm in which the processor is aware of the in-order, speculative and architectural states [24] on an individual data location basis, rather than with respect to a particular point in the program's execution. We refer to the traditional processors which adopt a lump-sum approach with respect to the processor state as *Coarse-grain State Processors (***CSP***)*, and those which can classify individual data locations belonging to a particular state as *Fine-grain State Processors (***FSP***)*. We

illustrate that if appropriate mechanisms are implemented to answer queries regarding the current state of data values on an individual basis, it is possible to salvage part of the work done during speculative execution after a mis-speculation, or, even better yet, to continue execution without a roll-back and recover only the damaged part of the state *in parallel with the execution of useful instructions*.

Various micro-architecture techniques that salvage work from a failed speculation attempt [48, 10, 44, 36], as well as reducing branch mis-prediction penalty [62, 16], all implement a variation of a fine-grain state maintenance mechanism. However, to the best of our knowledge no one to date pointed out the commonality of these micro-architectural mechanisms and named it. Furthermore, very few existing techniques [62, 16, 61] harvest the performance benefits of overlapping the state recovery with useful instruction execution.

In order to better illustrate this perspective, consider a simple example shown in Figure 1.1. Assume that the processor mis-predicted the branch instruction and reached point *C*. A processor that has the concept of only a coarse grain state needs to roll back to point *A*, restore the state to the in-order state at that point, and re-execute instruction I_4 to arrive at point *C* with a correct architectural state. On the other hand, if the processor knows which data values have been modified speculatively, it has two options. It can either restore the state as before by rolling back to point *A* but skip the execution of instruction I_4, i.e., salvage part of the work done during the speculative execution, or continue executing past point *C* without restoring the state, but clearly identify which values that make-up the

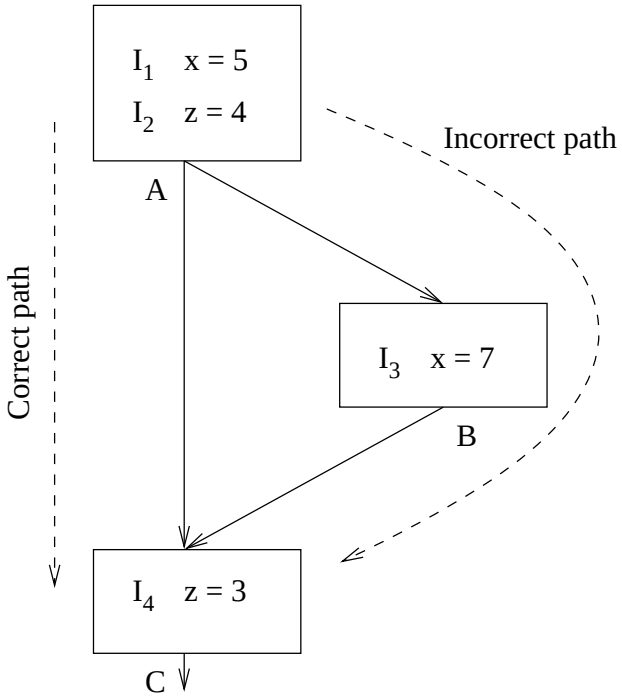


Figure 1.1. Mis-speculation and its effect on state

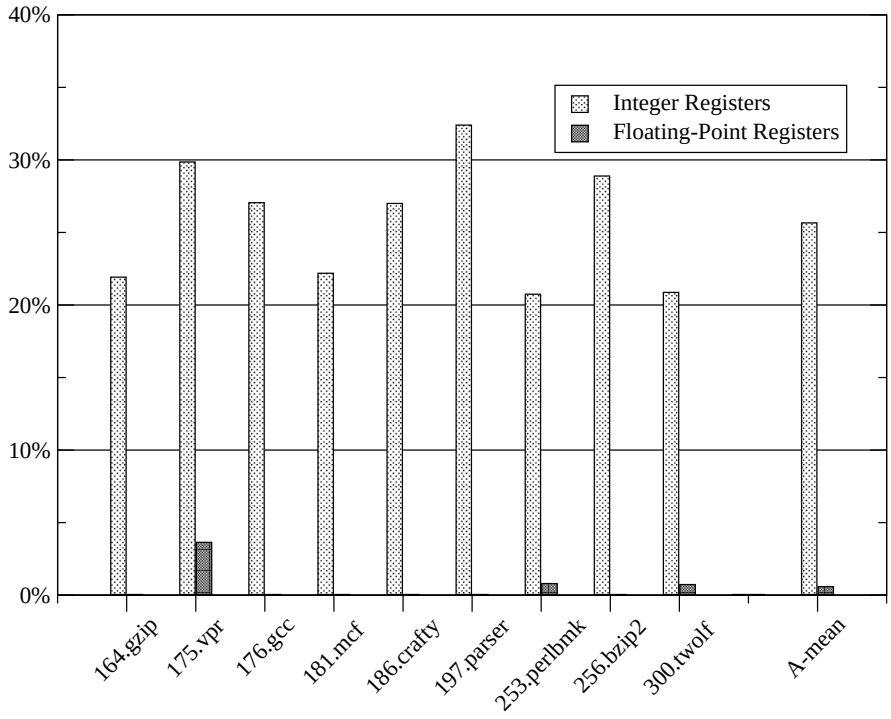
architectural state at point *C* have been damaged during the speculative execution (i.e., *x*) and block the references to those locations until their correct values are restored.

For most applications, rolling back and re-using salvaged results provide only limited benefits [35]. This is because the salvaged instructions may not be on the critical path of the program to shorten the execution latency when their results are reused. Furthermore, skipping over a subset of instructions is not easy and in general requires sophisticated micro-architecture techniques [8]. A majority of these techniques would pose significant design complexity in a processor implementation. Alternatively, continuing execution in parallel with the recovery of damaged values is quite feasible because all that is needed is the capability to identify the part of the state that is damaged and the means to restore these

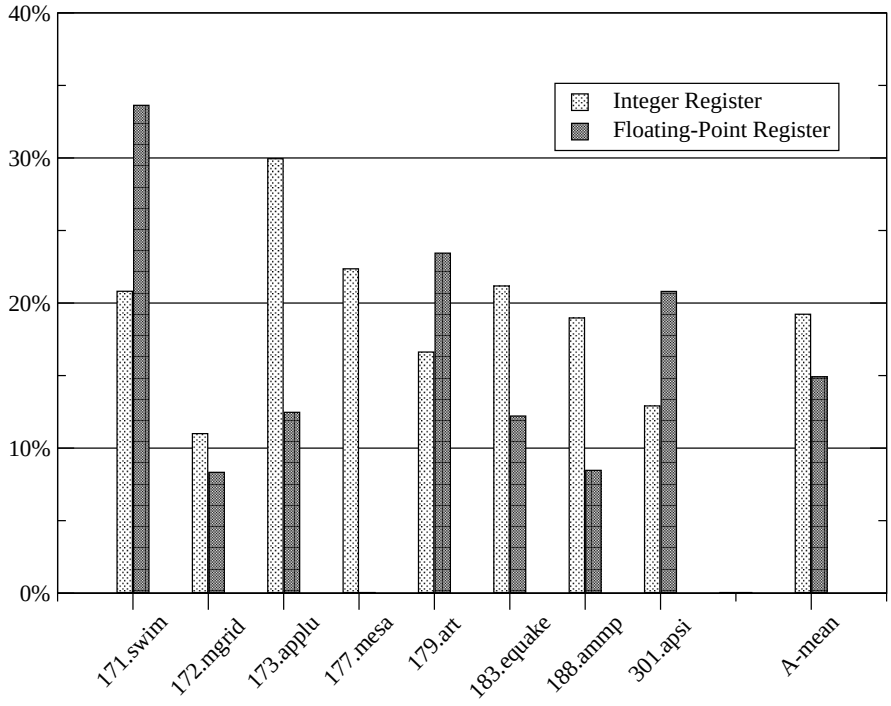
values on an individual basis. If the recovery process can be overlapped with the useful execution, this technique can significantly reduce and in some cases completely eliminate the performance penalty of mis-speculations.

Our experimental results show that values modified speculatively along the mis-predicted branch path are only a small part of the whole architectural state. Figure 1.2 illustrates that on an average the wrong speculative state upon branch mis-predictions of 17 SPEC2000 CPU benchmarks accounts for around 20% of the whole architectural state. Moreover, our experimental results show that more than 60% of the newly fetched instructions from the correct branch path do not reference those damaged values. Illustrated in Figure 1.3, on average only 18% and 40% of instructions along the correct path of a mis-predicted branch will reference damaged register values in CFP2000 and CINT2000, respectively.

As we can see, there exists a large potential for improving the performance. Upon a mis-speculation, an FSP which is aware of the processor state on an individual basis can potentially continue processing execution before the whole process state at the mis-speculation point is restored. For this purpose, newly fetched instructions accessing incorrect speculative values need to be blocked until the correct data values are restored. On the other hand, more than 60% instructions, which access only correct values, will be able to execute while the state recovery continues. Thus, the long latency of the branch mis-prediction recovery can be overlapped with those useful instructions. *Under ideal circumstances an FSP can achieve a zero-latency recovery if there are enough independent instructions.*

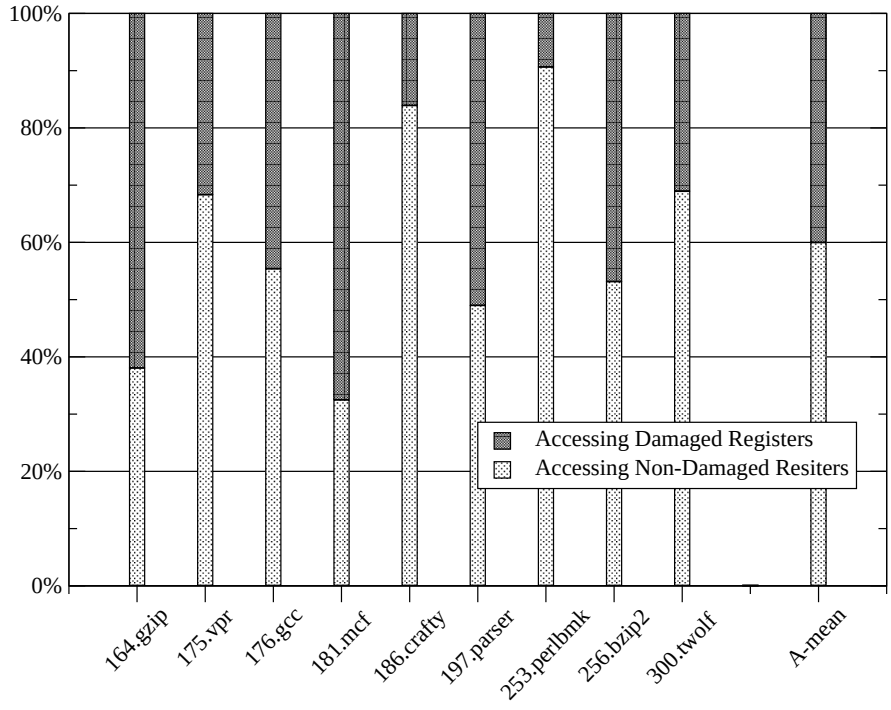


(a) SPEC CINT2000

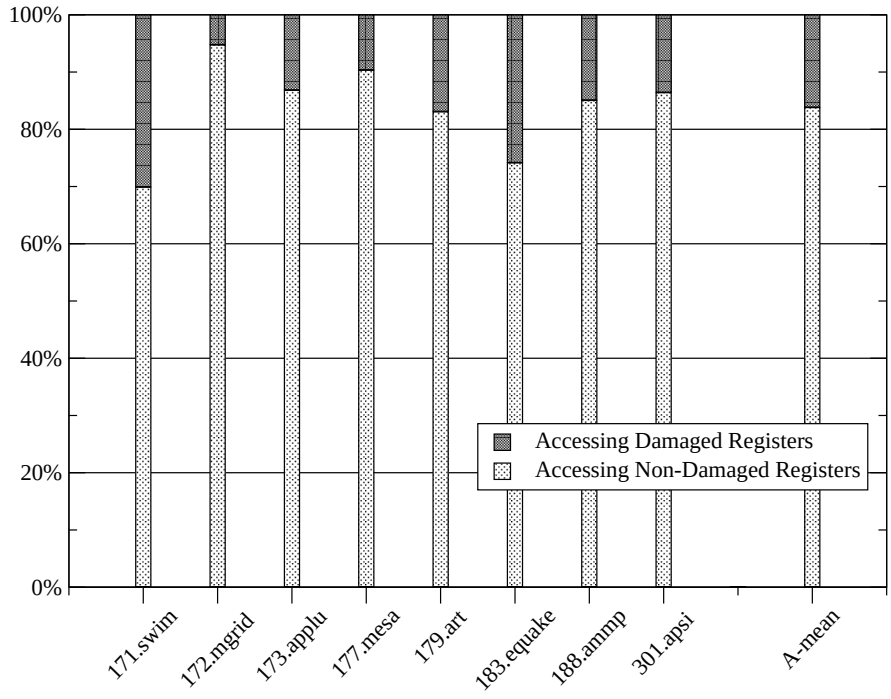


(b) SPEC CFP2000

Figure 1.2. Percentage of Speculative State upon Mis-predictions



(a) SPEC CINT2000



(b) SPEC CFP2000

Figure 1.3. Distribution of References to Damaged/Non-Damaged Registers upon Mispredictions

1.2 Research Goals

FSP breaks the main limitation of the state maintenance in CSP, which cannot utilize correct values within the architectural state when an exception occurs. Although retaining the usable results has been a significant focus in a number of recent proposals [48, 44, 35], most of these proposals have provided ad-hoc solutions tailored to the specific problem at hand. The fundamental concept of fine-grain state handling provides a key insight into the design of future systems where a systematic approach can be employed to separate usable values from dmaged values. Doing so can enable researchers to design processors which can aggressively pursue optimization opportunities upon an exception without waiting to restore the whole state to a known point. In this dissertation, our first research goal is to propose a general framework of fine-grain state handling. We define an FSP having the following properties:

1. **Identification property:**

   The processor can identify an individual data item such as a register file entry or a memory location as belonging to the *in-order* or *speculative* state, or as a *damaged* value;

2. **Block and shelve property:**

   The processor can block an instruction which references damaged values by shelving it until the damaged values are corrected;

3. **Correction property:**

The processor has the means to correct damaged locations on an individual basis after a mis-speculation;

4. **Unblocking property:**

   The processor can wake-up and execute shelved instructions which reference damaged values upon restoration of the damaged values in an arbitrary order;

5. **Parallelism-in-recovery property:**

   The processor can overlap the restoration of damaged values with the execution of instructions which do not reference damaged values. In other words, upon a mis-speculation, execution can continue with a partially correct state as the damaged values are repaired. This novel concept allows mis-speculation recovery to become free if there is independent work to do for the processor.

The design space of FSP is quite large. In principle, it can be applied in many different kinds of speculative execution environments by exploring the parallelism in mis-speculation recovery to improve performance. In this dissertation, we demonstrate that the FSP concept is applicable for both the control speculative execution and the value speculative execution. For this purpose we evaluate two FSP models: regarding control speculation the technique of Eager branch Mis-prediction Recovery, and for value speculation the technique of Fine-grain State Guided Runahead Execution.

Eager branch Mis-prediction Recovery (**EMR**) illustrates how one can apply the FSP concept for control speculation. In a nutshell, a traditional CSP either checkpoints the state

at branches for faster recovery, or sequentially re-constructs the in-order state at the mis-predicted point by waiting until the mis-predicted branch reaches the head of the reorder buffer. It cannot restart the execution from the correct path until the whole state at the mis-predicted branch is fully restored. However, the state checkpointing scheme is costly and the state re-constructing scheme is slow.

In contrast to the traditional schemes, EMR allows continuing the execution with a partially correct state, allowing branch mis-prediction recovery overlap with useful execution. The required hardware of EMR is modest. The hardware cost can be estimated roughly as the cost to save the checkpoints of the processor states. Comparing with the traditional checkpointing scheme, EMR needs to create the checkpoints only upon the mis-predicted branch instruction, instead of on every branch instruction.

Our second technique to illustrate the effectiveness of the concept of FSP investigates value speculation by using Runahead execution. Runahead execution was first proposed by Dundas and Mudge [14] for in-order processors and later applied to out-of-order processors by Mutlu *et al.* [37]. It is an effective method to tolerate rapidly growing memory latency in a superscalar processor. In this technique, When the instruction window is blocked by an L2-cache missing load instruction, the processor enters the “runahead mode” by providing a bogus value for the blocking load operation and pseudo-retiring it out of the instruction window. Under the “runahead mode”, all the instructions following the blocking load are fetched, executed, and pseudo-retired from the instruction window. Once the blocking load instruction completes, the processor rolls back to the point it entered the “runahead mode”

and returns to the "normal mode". Though all instructions and results obtained during the "runahead mode" are discarded, the runahead execution warms up the data cache and significantly enhances the memory level parallelism.

As it can be seen, Runahead execution behaves as a value speculation during which part of the state will become damaged. After the correct value of the blocking load instruction is fetched from the main memory, the Runahead processor, which is a CSP, has to roll back to the missing point and restart the execution with the correct processor state.

Application of FSP concepts in this realm results in a fine-grain state technique called Fine-grain State Guided Runahead execution (**FSG-RA**). FSG-RA is implemented as an SMT-like multi-threaded processor. When the missing load is resolved, FSG-RA is able to continue executing new instructions with a partially correct state, without rolling back. On the other hand, it only needs to re-execute those miss dependent instructions to repair the incorrect values updated during the "runahead mode". Furthermore, it can execute those instructions via an idle thread, in parallel with executing the newer instructions by the original thread. Thus, FSG-RA can improve the single-thread program's performance by exploiting the parallelism in the Runahead execution recovery in a multi-thread processor environment.

The concept of the fine-grain state is natural to reason about speculative executions and optimize speculative efforts. Comparing with the traditional CSP, our proposed FSP models, EMR and FSG-RA demonstrate that such an approach can provide impressive speed-ups without difficulties to scale processor key elements. With the fine-grain state

concept, mis-speculation recovery essentially becomes free if there is enough independent work to do for the processor. We therefore believe that the concept of FSP will open up new and exciting research opportunities in the micro-architecture community.

1.3 Dissertation Organization

The remainder of this dissertation is organized as follows. Chapter 2 presents the background for this dissertation. In Chapter 3, the simulation and experimental setup of this work is described. In Chapter 4, a taxonomy of FSP is introduced and related work are summarized and classified based on the taxonomy. Next, Chapter 5 introduces the design space of FSP and a general framework in which the concept can be implemented. This chapter also compares FSP and CSP regarding to the control speculation and the value speculation at a high level. Two FSP models, EMR and FSG-RA, based on the proposed general framework, are introduced in Chapter 6 and Chapter 7, respectively. Finally, a summary and the conclusion are given in Chapter 8.

Chapter 2

Background

This chapter is devoted to presenting a foundation for understanding this dissertation work. First, we discuss out-of-order execution and speculative execution as employed in contemporary processors. Then, the concept of processor states is illustrated. Next, we describe the traditional state maintenance and recovery mechanisms. Finally, the effect of register renaming technique on state recovery is presented.

2.1 Out-of-order Execution and Speculative Execution

Out-of-order execution and speculative execution are two milestones in the evolution of modern microprocessor architectures. These two techniques explore instruction level parallelism to achieve great performance.

Out-of-order execution breaks the limitation of the strict sequential execution defined by the program order. Instead of waiting for previous instructions to be finished, an instruction is issued and executed once its operands are ready. Given enough computing resources, a processor can issue multiple independent instructions out of program order to exploit parallelism at the instruction level. However, an out-of-order executed instruction might

modify the processor state before it should. In such a case, the processor state will not be consistent with the sequential execution model. For example, illustrated in Figure 2.1, suppose that instruction I_2 executes before I_1 out of order, and writes the result to the register file before I_1 executes. If later I_1 raises some exception, *e.g.,* a page fault, then the modification to the state introduced by I_2 needs to be reverted, before proper exception handling function is invoked.

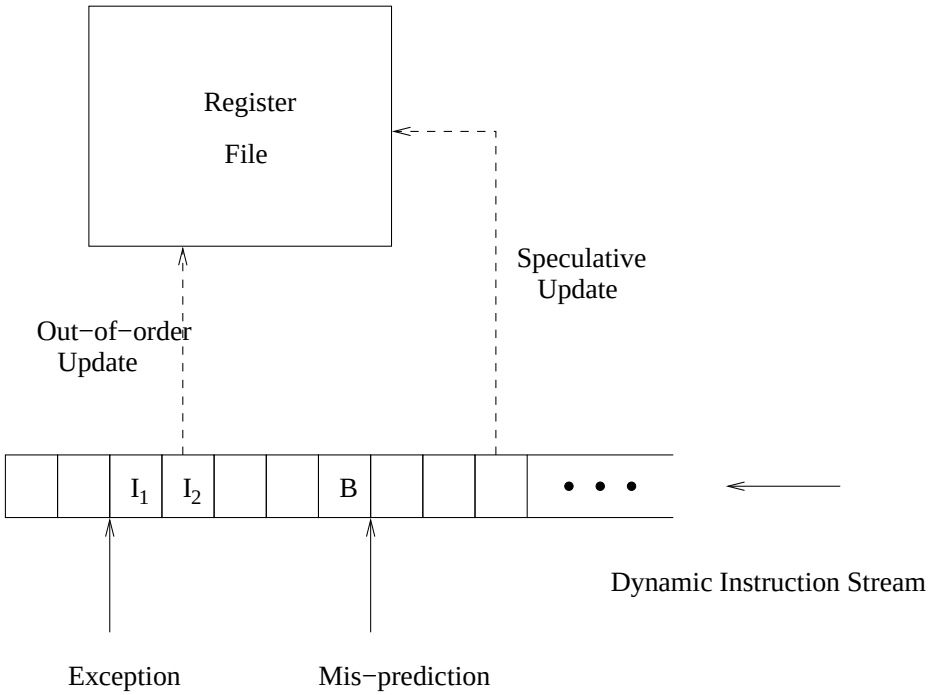


Figure 2.1. Exception and Mis-speculation Boundaries

Speculative execution is another important technique for modern microprocessors. There exists several different kinds of speculation techniques, such as control speculation, value prediction, and load speculation. Let us use the control speculation as an example to illustrate the effect of speculation on the processor state.

Control speculation is a technique which is based on the prediction of the direction and

the target address of branch instructions. Without control speculation, a processor has to stall upon encountering a branch instruction until its target address and direction become known. Given that on an average there is a branch instruction in every 3~5 dynamic instructions [34], it is not acceptable to stall the instruction stream upon each branch in a modern wide-issue superscalar processor. With control speculation, the target of a branch instruction is predicted based on the history pattern of dynamic branch instructions. The dynamic instruction stream can continue along a speculative path even before that branch is executed. Though a significant body of branch prediction methods has provided us with increasingly better prediction accuracies [59, 33, 50, 9, 23], branch predictors cannot be perfect. If a prediction is wrong, all instructions along the wrong path (i.e., following the branch *B* in Figure 2.1) have to be flushed from the pipeline. Accordingly, modifications to the processor state introduced by those speculatively executed instructions need to be eliminated.

Once an exception or a mis-speculation occurs, the machine needs to repair its state. This process is the state recovery. The machine in Figure 2.1 needs to roll back precisely to the exception boundary before I_1, if instruction I_1 brings an exception. Or, it needs to roll back to the mis-speculation boundary after *B* if the conditional branch *B* is mis-predicted (Note that the precise recovery point for *B* is the right boundary of the last delay slot instruction, if the delayed branch semantics is used.).

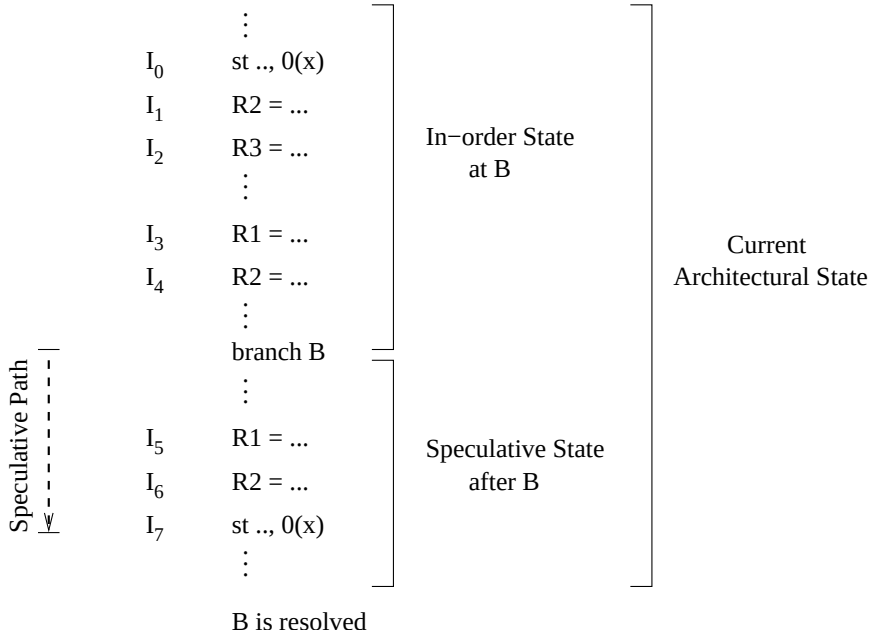
2.2 Processor States

In order to support out-of-order and speculative execution models and be able to recover from exceptions and mis-speculations, modern superscalar processors must be aware of different machine states, namely, in-order, speculative and architectural states [24] so that processor can always use the correct state for any externally visible changes in data locations and maintain correct program semantics.

The processor state contains the full set of architectural registers or logical registers that are visible at the ISA architecture level of the processor and the memory system. Let us consider Figure 2.2 which illustrates the different states when a branch is speculatively executed. For simplicity, assume that the architectural register file has 3 registers, $R1$~$R3$ and the memory has only one location at address x.

We define the in-order state as the state that would have been reached if the program is executed in program order, up to the point of interest, and the speculative state as the set of values produced that have not been committed. As it should be clear, newer instructions should use the values from the in-order state if the values have not been modified (i.e., they are not part of the speculative state) and should use values from the speculative state otherwise. The architectural state is defined as the union of the in-order and speculative states, and conveniently describes the set of values which any speculatively fetched instruction should reference. For example, in Figure 2.2 the set of values produced before the branch is defined as the *in-order state at B*, the set of speculative values produced after B is defined as the *speculative state after B*, and the architectural state at the point of

branch resolution is described as the *current architectural state*. Obviously, the in-order state at B is the same state as the architectural state when the branch has been fetched.



(a) State Definitions

In−order State at B		Speculative State after B		Architectural State When B is resolved	
I_0	(x) = ...			I_2	R3 = ...
I_2	R3 = ...	I_5	R1 = ...	I_5	R1 = ...
I_3	R1 = ...	I_6	R2 = ...	I_6	R2 = ...
I_4	R2 = ...	I_7	(x) = ...	I_7	(x) = ...

(b) State Sets

Figure 2.2. In-order, Speculative and Architectural States upon Speculation

As it can be seen from the figure, the in-order state at B includes x, $R3$, $R1$ and $R2$ which are defined by instruction I_0, I_2, I_3 and I_4, respectively. Let us express it as $\{x(I_0), R3(I_2), R1(I_3), R2(I_4)\}$. Note that the assignment to $R2$ in I_1 is superseded by the assignment to $R2$ in I_4. Therefore, it does not belong to the in-order state at B.

Assignments to $R1$, $R2$ and x in instructions I_5, I_6 and I_7 make up the speculative state after B, $\{R1(I_5), R2(I_6), x(I_7)\}$, because they are speculatively executed instructions and they are control depended on B. Any newer instructions after this code sequence will reference the state $\{R3(I_2), R1(I_5), R2(I_6), x(I_7)\}$, which is the architectural state combining the in-order state at B and the speculative state after B.

If the speculation is correct where B is resolved, all assignments along the speculative path become non-speculative. The speculative state becomes part of the in-order state, because the speculative path after B is the correct program stream. The current architectural state also becomes the in-order state at this point. In contrast, if B is mis-predicted, then the speculative execution was wrong. The processor needs to repair the incorrect architectural state back to the correct in-order state at B.

2.3 State Maintenance and Recovery

Traditionally, there have been two kinds of state maintenance and recovery mechanisms. One is referred to as the *State re-constructing* mechanism proposed by Smith and Pleszkun [47]. The other is referred to as the *Checkpointing* mechanism proposed by Hwu and Patt [21].

2.3.1 State re-constructing

To address the problems of out-of-order execution and precise interrupts in pipelined processors, Smith and Pleszkun proposed the reorder buffer (ROB), the history buffer and the future file designs to support the state recovery and maintenance. We refer to them all

as the *State re-constructing* mechanisms.

Reorder Buffer

The reorder buffer (ROB) is implemented as a circular buffer with a head pointer and a tail pointer. Shown in Figure 2.3, once an instruction is fetched and decoded, it is inserted into the tail of the ROB. After it is executed, its result and the exception flag are stored in the corresponding entry in the ROB. When this instruction reaches the head of the ROB, it will write its result to the logical register file and release the entry, if it is exception-free; Otherwise, the processor needs to flush the pipeline and restart the execution.

Similar to other producer instructions, a store instruction is only allowed to commit its result to the memory hierarchy, including the cache and the main memory, when it reaches the head of the ROB. This is, at this point, all previous instructions, including all memory operations, are already committed and known to be exception-free. Before it is committed to the memory, a store instruction can keep the value to be written to the memory in its allocated ROB entry, or, generally, in some associated buffer, *e.g.*, the store queue.

From the point of view of states, one can vision that the logical register file always has the in-order register state, and the memory system always has the in-order memory state. On the other hand, the ROB entries store the speculative state of registers, and the store queue entries store the speculative memory state. The union of the in-order state and the speculative state is the processor's architectural state.

Any newly fetched instruction will reference the architectural state of registers by accessing the logical register file and the ROB entries simultaneously. In order to do that,

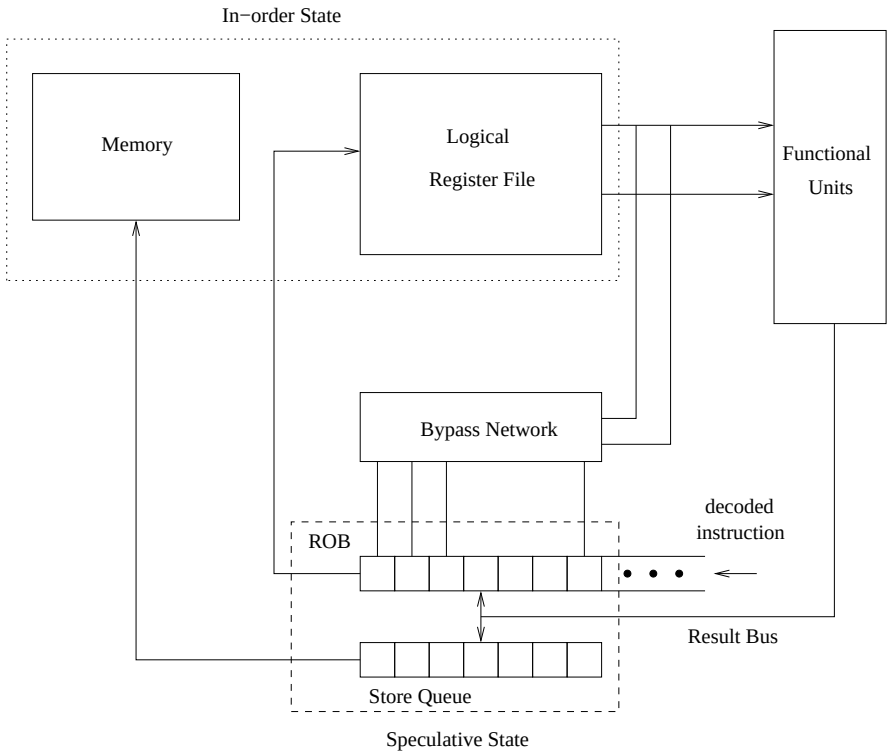


Figure 2.3. Reorder Buffer

processor implements the bypass paths from the ROB entries to the register file. In order to reference the architectural memory state, load instructions need to access the memory system and the store queue simultaneously.

In a speculative processor with a ROB, once an exception occurs, the exception is not handled until the instruction triggering the exception reaches the head of the ROB. At this point, it is easy for the processor to roll back the damaged architectural state into the correct in-order state, because the desired state is already stored in the register file and the memory. The processor simply discards the wrong speculative state stored in the ROB and the store queue, and then restarts from the exception point with the correct in-order state.

In another words, the in-order state at the exception point is constructed by waiting for

the instruction rising the exception to reach the head of the ROB, and retiring previous instructions one by one in the program order. This is the reason why the technique is referred to as the *State re-constructing* mechanism.

History Buffer

In the above reorder buffer mechanism, the speculative register state is stored in the ROB entries, and the in-order register state is always present in the logical register file. Alternatively, the register file can be used to store the architectural register state and the history information of the in-order register state can be buffered to support the state recovery if an exception occurs. This method is called the history buffer technique.

The history buffer is implemented in a similar way to the reorder buffer, illustrated in Figure 2.4. The history buffer is a circular buffer which has a head pointer and a tail pointer. Once an instruction is fetched and decoded, it is assigned an empty entry and inserted into the tail of the history buffer. Meanwhile, if it is a register producer instruction, the current value of its destination register is read from the register file and stored into the allocated entry. After the instruction is executed, the execution result is written into the register file immediately, and the exception flag is recorded in the history buffer entry.

From the point of view of states, the logical register file in the History Buffer mechanism always represents the latest architectural state of registers. The previous in-order values of all speculatively executed instructions' destinations are stored in the history buffer entries, in program order. In other words, the history buffer represents the *complement* set of the speculative register state.

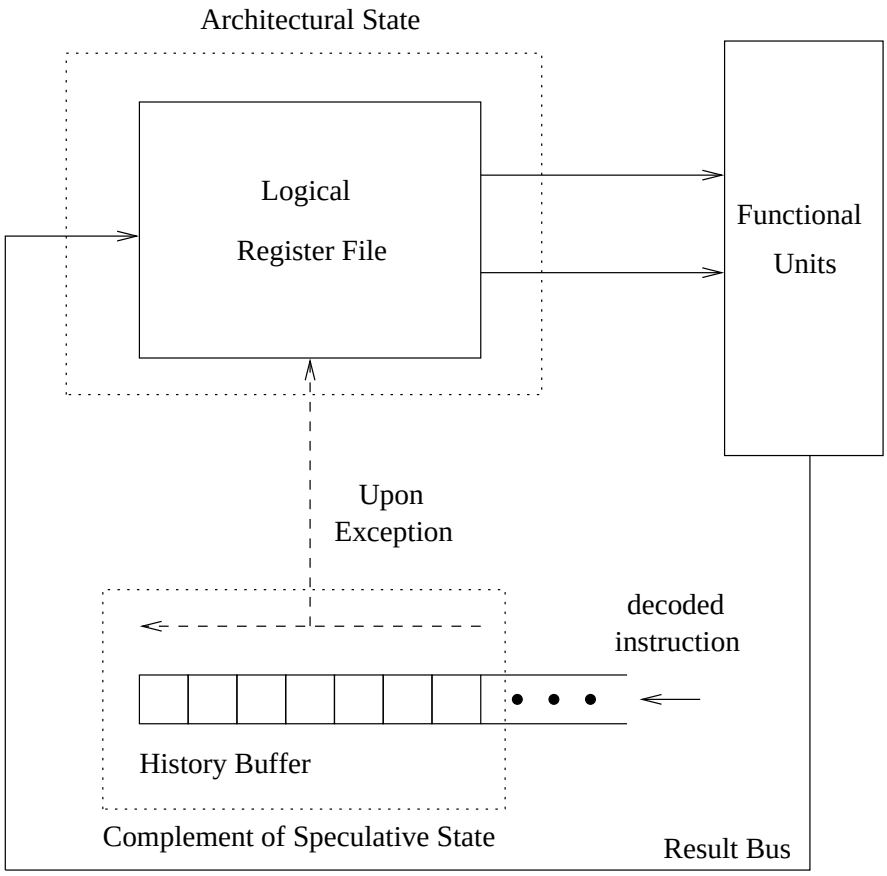


Figure 2.4. History Buffer

When an instruction reaches the head of the history buffer, its history buffer entry can be safely released if there are no associated exceptions. If this instruction's exception flag is set, the pipeline of the processor is stalled, and the state recovery process is invoked. In order to restore the correct state, the processor empties the history buffer entries one by one from the tail towards the head, and restores each saved history value back into the register file.

After all saved previous in-order values are written back into the register file in reverse program order, the wrong speculative state in the architectural state is totally eliminated,

and the correct in-order state at the exception point is restored. Once the head entry allocated for the exception instruction is scanned and processed, the processor is able to restart from the exception point with a correct in-order state.

The history buffer scheme is different from the reorder buffer scheme only with how to handle the register state. To handle the memory state, the history buffer scheme utilizes the same method as the reorder buffer scheme. That is, a store instruction only commits to the memory when all preceding instructions are committed without any exception. Before the retirement, a store instruction keeps the values to be written in the store queue. Like in the reorder buffer scheme, the memory hierarchy always represents the in-order memory state and the store queue holds the speculative memory state.

Future File

The third variant of the *State re-constructing* mechanism is the future file, an optimization of the reorder buffer implementation. The idea of the future file is to maintain two separate register files, the future file and the architectural file, illustrated in Figure 2.5. When an instruction is executed and finished, its result is written into the future file. When an instruction retires from the head of the ROB, it will update its result into the architectural file.

From the point of view of states, the future file maintains the architectural state of registers. It consists the in-order values and the speculative (future) values, so it is called the future file. The architectural file always reflects the in-order state of machine's architectural registers. If an instruction reaches the head of the ROB with an error, the architectural file

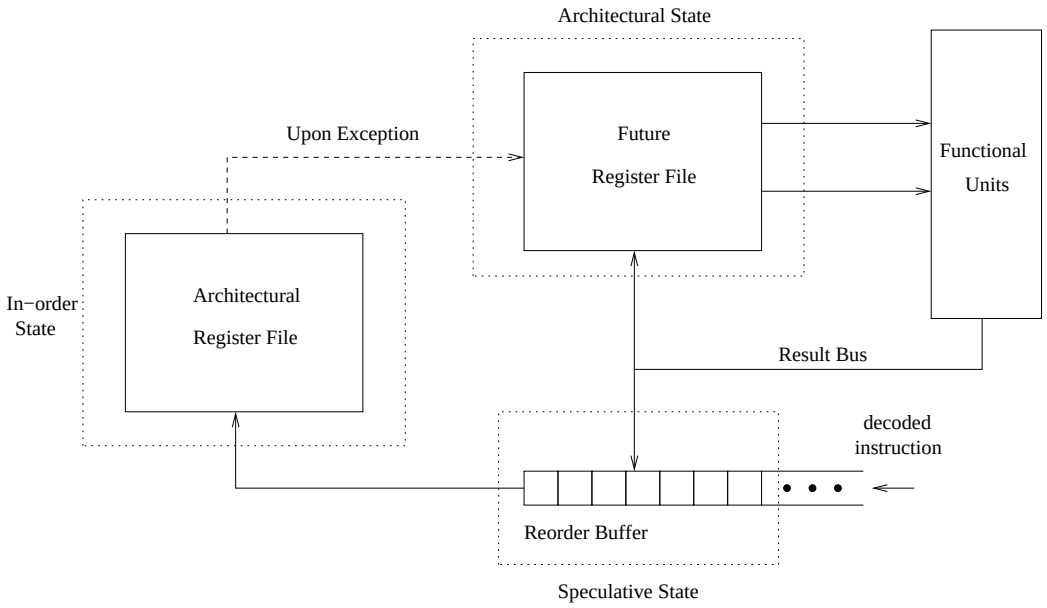


Figure 2.5. Future File

is then used to restore the future file. Either the whole register file is copied as a set, or only speculatively modified registers are restored. To implement the latter, those register designators associated with ROB entries are copied from the ROB's head to the ROB's tail.

As it can be seen from the figure, the future file in fact implements the functionality of the physical register file with respect to the register renaming, though Smith and Pleszkun did not use the term of the register renaming in [47]. The future file can be considered as a physical renaming register file which has the same size as the architectural logical register file. Therefore, the register renaming map table can be omitted.

In the future file scheme, the memory state is also handled in the same way as in the reorder buffer and the history buffer. The memory is always an in-order state memory, and the speculative memory state is buffered in the store queue.

2.3.2 Checkpointing

An alternative to sequential re-construction of processor states is to save a snapshot of machine state at appropriate points of the execution. This approach is referred to as the *Checkpointing* mechanism, which was first introduced by Hwu and Patt [21].

With checkpointing, the processor maintains a collection of *logical spaces*, where each logical space is a full set of architectural registers and memory locations visible at the ISA level of the machine. There is only one logical space used as the *current* space for the current execution which represents the architectural state. Other backup logical spaces contain the copies of the in-order state that correspond to previous points in the execution.

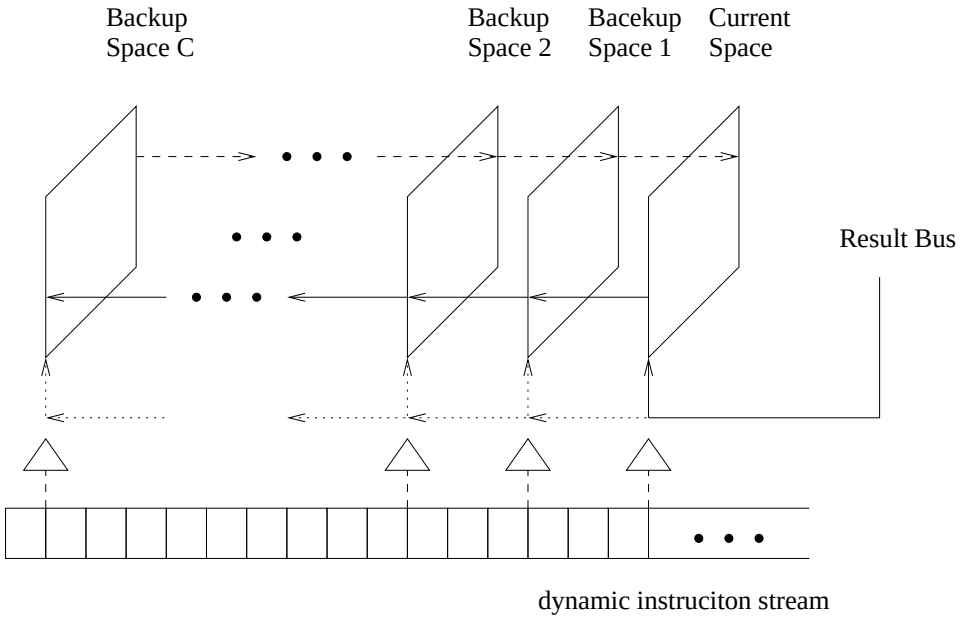


Figure 2.6. Checkpointing Maintenance and Recovery Scheme

During the execution, the processor creates a checkpoint of the state by copying the current logical space into a backup space, as shown in Figure 2.6. Upon each checkpoint,

the effect to the current architectural state introduced by all instructions to the left of that checkpoint are allowed, and the effect introduced by all instructions to the right of the checkpoint are excluded. Therefore, each logical space represents the in-order state at the creating point. The logical spaces are maintained as a stack. If the stack is full, making a checkpoint has to wait until the oldest one is safely released. When a fault exception or a branch mis-prediction occurs, the architectural state can be restored to the in-order state at the exception point by recovering the current space back to the corresponding logical space.

Ideally, a checkpoint should be created at each instruction boundary so that a processor is able to restore the correct state if any instruction meets an exception. Otherwise, if there is no backup state available at the exception point, the processor has to roll back to the nearest checkpoint and discard some useful work. However, the cost and overhead of creating a checkpoint at each dynamic instruction boundary is too high. This is the fundamental dilemma of the checkpointing recovery mechanism.

On one hand, we need to create as many checkpoints as possible to make the exception recovery fast. On the other hand, we need to keep the cost of checkpoints as low as possible. To address this issue to some degree, Hwu and Patt distinguished fault exceptions and branch mis-predictions with respect to the state recovery, because fault exceptions actually happen much less frequently than branch mis-predictions. Accordingly, they proposed to create a checkpoint at each branch instruction for the mis-prediction recovery, and to create checkpoints at widely separated points in the instruction sequence for the fault exception

recovery.

Another issue of the checkpointing scheme is how to implement the backup spaces of the state. Generally, there are two types of techniques for implementing multiple logical spaces: *copying* and *difference* techniques.

The copying technique is normally used to create the logical space of the register state. When a checkpoint is created, the architectural register state in the current space is copied into the logical space. Since some instructions to the left of the checkpoint might have not been issued yet, the copied state is not the in-order state at that time. Therefore, the copied state needs to be updated as instructions to the left of the checkpoint complete. To avoid extra increase of the access bandwidth of the register file, each bit of the registers is implemented by $C+1$ physical cells, one cell for the current space and C cells for C backup logical spaces.

The difference technique maintains a list which records the difference of the execution state from one instruction boundary to another. Normally, the checkpoint of the main memory state is implemented via the difference technique. There are two directions a state difference can be recorded, backward and forward. Each logical space implements either the backward difference or the forward difference to manipulate the memory state.

The backward difference maintains a history value list in which each entry keeps the history value prior to the modification to the state. Figure 2.7 illustrates the memory design when a backward difference is employed. When a store instruction commits its value to the memory, the address and the previous value at this location in the memory

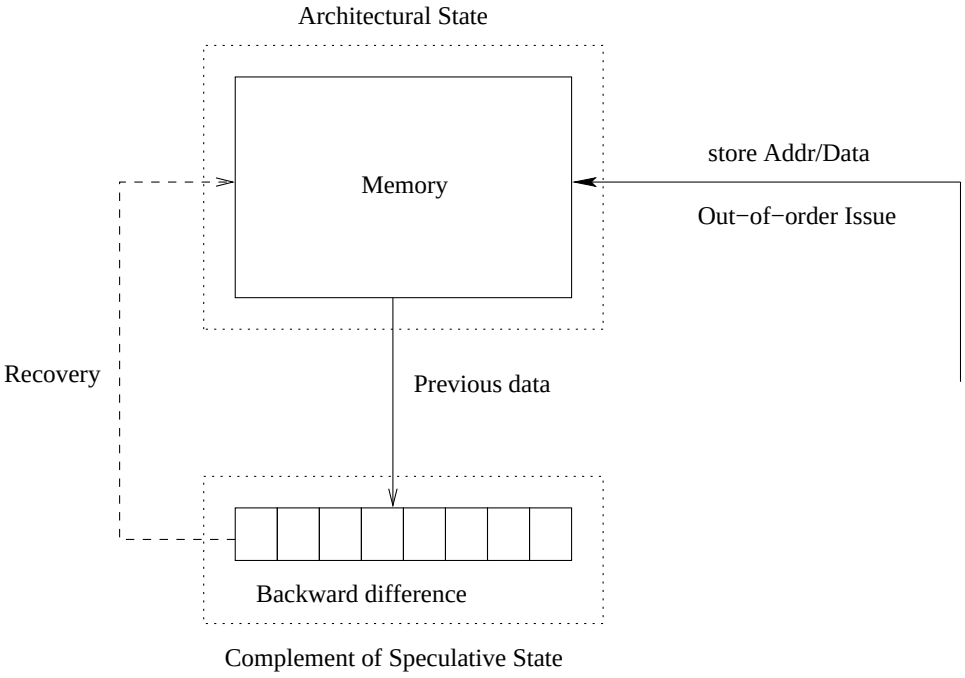


Figure 2.7. Logical Space of Memory by Backward Difference

are saved in the backward difference list. The backward difference list preserves the store instruction's committing order to the memory, not the program order they appear in the dynamic instruction sequence. As it can be seen from the figure, the main memory represents the architectural state, including the in-order state and the speculative state. The backward difference holds the complement of the speculative state. During the state recovery, the history data saved in the list are stored back to undo all modifications to the memory introduced by the wrong speculative state. Thereby the in-order state of the memory at the checkpoint can be repaired.

In contrast to the backward difference, the forward difference keeps all speculative values within a logical space, and preserves the program order of stores. Shown in Figure 2.8, when a checkpoint is created, the memory holds the in-order state at this point,

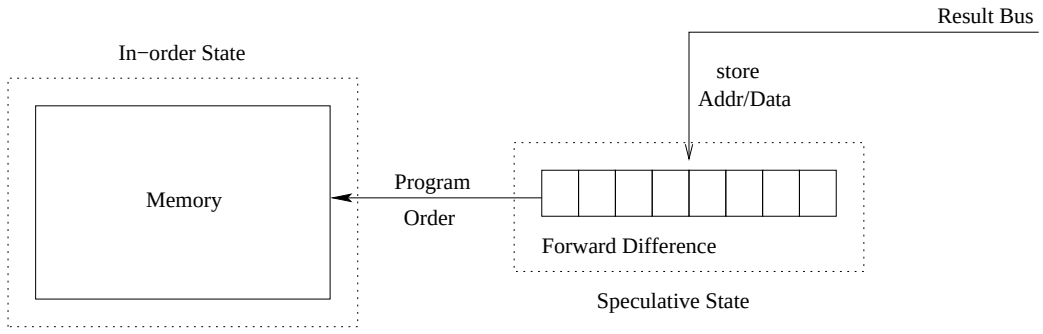


Figure 2.8. Logical Space of Memory by Forward Difference

and the forward difference will keep the speculative state of this logical space, until the next checkpoint is created. From this point of view, the memory manipulation scheme introduced in the previous state re-constructing mechanism is a special case of the forward difference technique on which the memory system always represents the machine's in-order memory state and the store queue keeps the speculative memory state introduced by all in-flight instructions.

2.4 Register Renaming and State Maintenance and Recovery

In order to eliminate WAR (write after read) and WAW (write after write) data hazards, modern superscalar processors normally utilize the register renaming technique. The design space of register renaming is large and it is beyond the scope of this work. In this work, we only consider the effect of register renaming on the state maintenance and recovery.

The reorder buffer scheme, described in Section 2.3.1, provides a straightforward implementation of register renaming. In ROB, an architectural logical register can have multiple definitions in-flight, which are kept in different ROB entries in program order. In

order to read the correct value of an operand, an instruction can associatively search the ROB and access the entry for the most recent definition of its source operand when it is issued. The ROB entries implement the renaming functionality.

Generally, processors utilize a physical register file to implement register renaming. The physical register file can be separated from the logical register file, or they can be combined together as a unified register file. In order to eliminate the false data dependences and exploit deep speculation, physical register file size will be larger than the size of the logical register file visible at the ISA level. Once a producer instruction is decoded, it is allocated a free physical register as the renaming location for its destination. This mapping of information between the logical register designator and the physical register designator is then recorded so that subsequent instructions can reference the latest value from the renaming physical register.

The mapping information is normally stored in the register map table (MAP), which is also referred to as the register alias table (RAT) [58, 20]. In a superscalar processor, if multiple instructions need to be decoded and renamed simultaneously in every cycle, then the map table, or the alias table, has to be multi-ported. For instance, if it is a 4-issue superscalar processor, then the map table needs 8 read ports and 4 write ports, assuming each instruction has two source operands and one destination.

The map table structure can be implemented in at least two ways: the RAM structure and the CAM structure [40]. The different structures have the different implications for the state maintenance and recovery.

2.4.1 RAM-structured MAP

The RAM-structured map table is illustrated in Figure 2.9. In this structure, the total number of the map table entries is equal to the total number of the architectural logical registers. Each logical register has a corresponding entry in the map table, and the logical register's designator can be used as the index to access the table. Each entry contains a physical register designator which is allocated and associated with this logical register. The RAM-structured map table itself is implemented as a register file in which each cell holds just enough bits to specify a physical register's designator, instead of a 32- or 64-bit value.

As it can be seen, the map table represents the register state. Register values stored in the physical register file can be referenced through indices in the map table. Normally, processor employs a front-end map table as the working map table, corresponding to the current architectural state. If an exception occurs, processor needs to repair the front-end map table to restore the correct register state. The restoration process can be done via either the checkpointing method or the state re-constructing method.

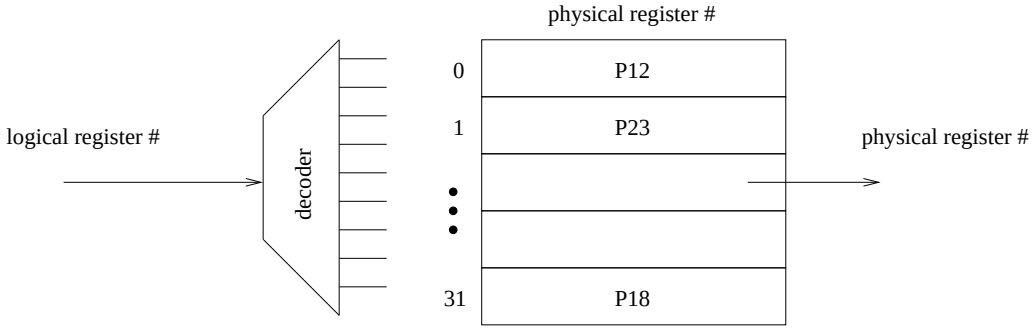


Figure 2.9. RAM-structured Map Table

Examples of processors which employ a RAM-structured map table are the MIPS R10000 processor and the Intel Pentium IV processor. Regarding to the branch mis-speculation recovery, the MIPS R10000 processor utilizes the checkpointing method to manipulate the state, and the Pentium IV processor utilizes the state re-constructing method.

MIPS R10000

The MIPS R10000 maintains a branch stack where each entry contains a complete copy of the integer and floating-point register map tables [58]. At the point of recognizing a mis-prediction, the processor restores the front-end map table from the corresponding checkpoint. While the checkpoint scheme yields fast recovery, its hardware cost can be prohibitive. Suppose there are N physical registers, and the instruction set contains 32 integer and 32 floating-point logical registers, then the size (bits) of each checkpoint is:

$$Checkpoint\ Size = 32 \times \log_2 N + 32 \times \log_2 N = 64 \times \log_2 N \tag{2.1}$$

If C checkpoints are implemented, then the total size of the checkpoint stack is:

$$Total\ Checkpoint\ Size = C \times 64 \times \log_2 N \tag{2.2}$$

As we can see, checkpointing the map table at each outstanding branch instruction is costly. The available space for storing the checkpoints, on the other hand limits the number of pending branches that can be in-flight. In the MIPS R10000, only 4 pending branches

are allowed to be in-flight because its branch stack has only 4 entries as shown in Figure 2.10.

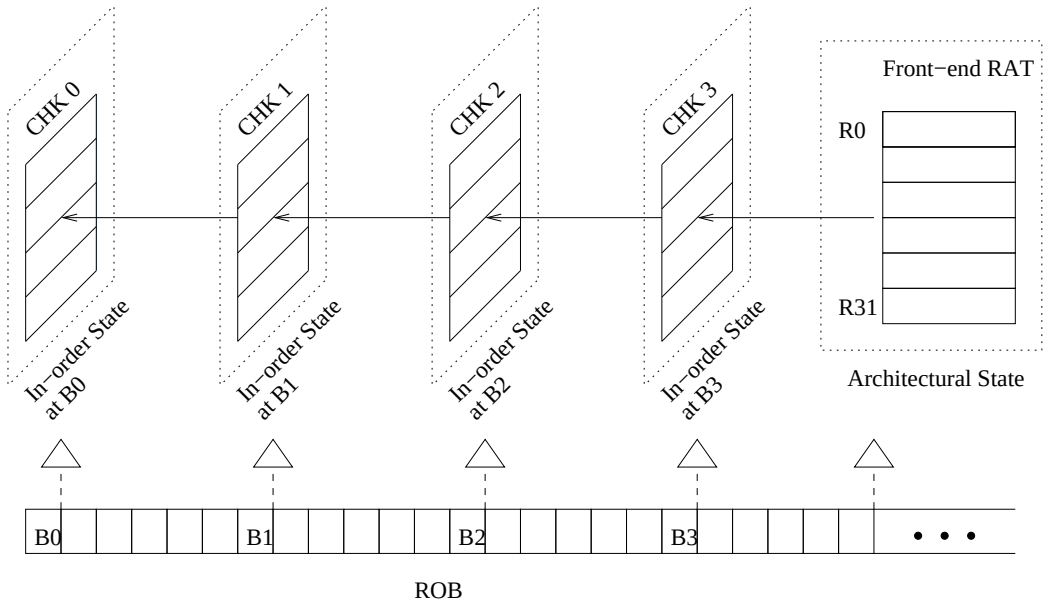


Figure 2.10. Checkpoint Stack in MIPS R10000

To some degree, the register renaming technique makes creating checkpoints easier. Consider the original checkpoint scheme described in Section 2.3.2. When a checkpoint is created, some instructions left to this point may have not been finished yet. In order to bring this checkpoint into the desired in-order state later, the results of these instructions have to be written into the corresponding backup space when they are ready.

Using the register renaming technique, processor allocates a free physical register for each instruction with a register destination during decoding. Therefore, in-order state at a particular point can be represented as the format of the physical register designators stored in the map table at that point. The in-order values can be referenced by means of the physical register designators. Thus, updating values into the backup space can be omitted.

Intel Pentium IV

The NetBurst micro-architecture of the Pentium IV also utilizes the RAM-structure. Unlike the MIPS R10000 processor which uses a checkpointing recovery mechanism, it utilizes a state re-constructing recovery scheme. Two map tables named the front-end RAT and the retirement RAT [20] are maintained. The front-end RAT stores the architectural state and the retirement RAT stores the machine's in-order state, shown in Figure 2.11.

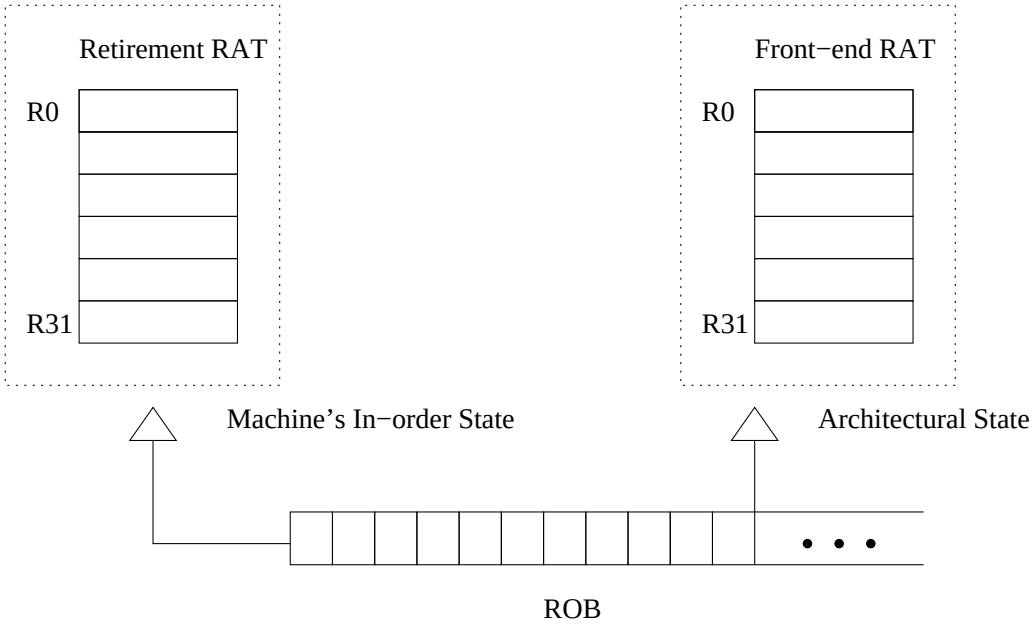


Figure 2.11. State Re-constructing in Pentium IV

Any newly fetched and decoded instruction will access and modify the front-end RAT, thus, it always contains the latest architectural state. When an instruction retires, it updates the retirement RAT to indicate that its result register is in the in-order state. The retirement logic ensures that an exception occurs only if the operation causing the exception is the oldest, non-retired operation in the machine. That is, an instruction can rise an exception

only when it reaches the head of the ROB. At this point, the machine's in-order state is also the in-order state at the exception point. Therefore, processor can restore the architectural state, or the front-end RAT, from the retirement RAT.

Although Pentium IV requires only two RAM-structured RATs, the recovery process may take a long time as renaming cannot start until all instructions prior to the mis-predicted branch retire. If a long latency operation prior to the branch exists, *e.g.*, a cache miss, the mis-prediction penalty increases significantly.

2.4.2 CAM-structured MAP

Alternative to the RAM-structured scheme, the other way to implement the map table is to use a content-addressable memory (CAM). Illustrated in Figure 2.12, in a CAM-structured map table, the total number of table entries is equal to the total number of physical registers. Each entry has two fields, the logical register designator field and the valid bit field. Since a logical register might have multiple definitions in-flight, the valid bit is always set by the latest definition. Once a logical destination register is mapped into a free physical register, this logical register number is written into the corresponding entry in the table. Also, its valid bit is set and the previous valid bit of the same logical register is located through an associative search and cleared.

When an instruction accesses the table to read its operand's renaming physical register designator, the operand's logical register number is used to search the table's logical register designator field associatively. If there is a match and the corresponding valid bit is set, then this entry's index is generated through an encoder as the renaming physical register number.

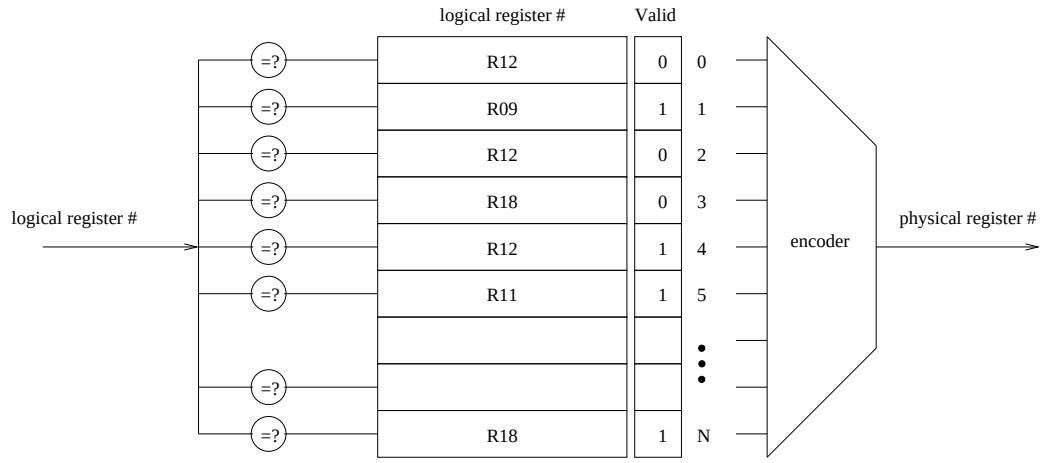


Figure 2.12. CAM-structured Map Table

The valid bit indicates the latest renaming physical register of each logical register. From this perspective, the valid bit vector presents the current architectural register state. To create a checkpoint of the register state, processor needs only to make a shadow copy of the valid bit vector. Suppose there are N physical registers, the size (bits) of each checkpoint is:

$$Checkpoint\ Size = N \tag{2.3}$$

If C checkpoints are implemented, then the total size of C checkpoints is:

$$Total\ Checkpoint\ Size = C \times N \tag{2.4}$$

In the CAM-structure implementation, creating checkpoints is not expensive. Only valid bits, not the whole map table, need to be copied. The CAM-structure map table is utilized in the DEC Alpha 21264 [13, 26]. The Alpha 21264 supports up to 80 instructions in-

flight, and a checkpoint is created at every instruction boundary. Illustrated in Figure 2.13, it provides the capability of precise state recovery at any of the 80 in-flight instructions.

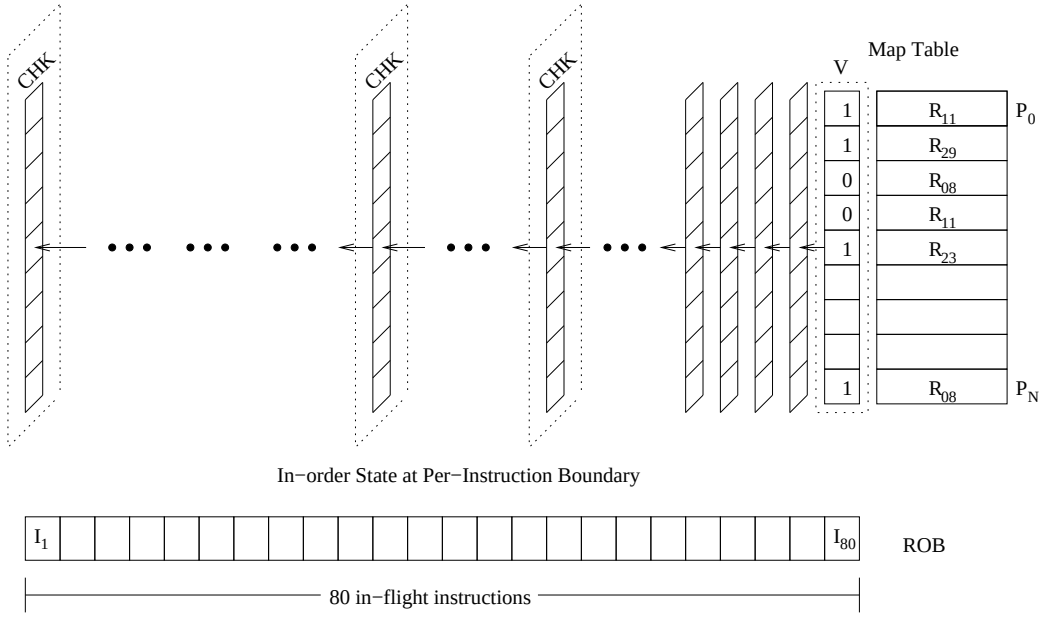


Figure 2.13. Per-Instruction Boundary State Recovery in Alpha 21264

Though the hardware cost of creating checkpoints with the CAM-structure map table is not too expensive, the CAM-structure itself may become a bottleneck. With the CAM-structure, the total number of entries in the maple table is equal to the total number of physical registers. Moreover, processor needs to perform an associative search to access it. On the other hand, with the RAM-structure, the number of entries in the map table is equal to the number of ISA logical registers, independent of the number of physical registers, and the table access is fast. That makes the CAM-structure less scalable than the RAM-structure.

Given these observations, the CAM-structure may not scale well with future wide-issue high performance microprocessors. In this work, we mainly focus on the RAM-structure

designs with respect to the state maintenance and recovery.

2.5 Summary of Background

In this chapter, the basic concept of process states is described. In order to support out-of-order and speculative executions, processor needs to be aware of the in-order state, the speculative state, and the architectural state at different interest points. To recover from an exception or a mis-speculation, processor needs to eliminate the wrong speculative state, introduced within the speculative execution, from the architectural state and restore it back to the in-order state at the exception point.

Traditionally, there have been two main mechanisms to manipulate states, state re-constructing and checkpointing. Despite the fact that they appear to be drastically different, they share a common property. Both are based on the coarse-grain state concept. As we can see, any processor utilizing these mechanisms can be classified as a CFP. Upon an exception, neither of these mechanisms allows the resumption of fetching and renaming of new instructions from the correct path until the whole set of processor state is restored.

Chapter 3

Simulation and Experimental Setup

In this chapter the simulation tool and the experimental setup used for this dissertation are introduced and presented.

3.1 Simulation Tools

For processor architecture researchers, realizing novel micro-architecture designs in hardware is too expensive and time-consuming, especially at the prototype stage of development. Thus most research relies on simulation tools that can estimate the performance of the micro-architecture by simulating a variety of benchmark programs. A commonly used approach for developing micro-architecture simulators is hand coding them in a general purpose programming language. For instance, widely used in the computer architecture research community, the SimpleScalar simulator [7] is written in C and the M5 simulator [6] is written in C++.

Although writing simulators in the general purpose languages is a straightforward process, it is difficult to retarget such simulators to a modified micro-architecture or an instruction set architecture once they are built. In this work, instead of hand coding

simulators, we use the *Flexible Architecture Simulation Tool* (FAST) [38] to generate simulators automatically, in which processor specifications are written in a domain specific language called *Architectural Description Language* (ADL). FAST currently supports the MIPS ISA [41] which is also implemented in ADL.

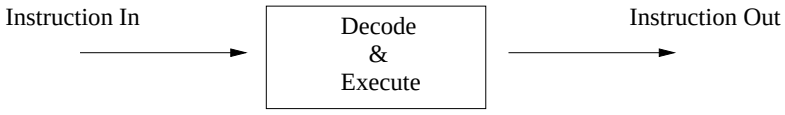


Figure 3.1. Fast Functional Simulator

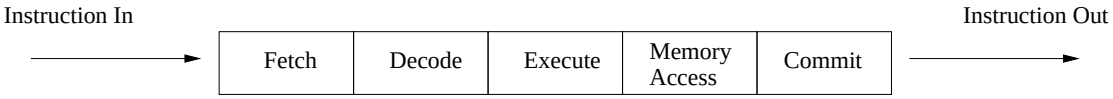


Figure 3.2. 5-Stage Pipeline Simulator

Shown in Figure 3.1, Figure 3.2 and Figure 3.3, FAST is comprised of a variety of different cycle-accurate processor simulators that are from a fast functional simulator which executes instructions one by one through a simple one stage pipeline, and a 5-stage in-order pipeline processor simulator, to a complex out-of-order superscalar simulator [38]. In this work, the out-of-order superscalar simulator is mainly used for exploring the fine-grain state processor designs and evaluating their performance.

The out-of-order superscalar description of FAST lacked the cache simulator and assumed a perfect cache hierarchy system. As one of the contributions, this work implements a cache hierarchy simulation system written in ADL and integrates it into the FAST superscalar description and extends the previous simulators. This cache simulation

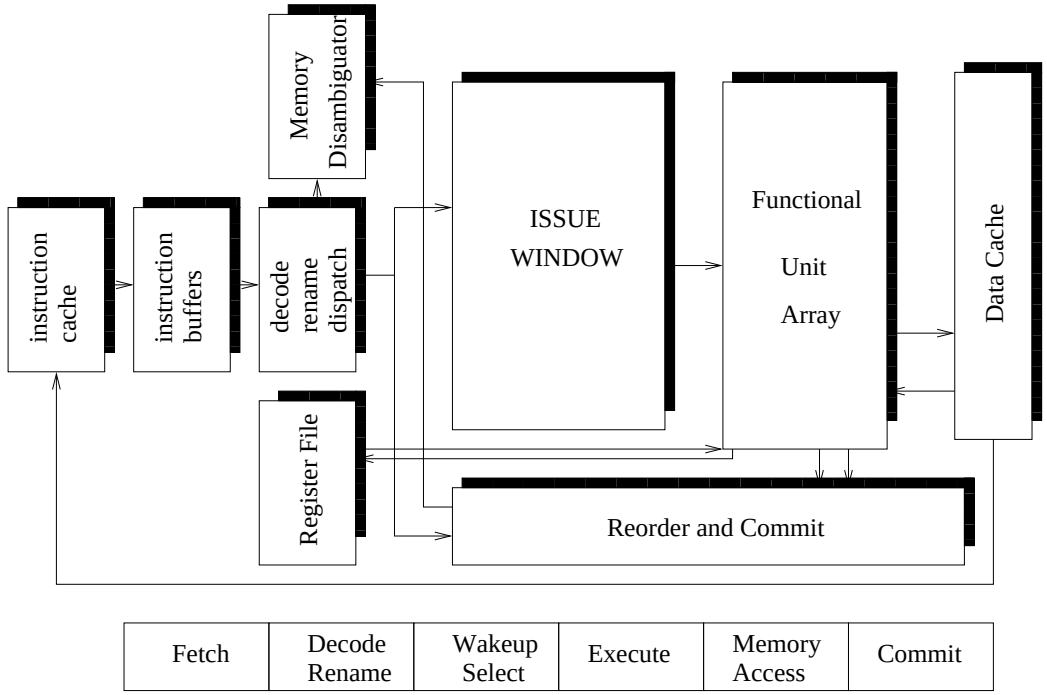


Figure 3.3. Superscalar Simulator

system supports multiple levels of caches, either blocking or non-blocking, from direct-mapped to n-way associativity. It also supports different replacement policies, including FIFO, LRU, and RANDOM. Moreover, it actually moves data from and to the CPU cores. Unlike the Dinero IV [15] cache simulator which was tried to be integrated with FAST, it can generate not only the hit and miss information but also the accurate latency information.

Shown in Listing 3.1, each level of the cache is defined as a *Cache Artifact*, allowing it easily to be integrated with different simulators.

3.2 Benchmark Suites and Environment

In this work, we use the SPEC CPU2000 V1.2 [49] benchmark programs to evaluate different Fine-grain State Processor designs. SPEC CPU2000 V1.2 benchmark suites were

released by SPEC in 2001, including two sub-suites, CINT2000 and CFP2000. CINT2000 is used to measure and compare the compute-intensive integer performance. It contains 12 applications (11 in C and 1 in C++). CFP2000 is used to measure and compare the compute-intensive floating-point performance. It contains 14 applications (6 in Fortran-77, 4 in Fortran-90 and 4 in C).

```
artifact L1_cache
   attributes
      (    size,           # cache size in KB
           bpl,            # cache line size in bytes
           assoc,          # n-way associativity
           repl_policy,    # block replacement policy
           num_ports,      # number of cache ports
           num_mshrs,      # number of outstanding misses
           hit_latency,    # latency to access cache
           bandwidth       # data bus bandwidth
      )
begin
   ...
end
```

Listing 3.1. Cache Artifact

Our experimental hardware consists of a cluster of 20 machines with Intel(R) Xeon(R) CPUs, running the Linux OS. The SPEC CPU2000 benchmark programs are compiled with the GNU GCC cross compiler targeting the MIPS IV instruction set. Some of these benchmarks have not been simulated because of system libraries, e.g., Fortran 90 and C++ benchmarks. Therefore they have been excluded from our experiments. Nine integer programs and eight floating-point programs are used to analyze the FSP performance, which are shown in Listing 3.2 and Listing 3.3, respectively.

Name	Remarks
164.gzip	Data compression utility
175.vpr	FPGA circuit placement and routing
176.gcc	C compiler
181.mcf	Minimum cost network flow solver
186.crafty	Chess program
197.parser	Natural language processing
253.perlbmk	Perl
256.bzip2	Data compression utility
300.twolf	Place and route simulator

Listing 3.2. CINT2000 Benchmarks

Name	Remarks
171.swim	Shallow water modeling
172.mgrid	Multi−grid solver in 3D potential field
173.applu	Parabolic/elliptic partial differential equations
177.mesa	3D Graphics library
179.art	Neural network simulation; adaptive resonance theory
183.equake	Finite element simulation; earthquake modeling
188.ammp	Computational chemistry
301.apsi	Solves problems regarding temperature, wind, velocity and distribution of pollutants

Listing 3.3. CFP2000 Benchmarks

Another important consideration about the simulation is the simulation time. As it is well known, it is really time-consuming and resource-consuming to run an application through an execution-driven cycle-accurate superscalar processor simulator. It can easily take a couple of weeks to emulate a SPEC CPU2000 benchmark running with the default reference input. In this work, in order to evaluate and compare different FPS designs, we

need to run CPU benchmarks using different simulators with many different configurations. It will be impracticable to run them all to completion with the default reference inputs. We therefore choose to run SPEC CPU2000 benchmarks to completion with a *reduced input* [29].

When SPEC CPU95 benchmark suites were integrated into FAST, the reduced input scheme was already applied. Compared to the partial running schemes, it can achieve much more accurate analysis within a reasonable time. In this work, we apply the MinneSPEC [29] reduced workload to emulate SPEC CPU2000 benchmark suites in FAST. MinneSPEC has been officially recognized by SPEC and is distributed with Version 1.2 and higher of SPEC CPU2000 benchmark suites. Although some benchmarks produce very different behavior with the MinneSPEC workload, most match the default reference workload program behavior very closely, in terms of function-level execution patterns, instruction mixes, and memory behavior [29]. From our experiments, MinneSPEC works well in FAST and it satisfies our research requirements.

Chapter 4

Taxonomy of Fine-grain State Processors

In this chapter, we introduce a taxonomy of the fine-grain state processors. It has two main categories, which can be divided further into four sub-categories, shown in Figure 4.1. Based on this taxonomy, those traditional proposals can be summarized and classified into the proper categories.

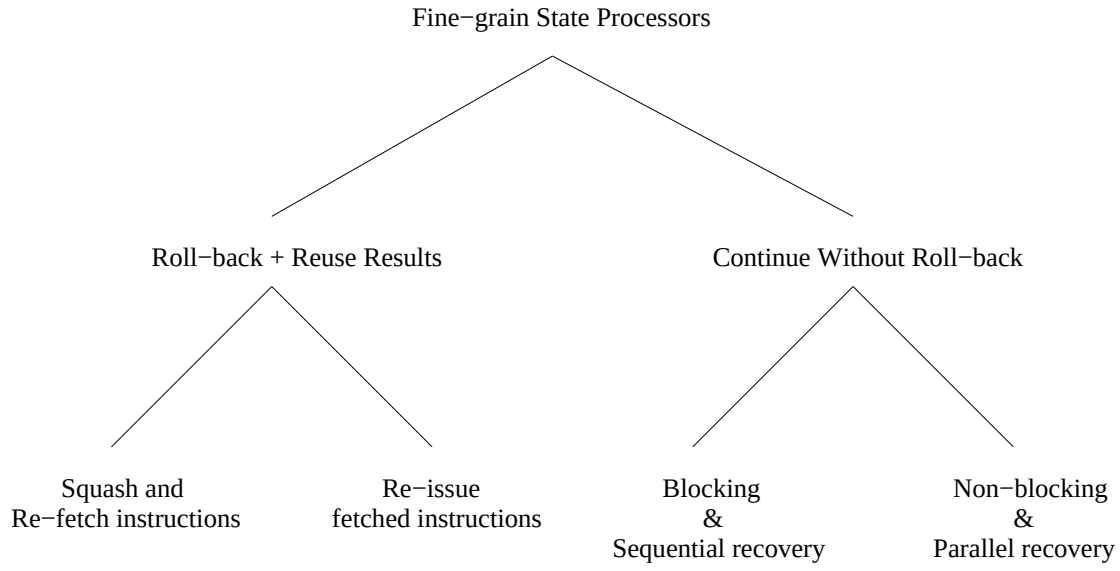


Figure 4.1. Taxonomy of Fine-grain State Processors

4.1 Roll-back + Reuse Results

The first main category is referred to as *Roll-back + Reuse Results*. Utilizing this technique, a processor is able to salvage part of finished work during the speculative execution after a missed speculation. When a mis-speculation happens, the processor rolls back from the exception point and restarts the execution. Those mis-speculation independent instructions may reuse previously generated results and skip different phases of execution (e.g., issue, execute, and result bypass). Thus, performance is improved. Furthermore, it can be divided into two sub-categories: *Squash and Re-fetch instructions* and *Re-issue fetched instructions*.

4.1.1 Squash and Re-fetch Instructions

Utilizing this technique, upon a mis-speculation, processor squashes all instructions following the mis-speculation point. It rolls back to the mis-speculation point and re-fetches all instructions. With the fine-grain state information, fetched instructions will reuse the buffered previous results if they are mis-speculation independent. The previously generated results are explicitly reused.

A typical application of this technique is the recovery of branch mis-predictions. The scenario of control independence gives the opportunity to reuse results based on the fine-grain state concept. A study of the control independence in superscalar processors [43] reported typical performance improvements of 10-30% when micro-architectural implementation issues are realistically modeled to exploit the control independence.

Several mechanisms have been proposed to reuse the results of control independent instructions.

Sodani and Sohi introduced the concept of dynamic instruction reuse in [48]. In their mechanism, the fine-grain state of previously executed instructions are buffered in the Reuse Buffer (RB), including the operands and destination results. Querying RB via the program counter, an instruction can reuse the result of the previous instance if the operands are the same. Their initial goal was to reduce the branch mis-prediction penalty by reusing control independent instructions. Interestingly, this concept can be extended to the general reuse.

Roth and Sohi proposed the register integration mechanism in [44] to reuse the results. Upon a mis-speculation, instructions after the mis-speculation point are squashed in the reverse order. Instead of recycling the allocated physical registers, register mappings of each instruction's operands and results are entered into the Integration Table (IT). Any new instruction will index IT using its PC to check if the register mappings of the operands match. If so, it can reuse the previous instance's destination physical register. Reusing is achieved through the register renaming, and no additional values are read from or written to the physical register file.

Chou *et al.* [10] presented the concept of dynamic control independence (DCI) to implement the reuse. A shadow copy of the Reorder Buffer, DCI buffer, is used to remember the state of recently fetched instruction. After a branch mis-prediction, all instructions following the mis-predicted branch are flushed from the ROB but they remain

in the DCI buffer. When a new instruction is fetched from the correct path, it associatively searches the DCI buffer. It can reuse its previous state stored in the DCI buffer if it proves to be control and data independent.

Besides the branch speculation, *Squash and Re-fetch instructions* is also utilized for other speculation mechanisms. Mutlu *et al.* [35] evaluated reusing the results of pre-executed instructions in a runahead execution processor. They reported that even an ideal reuse scheme can achieve only 3% improvement. The reason is the results of a small number of instructions pre-executed in runahead mode can be reused.

4.1.2 Re-issue Fetched Instructions

Using the *Re-issue fetched instructions* technique, upon a mis-speculation, the processor still needs to roll back to the mis-speculation point and restart execution. However, it does not need to squash instructions and re-fetch instructions from the cache. The key point is that speculation dependent instructions are kept in the scheduling window even after they are issued. If the speculation is resolved as a miss, the processor can roll back to the mis-speculation point and re-issue only dependent instructions without re-fetching. On the other hand, since the independent instructions were already processed, their results can be implicitly reused.

A typical application of this technique is the recovery of load mis-speculations. When a load mis-speculation is detected, instead of squashing all instructions following the load and re-fetching instructions from the cache, the processor will re-issue instructions dependent on the mis-speculated load. In [20], Pentium 4 processor schedules instructions

dependent on loads assuming that loads would hit in the L1 data cache. If a load misses, a selective recovery mechanism called *replay* is used to wake up and re-issue dependent instructions previously executed using incorrect data.

Gandhi *et al.* [16] utilized this technique and proposed Selective Branch Recovery (SBR) to reuse results of convergence instructions. SBR exploits a frequently occurring type of control independence, called exact convergence, where the mis-predicted path converges exactly at the beginning of the correct path. Thus, upon a mis-prediction, correct instructions from the correct path are in fact already in the pipeline. In this case, the processor can reuse results of data independent convergent instructions and re-issue convergent false data dependent instructions without having to fetch/rename them again.

4.2 Continue Without Roll-back

The second main category of the fine-grain state guided speculation recovery technique, *Continue Without Roll-back*, utilizes the fine-grain state concept more efficiently than *Roll-back + Reuse Results*. It is able to continue execution without rolling back to the mis-speculation point. It can also be divided into two sub-categories: *Sequential recovery* and *Parallel Recovery*.

4.2.1 Sequential Recovery

With the *Sequential recovery* scheme, the processor stops moving forward when a speculation is resolved as a miss. The miss dependent instructions, which should have been already buffered on on-chip structures, *e.g.*, *slicing buffer,* are re-executed to repair the

state. Once recovery is processed, the processor achieves a correct state at the resolution point and it can move forward.

Sarangi *et al.* [45] applied this technique in the context of Thread-Level Speculation (TLS) and proposed a novel architecture, *ReSlice*. If a value mis-prediction is declared, only the speculatively-retired instructions depended on the mis-predicted value, *Forward Slice*, are re-executed to restore the damaged state. Once the damaged state is repaired and merged into the processor state, processor is able to resume execution at the mis-speculation resolution point.

The *Sequential recovery* technique is also applied in the context of data cache miss speculation. In [51], Srinivasan *et al.* proposed the continual flow pipelines (CFP) model. Once a load instruction misses in the L2 data cache, the miss-dependent instructions (slices) are drained out of the pipeline into a slice buffer. In the meantime, CFP executes independent instructions. After the L2-miss is serviced, new front-end instructions wait until the slice instructions are inserted back into the pipeline to construct the correct state.

4.2.2 Parallel Recovery

Although the *Sequential recovery* scheme is an efficient fine-grain state based technique, it does not fully exploit the power of the fine-grain state concept. That is, at the mis-speculation resolution point, the execution is blocked needlessly until the recovery is done. Because of the blocking, the parallelism available during the recovery is not exploited. Since an FSP is aware of the state at the individual basis, it can continue processing seamlessly with a partially correct state, before the whole state is repaired. In parallel

with the recovery, the processor can move forward and execute new instructions.

This method is referred to as the *Parallel recovery* technique because it is able to exploit the parallelism-in-recovery. Parallel recovery hence is capable of utilizing the full power of the fine-grain state concept. As described in Chapter 1, our research goal is to design such an FSP model and apply it in the context of the control speculation and the runahead speculation. Parallel recovery is one of the most important contributions of this dissertation.

4.3 Summary of Taxonomy

In this chapter, a taxonomy of the fine-grain state maintenance and recovery is described. The concept of the fine-grain state have been utilized for the control speculation, the value speculation, the load speculation, and the thread level speculation. Based on the introduced taxonomy, those proposals are classified from the simple *Re-use results* model, to the *Sequential recovery* model. The most efficient model, *Parallel recovery*, is able to exploit the full power of the fine-grain state concept. In the next chapter, a general framework for the *Parallel recovery* FSP model is presented.

Chapter 5

Fine-grain State Processor

In this chapter, a general framework for FSP is introduced. This framework is comprised of the five properties described in Chapter 1.2. We discuss the design space of each property and compare FSP and CSP regarding to the control speculation and the value speculation.

5.1 A General FSP Framework

The essential idea behind an FSP is that it breaks the atomic state set into finer granularity at the individual value level. Breaking up the state in this manner allows treatment of the recovery process efficiently. After a mis-speculation, mis-speculation dependent values, in either registers or memory locations, are invalid and need to be repaired. On the other hand, mis-speculation independent values are immediately available and can be used as necessary. This separation enables the processor to continue execution with a partially correct state and still maintain correct program semantics, while the damaged part of the state is being repaired. Ideally, the latency of recovery can be fully hidden by useful execution so that a zero-penalty speculation can be realized. To implement such an FSP, a general high-level framework should implement the five properties described in Chapter

1.2.

1. Identification property: First of all, a mechanism is needed to precisely identify in-order or speculative state on an individual location basis. A common method is to use the *tag* scheme. A tag, normally one bit, is associated with each value location to indicate if it is speculative or not. For the register values, the speculative bits can be appended to the register file, or the register renaming table, or both. For the memory values, the speculative bits can also be attached with the entries in the store queue, or with the memory words/lines in the memory hierarchy system. Since the memory size normally is large, an alternative method is to keep the individual in-order history values in a backward list [21] while the memory hierarchy system presents the whole architectural state.

Although setting and propagating tags is normally straightforward, it is slightly different between the two kinds of speculations, the value speculation and the control speculation. In case of value speculation, the destination of the first value speculative instruction is set as the seed. Then its speculative bit is propagated through the dependence chain. Any instruction accessing a speculative value will set its destination as speculative as well. On the other hand, if all the operands of a producer instruction are non-speculative, its destination's tag is reset.

In case of control speculation, there are two scenarios. If the branch convergence is not considered, then all instructions along the speculative path, from the point of the branch instruction to the resolution point, are treated as speculative. If the convergence is considered, instructions after the convergence point are control independent. However,

some of them still belong to the control-speculation if they access values which are modified through the speculative path. Like in the value speculation, the data dependence tracking can be done by means of propagating the speculative bits.

2. Block and shelve property: With the identification property, after a speculation is resolved as a miss, FSP is aware of which values are speculative and need to be repaired. New fetched instructions which access damaged values have to be blocked and shelved until the values are corrected.

A simple method is to block those instructions in the reservation stations (RS). When a new instruction is decoded, it is marked as operand-not-ready and blocked in RS if any speculative tag of its operands is set. This method demands little modification to the superscalar pipelines. Our proposal of EMR utilizes this *blocking in RSs* method, described in Section 6. However, it may prevent FSP from exploring the far-flung ILP and degrade the performance significantly because blocking dependent instructions in the pipeline will consume critical resources.

An alternative method to blocking is to pseudo-retire dependent instructions from the pipeline and then release resources to execute further independent instructions. In this manner, critical resources can be released and used. The drained dependent instructions will be re-fetched and re-executed once the damaged values on which they depend are restored. *Draining* technique is an attractive choice if the blockage of critical resources is crucial for the performance. More important, by doing so opens up an opportunity for FSP to utilize idle thread/core resources in a multi-thread/core environment.

For instance, in the case of a value mis-speculation, instructions which were executed with wrong operand values before the resolution point need to be re-executed to restore the state. If the draining method is utilized, drained instructions can be sent to an idle thread. This thread is responsible for executing only dependent instructions to maintain the correct state, while the original thread still keeps processing. In Section 7, an FSG-RA model is proposed which utilizes this method for the runahead execution in an SMT environment.

3. Correction property: After a mis-speculation is detected, FSP needs to restore damaged values during the speculative execution. Generally, there are two scenarios. If a correct in-order version of a damaged data location is available, e.g., in a checkpointed register file or in a history buffer, then FSP can copy the correct value into the data location to restore it. If no such an in-order value saved in somewhere, then FSP needs to re-execute the latest producer instruction to generate the correct value for this destination location, as soon as needed operands become ready. Typically, FSP will utilize the former method regarding to the control speculation, and the latter method regarding to the value speculation. Note that both methods are processed on the individual location basis, not on an atomic state set basis. Once a single damaged value is repaired, it is available and all blocked instructions depending on it can execute.

4. Unblocking property: After damaged values are restored, the process of unblocking and executing shelved dependent instructions is straightforward. If dependent instructions are blocked in RS, they can be woken up and selected via the conventional wake-up logic in RS automatically. If the instructions had been drained from the pipeline, they will be re-

fetched and re-executed once the damaged values on which they are depended are repaired. Note if a second thread/core is forked to maintain the state, it may fetch drained instructions immediately, before the necessary values have been restored, to shorten the pipeline filling latency.

5. Parallelism-in-recovery property: Upon a mis-speculation, FSP can continue execution with a partially correct state while the damaged values are being repaired. Thus, it can explore the parallelism in speculation recovery.

The parallelism-in-recovery can be achieved in either a uni-thread environment or a multi-thread environment. Under a uni-thread environment, FSP can interleave the recovery with the execution of independent instructions. In this manner, available resources which would otherwise be idle can be used for recovery. Thus, processor's resources will be utilized more efficiently. We refer to such recovery schemes as *implicit parallel recovery* techniques. In a multi-core environment, an idle thread/core can be forked exclusively to repair the state while the original thread/core is executing new instructions. As a result, a single-threaded program can obtain better performance by utilizing multi-threading resources. It makes the fine-grain state design as an interesting extension to the current multi-core design trend. We refer to such recovery mechanisms as *explicit parallel recovery* techniques. In this work, both implicit and explicit parallel recovery techniques have been implemented and demonstrated in proposed EMR and FSG-RA designs, respectively.

5.2 Coarse-grain State VS. Fine-grain State

Figure 5.1 depicts the difference between a CSP and an FSP with respect to the control speculation recovery. In this figure, a branch instruction is followed by block B along the wrong path and block C along the correct path. Both paths eventually converge at block D, which is control independent. As a reference, an ideal speculation processor would make the correct prediction at t_1, and execute block C and D, respectively. Timeline (b) through timeline (e2) show different recovery techniques for an incorrect prediction which resolves at t_2. In timeline (b), a processor with the ideal recovery scheme can immediately restart from the correct path once the speculation is detected as a miss at t_2, if a zero-latency recovery is supported.

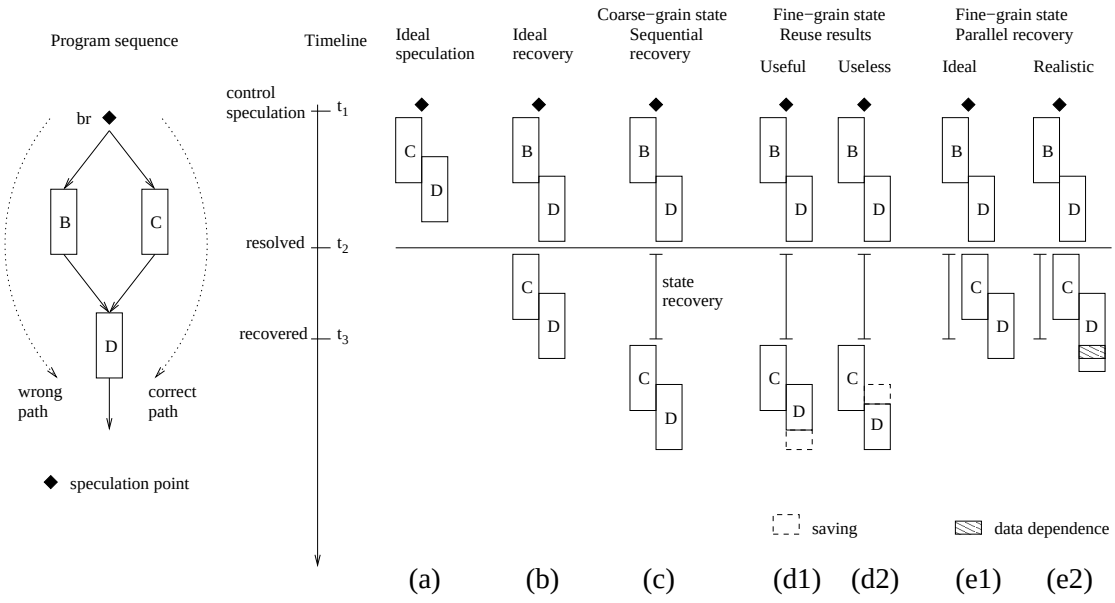


Figure 5.1. Control Speculation

Timeline (c) shows a CSP that has to sequentially restore the state by t_3. Either a

retirement map table method, RAMP [20], or a checkpointing method can be used. A CSP cannot execute any instructions from block C until the whole state has been fully restored (albeit the time between the detection of the mis-prediction and resumption being considerably shorter in case of checkpointing). In Timeline (d1), a processor which is aware of the fine-grain state is able to improve the performance by salvaging part of block D when it restarts from the mis-speculation point. However, if the salvaged instructions are not on the critical path or they are already hidden within the generic ILP, shown in Timeline (d2), re-using will provide little benefit. In contrast, shown in Timeline (e1), an FSP can restore the damaged state in parallel with the execution of block C and D. Ideally, the latency of recovery can be fully overlapped with the useful execution. In this case, FSP will achieve the same performance as that of ideal recovery. In reality, however, it is hard to obtain the same performance of an ideal recovery processor due to the data dependences. A realistic FSP model is illustrated in Timeline (e2).

Similar to the above control speculation example, a value speculation example is illustrated in Figure 5.2. In this figure, a lead producer instruction, A, is followed by a number of independent (empty circles) and dependent (full circles) instructions. Assuming one instruction per cycle, a speculation for the lead instruction is made at t_1 and resolved at t_2. Timeline (a) shows an ideal speculation processor that never misses; it will continue the execution from instruction B and reach C at t_3. If the speculation is wrong, the state at point of B is damaged and it needs to be restored. Timeline (b) through timeline (e) show the different recovery techniques. In timeline (b), a CSP has to roll-back to A and re-execute all

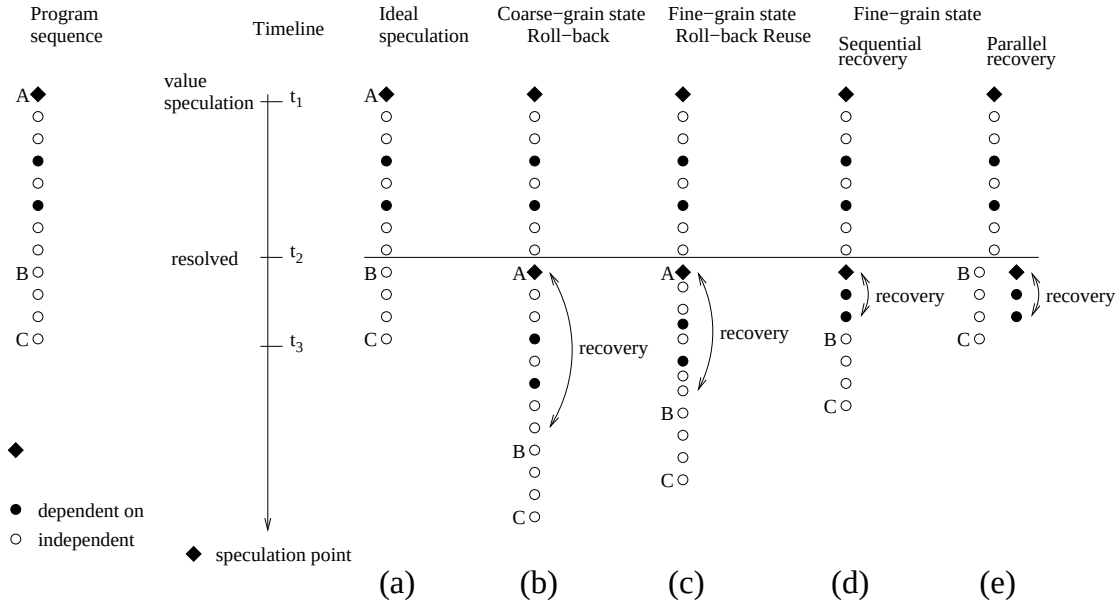


Figure 5.2. Value Speculation

instructions until B to repair the state. In this case, all instructions between A and B, will be executed twice, and the execution of independent instructions during the speculation phase will be wasted. In Timeline (c), a processor which is aware of the fine-grain state is able to avoid wasting finished work. It can reuse independent instructions' results generated in the speculation pass and shorten the recovery process. However, similar to the control speculation case, salvaging may achieve little improvement if those instructions are not on the critical path.

A better implementation of the fine-grain state would be to re-execute only dependent instructions, shown in Timeline (d). During the speculation phase, dependent ones are inserted into a *slicing buffer*. At resolution point t_2, the processor restores the correct state via re-executing only dependent instructions. *Slicing* was first introduced by Weiser [57]

and it used to recover the state in [51, 45]. Although Slicing outperforms salvaging, it still has to restore the state first and then restart from point B. In contrast, a fully implemented FSP can seamlessly process independent instructions crossing B, which access only correct values, while the damaged state are being repaired. Shown in Timeline (e), from t_2, the parallelism-in-recovery is exposed. If there is enough independent work, FSP is able to obtain a zero-latency recovery.

The above two figures illustrate the high-level view of the FSP properties for the control speculation and the value speculation. In the next two sections, based on the framework, two detailed realistic models are implemented for control speculation recovery and value speculation recovery, respectively.

5.3 Summary of FSP's Framework

A general framework of FSP and its design space is introduced in this chapter. An FSP based on this framework will satisfy the five essential FSP properties and implement the *Parallel recovery* mode. The comparison between FSP and CSP with respect to the control speculation and the value speculation is also discussed, as a timeline view.

Based on the presented general FSP framework, we propose two novel FSP models to speed up the branch mis-prediction recovery and the runahead execution recovery. Both models achieve much higher performance via exploring parallelism-in-recovery, compared to the traditional CFP models.

Chapter 6

Eager branch Mis-prediction Recovery

In this chapter, we apply the FSP concept in the field of control speculative execution. We demonstrate that such an FSP can break the limitation of CSP and exploit the performance potential which exists in recovery from control mis-speculations. The application of the FSP concept in control speculation is called Eager branch Mis-prediction Recovery (EMR). By exploring parallelism in recovery, EMR obtains almost the same performance with a machine utilizing unlimited checkpoints. We also show that this technique uses on-chip checkpoint buffers more effectively.

6.1 Introduction

In modern out-of-order processors, the branch mis-prediction penalty becomes a critical factor in overall performance. To recover from the mis-prediction, a traditional CSP either utilizes checkpointing at branches for faster recovery, or sequentially rolls back to the in-order state by waiting until the mis-predicted branch reaches the head of the reorder buffer. However, both methods treat the processor state as a unique set at a coarse-grain level.

A checkpointing [21] mechanism would either have to dedicate a large fraction of the

chip area for checkpoint data, or limit the number of in-flight instructions, in essence limiting the amount of instruction level parallelism that can be exploited. For example, the MIPS R10000 allows only 4 pending branches to be in-flight since its branch stack has only 4 entries, in which each entry contains a complete copy of the integer and floating-point RAM-structured map tables [58]. The Alpha 21264 [13], which uses the CAM-structured map table, supports up to 80 checkpoints, in essence providing the capability to recover the state associated with any of the 80 in-flight instructions. However, as discussed in Chapter 2.4.2, the CAM-structure itself may not scale well since higher degrees of ILP with increased issue widths require a large number of physical registers.

A commonly employed mechanism of sequential recovery is to use a retirement map table called RMAP [20]. If a RMAP is used, when an instruction retires, it updates the retirement map table to indicate that the result register is in the in-order state. The retirement logic ensures that exceptions occur only if the operation causing the exception is the oldest, non-retired operation in the machine [20]. At this point, the retirement state is also the in-order state of this exception point. If a mis-prediction occurs, the processor restores the architectural state, or the front-end map table, from the retirement map table. Although RMAP needs only one extra map table, its recovery takes longer than necessary as renaming cannot start until all instructions prior to the mis-predicted branch retire. If a long latency operation prior to the branch exists, *e.g.*, a cache miss, the mis-prediction penalty increases significantly. Akkary, *et al.*, discuss some optimizations when using a retirement map table [1, 2]. These optimizations walk through the reorder buffer to restore

the map table without waiting for all prior instructions to retire. However, the front-end stalls and instruction processing occurs sequentially, giving an $O(n)$ complexity where n is the number of instructions in-flight. Since more instructions appear in a processor with a large instruction window, this results in a significant increase in the mis-prediction penalty.

As we can see, due to the limitation of coarse-grain state handling, neither of above mechanisms allows resumption of the fetching and renaming of new instructions from the correct path until a known processor state is restored. As a result, the opportunity to utilize the correct values within the architectural state is lost.

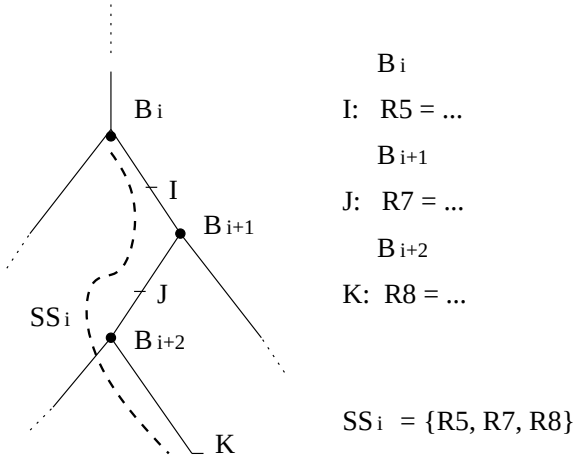
In this chapter, we develop an FSP model based on the framework described in Chapter 5.1 to explore this opportunity that has not been previously considered. EMR *allows fetching and renaming instructions immediately upon a branch mis-prediction* and restores the state to the correct state *as the instruction fetching/renaming continues*. In other words, it effectively explores the parallelism in branch mis-prediction recovery and hides the state recovery latency in a RAM based map table design. The design space of EMR is described in the following section.

6.2 Design Space

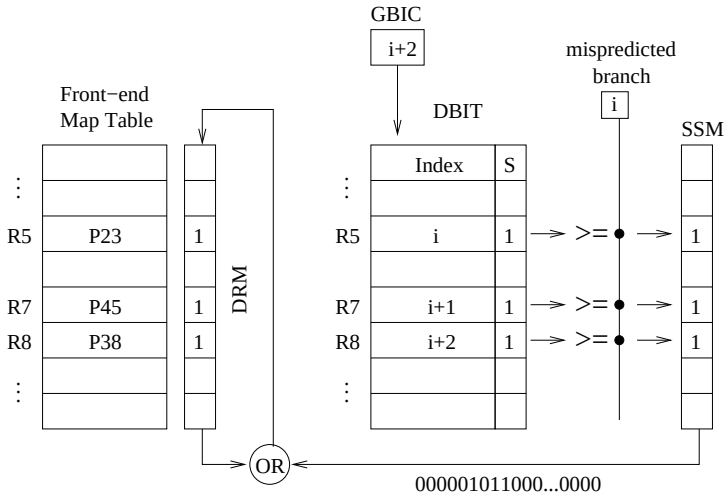
6.2.1 Identifying Speculative State

When a branch mis-prediction occurs, the set of all registers that are defined on the speculative path comprises the speculative state. Note that no speculative memory data is written into the cache/memory during the branch speculation. The memory hierarchy

system always keeps the in-order state. As a result, EMR only needs to identify the speculative register values.



(a) Speculative State



(b) State Masks

Figure 6.1. Identifying Speculative State

To identify the speculative register state, EMR maintains a Global Branch Index Counter (GBIC) for branches and a Dependent Branch Index Table (DBIT) for logical registers. The GBIC records the index of the youngest in-flight branch. When a branch is decoded, the GBIC is incremented by 1 and assigned to it. The DBIT is indexed by the logical register

number, which includes two fields: one is the speculative (S) bit; the other is the branch index field. Initially, all S bits are reset. When a producer instruction is decoded, the current GBIC value is copied into the corresponding entry of its destination in the DBIT, and the S-bit is set. That indicates the destination register is speculative and it is dependent on the current youngest branch. When a producer instruction retires, if it is still the latest definition of its logical destination, the corresponding S-bit of the logical destination register is reset since it is in-order now and it does not depend on any branch. The DBIT can be accessed in parallel with the front-end map table, therefore it will not increase the cycle time of the decoding stage.

When a branch is mis-predicted, in the DBIT entries, those registers whose S-bit is set and the index value is greater than or equal to this branch index are defined on its speculative path. They make up the speculative state for the mis-predicted branch.

The index counter needs $\log_2 N$ bits if the maximum number of branches allowed to be in-flight is N. Since the counter zeroes when it overflows, an extra *color bit* is needed to handle the relative order of branches correctly. Once the counter overflows and zeroes, the *color bit* is flipped, from 0 to 1 or from 1 to 0. Each index is assigned both the counter value and the color bit. When two indices A and B are compared: A is greater than B if A's value is greater than B's and both color bits are the same. Or, A is greater than B if A's value is less than B's and their color bits are different. As a result, the GBIC and each branch index field in the DBIT need $\log_2 N + 1$ bits.

Figure 6.1 illustrates the speculative state identification process. The speculative state is

represented by a mask of registers, called Speculative State Mask (SSM). Suppose when a mis-prediction occurs on the branch B_i, the branches B_{i+1} and B_{i+2}, and the producer instructions *I*, *J* and *K* have already been fetched and decoded speculatively, as shown in Figure 6.1(a). Figure 6.1(b) shows that the processor generates the SSM of B_i by comparing its index *i* with index values in the DBIT entries whose S bits are set. If a register's index value is greater than or equal to *i*, with respect to the circular order, then it is in the speculative state set of B_i, and the corresponding bit in the SSM is set. In this example, *R*5, *R*7 and *R*8 comprise the speculative state set of B_i. They are damaged and not available as the operands for subsequent instructions until the correct values are restored.

The speculative state represents exactly what registers need to be restored. Specifically, the recovery process only needs to recover damaged registers contained in the SSM. If the speculative state only contains a few registers, the recovery process will be effectively hidden by the execution of useful instructions from the correct path.

Our experimental results show that on an average the speculative state upon mis-predictions for 17 SPEC2000 benchmarks accounts for around 20% of the architectural state. We obtain these results by running the benchmarks on our baseline micro-architecture model presented in Section 6.4.

6.2.2 Handling Multiple Mis-predictions

Since EMR continues execution in parallel with recovery, new mis-predictions may occur before the current one is fully restored. Multiple mis-predictions may be in-flight. To handle this situation, we use a global Damaged Register Mask (DRM), that is visible

to new instructions, as shown in Figure 6.1(b). Once the speculative state is identified, we combine it with the DRM, $DRM = DRM \vee SSM$, to reflect the new global speculative state.

In addition, EMR needs to record the precise fine-grain state information for each individual mis-prediction. Upon a mis-prediction, EMR creates a copy of the current front-end map table and the SSM. The combination is called the Mis-predictions Map Table (MMAP), which has two fields, the Mapping Tag and the Speculative bit, as shown in Figure 6.2(a). The Speculative bit decides whether the corresponding logical register is damaged or not. The mapping tag shows the renaming register to which the correct value needs to be restored.

An N-entry circular queue of checkpoints is needed if as many as N pending mis-predictions are allowed to be in-flight. Shown in Figure 6.2(b), the checkpoint queue is maintained as a circular buffer. The head pointer refers the first checkpoint, and the tail pointer always points the next free entry. Upon a mis-prediction, the MMAP of it is created and inserted into the tail of the checkpoint queue. When the first pending mis-prediction is recovered, the head pointer moves to the next entry towards the tail pointer and its allocated checkpoint entry is released. To maintain multiple mis-predictions, we need to consider three cases when a mis-prediction occurs:

Case 1 No pending mis-prediction exists. Since the current mis-prediction is the only one in the pipeline, $DRM = 0$. Let B_f be the mis-predicted branch. SSM is generated as described in Section 6.2.1, which represents the speculative state set of B_f. In this case, the DRM is set to the SSM since the set of damaged registers consists only of

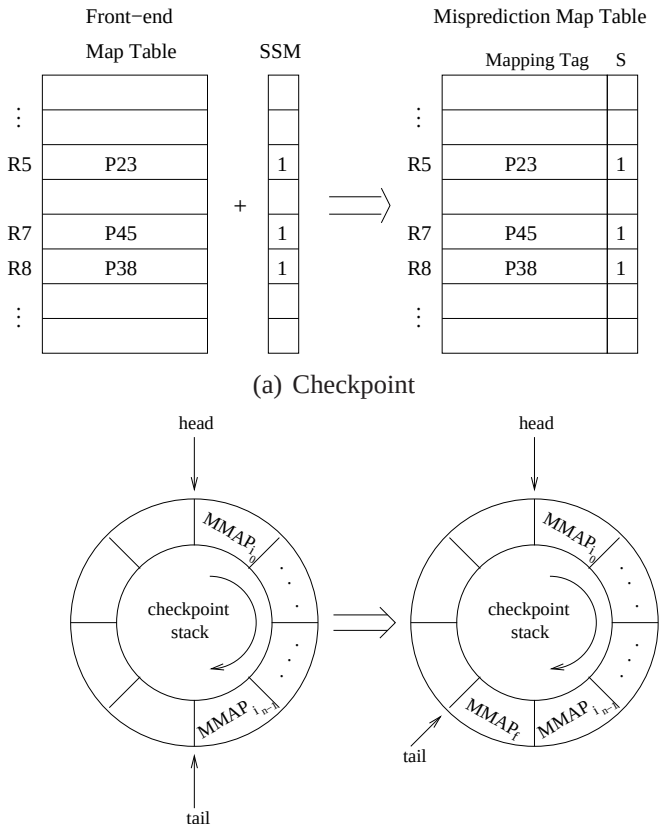


Figure 6.2. Checkpointing to Handle Multiple Mis-predictions

those on the speculative path of B_f. Thus,

$$DRM = DRM \vee SSM = SS_f \tag{6.1}$$

The current front-end map table and the SSM are copied into MMAP_f and MMAP_f is inserted into the checkpoint queue as shown in Figure 6.3(a). B_f is the only mis-prediction in-flight.

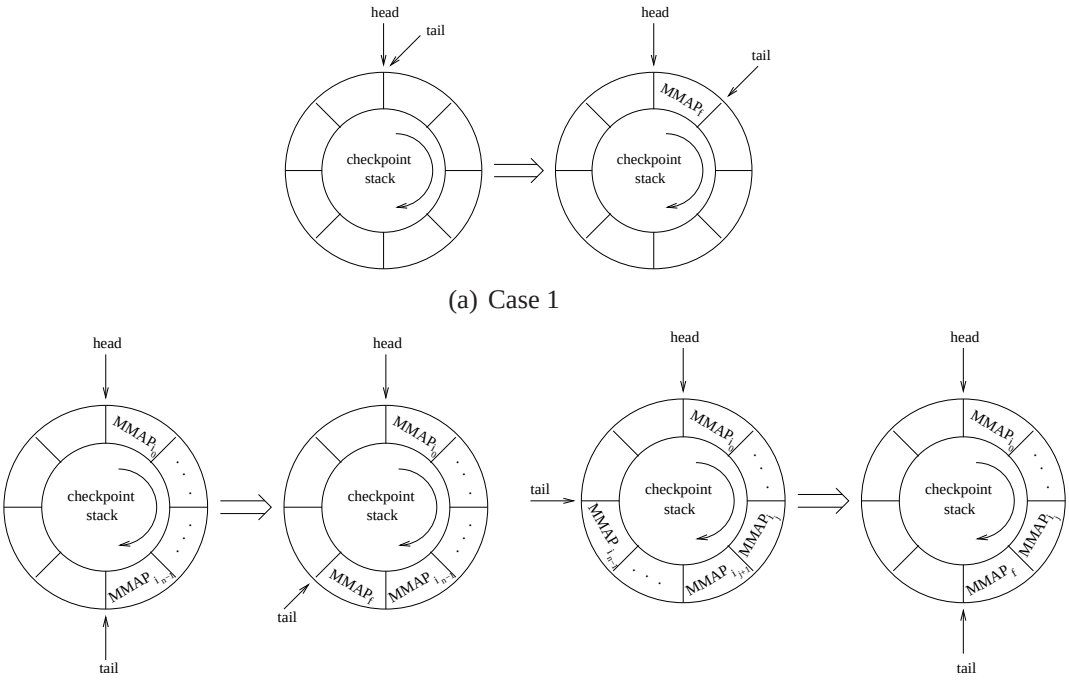
Case 2 A mis-prediction occurs while the processor recovers from *n* earlier mis-predictions.

In this situation, the younger branch, B_f occurs while the processor is recovering

from n previously mis-predicted branches, $B_{i_0} \ldots B_{i_{n-1}}$. Here $i_0 < \ldots < i_{n-1} < f$. Then the DRM will be the union of the speculative state of B_f, SS_f in the SSM, and the speculative states of $B_{i_0} \ldots B_{i_{n-1}}$, which are $SS_{i_0} \ldots SS_{i_{n-1}}$. Since $SS_{i_0} \ldots SS_{i_{n-1}}$ have already been generated and are contained in the DRM,

$$DRM = DRM \vee SSM = SS_{i_0} \vee \ldots \vee SS_{i_{n-1}} \vee SS_f \tag{6.2}$$

In Figure 6.3(b), the copy of the current front-end map table and the SSM are copied into MMAP_f and MMAP_f is inserted into the checkpoint queue. There are $n+1$ mis-predictions in flight after B_f is mis-predicted.



(a) Case 1

(b) Case 2

(c) Case 3

Figure 6.3. Three Cases of Multiple Mis-predictions

Case 3 A mis-prediction occurs while n mis-predictions are in-flight. Assume that the

processor is recovering from n mis-predicted branches, B_{i_0} ... $B_{i_{n-1}}$. The DRM contains the union of $SS_{i_0} \ldots SS_{i_{n-1}}$. Since branches can be resolved out-of-order, it is possible the new mis-prediction is detected on a branch B_f, which is younger than B_{i_j} and older than $B_{i_{j+1}}$. Here $i_j < f < i_{j+1}$, and $i_0 \leq i_{j+1} \leq i_{n-1}$. Obviously, branches from $B_{i_{j+1}}$ to $B_{i_{n-1}}$ are on the speculative path of B_f. Any mis-predictions of $B_{i_{j+1}}$ through $B_{i_{n-1}}$ are false mis-predictions. The speculative state SS_f generated in the SSM contains the speculative states $SS_{i_{j+1}}$ ... $SS_{i_{n-1}}$. Thus,

$$\begin{aligned} DRM &= DRM \vee SSM \\ &= SS_{i_0} \vee \ldots \vee SS_{i_j} \vee SS_{i_{j+1}} \vee \ldots \vee SS_{i_{n-1}} \vee SS_f \\ &= SS_{i_0} \vee \ldots \vee SS_{i_j} \vee SS_f \end{aligned} \tag{6.3}$$

The DRM now represents the new union of the speculative state of B_f and the speculative states of $B_{i_0} \ldots B_{i_j}$. Any false mis-predictions that are caused by invalid branches through the speculative path of B_f are covered by the mis-prediction of B_f. In Figure 6.3(c), the MMAPs for $B_{i_{j+1}} \ldots B_{i_{n-1}}$ are flushed from the checkpoint queue. The MMAP for B_f is created and inserted into the queue.

The DRM always represents the complete set of all damaged registers of multiple in-flight mis-predictions. With the DRM, the processor can easily distinguish those instructions from the correct path that reference any incorrect state.

6.2.3 Blocking and Shelving Dependent Instructions

After the processor identifies the speculative state, it changes the PC to the correct target of the mis-predicted branch and continues the execution. To handle new instructions which may reference values in incorrect speculative state, EMR utilizes the *blocking in RSs* method.

Traditionally, when the processor dispatches an instruction into the instruction window, the ready bit of an operand is set to valid if the operand has already been computed [46]. In our mechanism, if an operand belongs to the in-order state, the ready bit will be set normally, depending on whether this value is computed or not. If it belongs to the incorrect speculative state, the ready bit should be set to *invalid*, even if the value has already been computed. Using the DRM, simple logic is enough to handle both cases: $R = \overline{D}_i \wedge V_j$, where D_i is the i^{th} bit in the DRM, corresponding the i^{th} logical register, R is the operand ready bit and V_j is the value ready bit of the physical register allocated to the i^{th} logical register. If D_i is 0, this operand is not damaged and is ready if the value has already been computed. If D_i is 1, this operand is damaged and is not ready.

During the renaming stage, each producer instruction resets the D-bit of its logical destination in the DRM. Any subsequent instruction which needs that logical register as an operand will reference the new, undamaged state.

When the D-bit is set to 1, instructions that reference the damaged speculative state wait in the reservation stations until the correct state is restored. Instructions that access only undamaged registers proceed without waiting.

Our experimental results show that in the SPEC2000 benchmark suite, on average only 18% and 40% of all instructions reference damaged registers in CFP2000 and CINT2000, respectively. Using EMR, instructions referencing undamaged registers never wait unnecessarily because of a branch mis-prediction as new instructions are fetched and renamed using the current map table values without interruption.

6.2.4 Correcting Incorrect Speculative State

To maintain the program's correctness, EMR needs to repair the speculative state by restoring the correct data from the in-order state. Once the correct state is restored, instructions which reference the damaged state can be executed correctly and correct program semantics is maintained.

Like RMAP, EMR uses a retirement map table to construct the in-order state at the mis-prediction point sequentially through the retire logic. When an instruction retires, it updates the retirement map table to indicate that the result register is in the in-order state. When a mis-predicted branch reaches the head of the reorder buffer, the retirement state is also the in-order state of this mis-predicted branch. EMR records the speculative state in the MMAP when a mis-prediction happens. When the mis-predicted branch causing it reaches the head of the reorder buffer, all previous mis-predictions should have already been recovered. The first entry of the checkpoint queue contains the MMAP of this mis-prediction.

With the retirement map table, RMAP, and the MMAP popped from the checkpoint queue, EMR can restore the correct data from the in-order state to the speculative state.

Figure 6.4 illustrates the recovery process for the mis-prediction of B_f. The S-bit in the

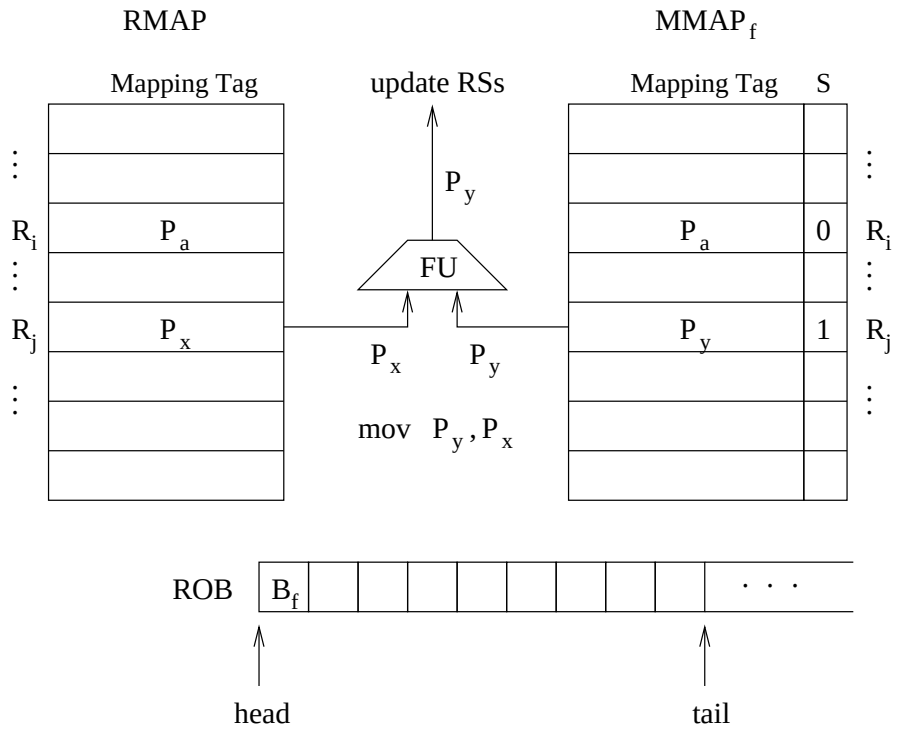


Figure 6.4. Restoring Speculative State

MMAP indicates whether a logical register is in the speculative state or not. For example, the S-bit of R_i is not set indicating that R_i is not damaged through the speculative path of B_f. As a result, the corresponding entries are the same both in the RMAP and the MMAP (P_a). On the contrary, the S-bit of R_j is set indicating it belongs to the speculative state of B_f. The correct data needs to be restored from the in-order state to the speculative state: $P_x \longrightarrow P_y$. After restoring the correct data value, EMR broadcasts the tag of P_y to the reservation stations to wake up blocked instructions that need this value. If P_y is still the latest renaming tag of R_j in the front-end map table, the corresponding D-bit of R_j in the DRM is reset. Doing so will permit subsequent instructions which may access R_j as their operand to continue normally.

After all registers in the speculative state are restored, the recovery of the current mis-prediction is done. At this point, retirement continues normally and the first entry in the checkpoint queue is released.

6.2.5 Parallelism in Recovery

Since EMR is based on a uni-thread model, it utilizes the *implicit parallel recovery* scheme. EMR interleaves the recovery process with the normal execution without increasing the processors' complexity.

Simply stated, EMR issues copy operations of the form *mov* P_y, P_x, into the free functional units to restore the correct data. The copy operations execute as normal instructions as they read from and write to the register file, and wake up dependent instructions blocked in the reservation stations. Moreover, the copy operations update the retirement map table and the DBIT after they restore the data just as normal producer instructions do. Thus, a uniform pipeline design can be used for normal execution and recovery operations. Furthermore, the complexity of the pipeline design does not increase. No extra read/write ports of the register file, and no extra tag buses are needed.

Each cycle, EMR can restore as many damaged registers as there are free functional units. Note that, if there are not many free functional units, this implies that the newly fetched instructions do not reference the damaged state. In contrast, when there are many free functional units, the newly fetched instructions reference the damaged state. In the former case we can afford to be slow in the recovery; in the latter case we can quickly restore values and unblock waiting instructions. In either case, the parallelism between the

recovery and the newly useful instruction execution can be achieved.

6.3 Optimization

So far the fundamental design space of EMR has been discussed. Although EMR allows execution of instructions which do not reference damaged values, it cannot start repairing damaged values before a known in-order state is obtained. This state is obtained by waiting until the mis-predicted branch reaches the head of the reorder buffer and under normal circumstances this may not be a significant problem. However, when the head of the reorder buffer is blocked by a long latency operation such as a cache miss, the time for the mis-predicted branch to reach the head of the reorder buffer may become significant. During this time, the likelihood of finding instructions which do not reference the damaged state will rapidly diminish and the processor will eventually stall.

From the point of view of state, EMR still handles part of the repair process at a coarse-grain level. It has to wait until the whole in-order state at the mis-predicted branch point is fully restored. In order to fully utilize the power of the FSP concept, EMR needs to repair the damaged values as early as possible, at a single value level.

We therefore augment our basic technique with an appropriate variation of WALK algorithms [1, 2]. Both RMAP+WALK and HISTORY+WALK are optimizations on the basic RMAP mechanism and both methods walk through the reorder buffer entries to reconstruct the in-order state without waiting for all instructions prior to the mis-predicted branch to retire. This technique is orthogonal to EMR and EMR can also be improved by incorporating the WALK scheme. We refer to the combined technique as EMR+WALK.

As illustrated in Figure 6.5, the technique requires some additional fields in the MMAP.

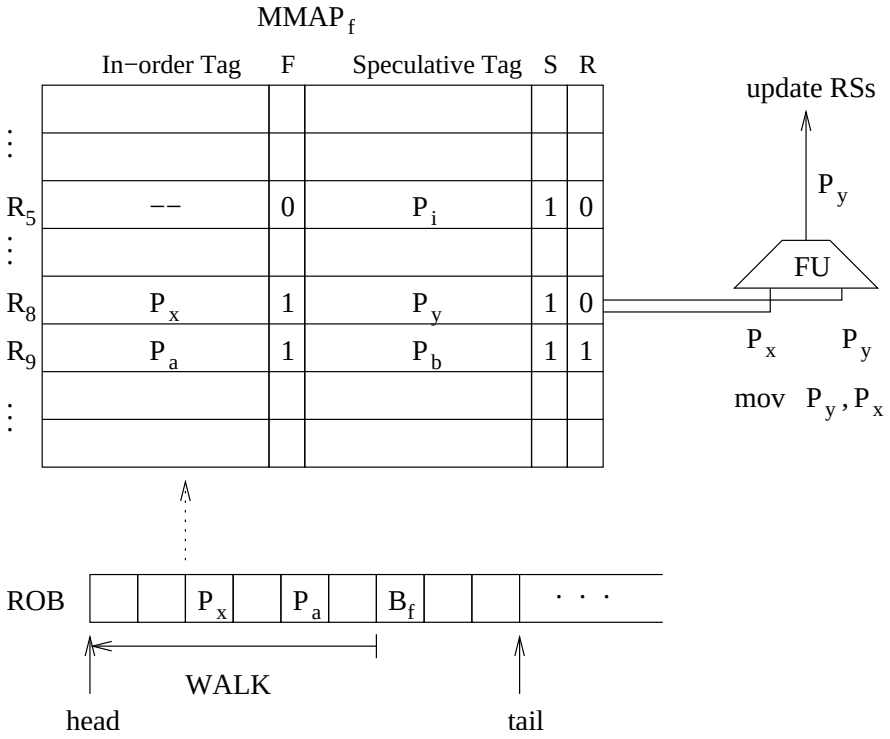


Figure 6.5. EMR+WALK

In order to repair the damaged speculative state in the MMAP, EMR+WALK needs to restore the correct value for each damaged register from the latest definition of the same logical destination prior to the mis-prediction point. EMR+WALK walks from the mis-predicted branch towards the head of the reorder buffer to retrieve the latest definition information from each ROB entry. When a definition ROB entry is scanned, it is the latest definition of the speculative destination register prior to the mis-prediction point if the corresponding *S*-bit (Speculative) is 1 and the *F*-bit (Found) is 0 in the MMAP. If this is the case, the renaming tag of the destination register is put into the *In-order Tag* field and the *F*-bit is set. After EMR+WALK walks to the head of the ROB, if there is any entry with

$S = 1$ and $F = 0$ left, its *In-order Tag* can be retrieved from the retirement map table.

Since this *WALK* process is independent from the retirement logic, restoring correct values can be started as early as possible, without waiting for all instructions prior to the mis-predicted branch to retire. Any entry in the MMAP with $S = 1$, $F = 1$ and $R = 0$ (Recovered) will trigger a move operation: *In-order Tag* $\longrightarrow$ *Speculative Tag*, if the correct value is ready and there is a free functional unit. After the correct value is restored, its *R*-bit is set.

Since there can be multiple mis-predictions in-flight, multiple walk units are needed and the walk process of a younger mis-prediction may cross older ones. All make the implementation complicated. To simplify the implementation, we only allow a simple walk process for the first pending mis-prediction. Once a mis-prediction becomes the oldest one, its walk process and restoring process can start immediately.

6.4 Experimental Evaluation

6.4.1 Experimental Methodology

The simulation tool and the experimental environment are presented in Chapter 3. In order to evaluate the performance of EMR, we have collected results from 17 benchmarks of the SPEC2000 benchmark suite. All benchmarks were run to completion using the reduced reference inputs from the MinneSPEC workload [29]. The cycle-accurate superscalar simulator is used as the baseline. The parameters of the baseline model are shown in Table 6.1. Both load and store instructions are allowed to issue out-of-

order [11, 39] using the store set memory dependence predictor. Five models with different mis-prediction recovery mechanisms have been evaluated and compared:

1. *RMAP*, the traditional coarse-grain sequential recovery model. A retirement map table is used to restore the state.

2. *RMAP+WALK*, the optimization of the above method. With the retirement map table, it walks from the head of ROB towards to the mis-prediction point to restore the state.

3. *EMR (M=4)*, the FSP model which can handle 4 pending mis-predictions.

4. *EMR+WALK (M=4)*, the optimization of EMR, which is combined with the WALK method.

5. *UL_CHK(UNLIMITED CHECKPOINTS)*, the ideal recovery model, in which a checkpoint is made with every branch. It can immediately restore the correct state from the checkpoint when a mis-prediction is detected.

All five machines were kep identical in all aspects except the branch mis-prediction recovery scheme. RMAP+WALK and EMR+WALK both use the resources available to the retirement logic. The walking step assumes the same number of instructions as the retirement width.

6.4.2 Performance Results

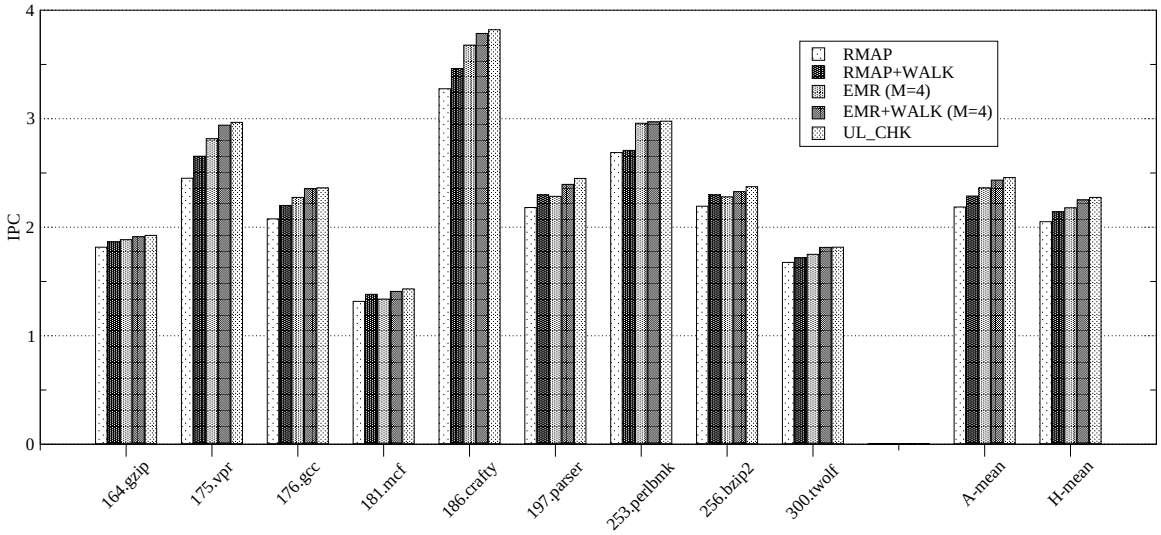
The instructions per cycle (IPC) for each program in the benchmark suite using the 5 recovery models stated previously is shown in Figure 6.6. EMR/+WALK has been

Parameter	Configuration
Issue/Fetch/Retire width	8/8/8
Instruction window size	128
Reorder buffer size	256
Register file entries	256
Functional units	Issue width Symmetric
Branch predictor	16K gshare
BTB	1024-entry
Return-address stack	32-entry
Dcache	L1: 32KB, 4-way, 64B/line, 2 cycles L2: 512KB, 8-way, 64B/line, 10 cycles
Memory	8B/line, 40 cycles first chunk, 4 cycles inter-chunk.

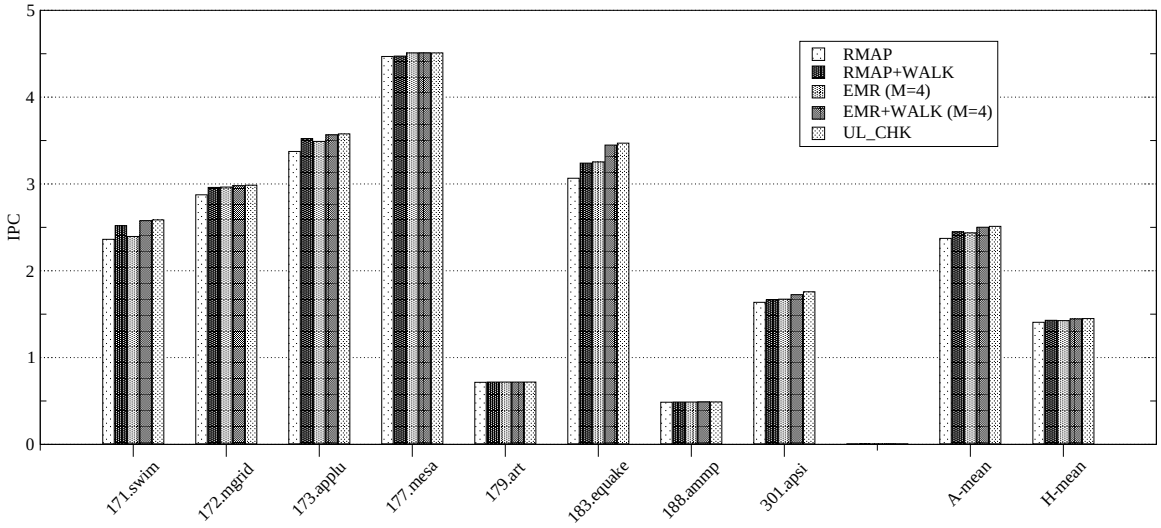
Table 6.1. Machine Configurations

implemented using $M = 4$, handling at most 4 branch mis-predictions simultaneously. We will discuss the selection of different values of M in Section 6.4.3. As can be seen from Figure 6.6, EMR outperforms the traditional RMAP mechanism across all benchmarks, while EMR+WALK performs better than RMAP+WALK. Furthermore, EMR+WALK performs nearly as well as UL_CHK.

To help understand the performance results for the 5 different models, Figure 6.7 illustrates the percent speedup over RMAP of the other four models. As shown in Figure 6.7, RMAP+WALK obtains a 2.9% and 0.4% harmonic mean improvement over RMAP on CINT2000 and CFP2000, respectively. EMR achieves a 4.7% and 1.0% improvement over RMAP. Recall as discussed in Section 6.3, the restoring process of EMR can be delayed significantly due to some long latency operations, such as cache misses or floating point operations. Long latency operations cause EMR to perform worse than RMAP+WALK on several CFP2000 benchmarks, particularly on 181.mcf, which

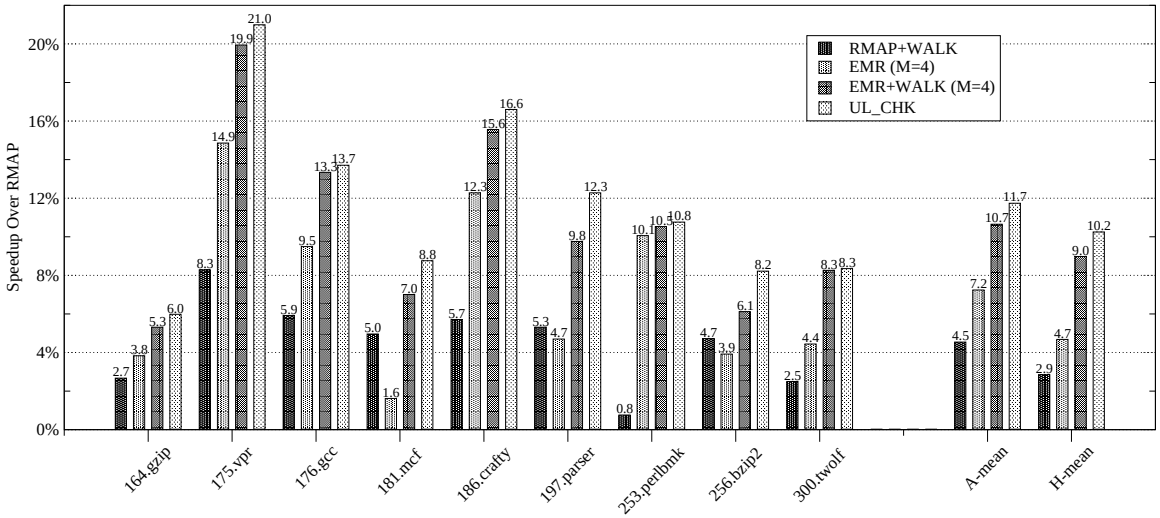


(a) SPEC2000 INT

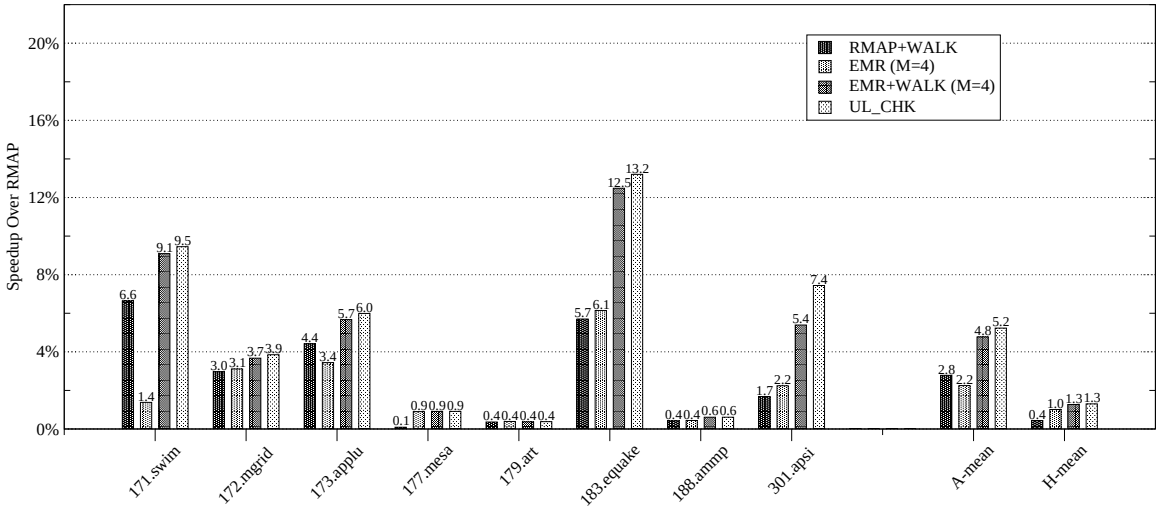


(b) SPEC2000 FP

Figure 6.6. Performance of five models



(a) SPEC2000 INT



(b) SPEC2000 FP

Figure 6.7. Speedup of RMAP+WALK, EMR/+WALK (M=4) and UL_CHK over RMAP

has a high cache miss rate. On the other hand, EMR+WALK appropriately overcomes this shortcoming. As shown in Figure 6.7(a), on the nine benchmarks. As shown in Figure 6.7(a), on the nine with a maximum improvement of 19.9% (175.vpr). The best method, UL_CHK, improves the performance by a harmonic mean of 10.2%. In other words, EMR+WALK achieves $(1 + 9.0\%)/(1 + 10.2\%) = 99\%$ of the harmonic mean performance of UL_CHK.

Although EMR+WALK obtains a lower performance improvement on CFP2000 compared to CINT2000, our technique obtains an arithmetic mean improvement of 4.8% and a harmonic mean improvement of 1.3% on eight CFP2000 benchmarks. As shown in Figure 6.7(b), EMR+WALK achieves the same harmonic mean performance as UL_CHK. In most cases, floating-point programs have relatively better branch prediction accuracies using advanced branch prediction techniques. Therefore, they are less sensitive than integer programs to the mis-prediction recovery mechanisms.

	164.gzip	175.vpr	176.gcc	181.mcf
RMAP	95.31	88.68	85.79	94.92
RMAP+WALK	95.31	88.70	85.78	94.92
EMR	95.28	88.68	85.76	94.94
EMR+WALK	95.29	88.68	85.82	94.98
UL_CHK	95.30	88.70	85.78	94.97

	186.crafty	197.parser	253.perlbmk	256.bzip2	300.twolf
RMAP	89.53	94.21	90.37	94.08	88.54
RMAP+WALK	89.53	94.21	89.84	94.11	88.45
EMR	89.53	94.18	89.84	94.07	88.54
EMR+WALK	89.52	94.19	89.84	94.11	88.38
UL_CHK	89.53	94.20	89.84	94.09	88.46

Table 6.2. CINT2000 Branch Prediction Accuracies(%)

Table 6.2 and Table 6.3 show the branch prediction accuracies of 17 benchmarks in five models with a 16K *gshare* [33] predictor. As it can be seen from the table, almost all floating-point benchmarks have higher prediction accuracy than integer benchmarks. Therefore the possible performance improvement is relatively small in the floating-point programs.

	171.swim	172.mgrid	173.applu	177.mesa
RMAP	96.63	95.43	97.38	99.20
RMAP+WALK	96.63	95.44	97.36	99.20
EMR	96.63	95.44	97.38	99.20
EMR+WALK	96.63	95.44	97.38	99.20
UL_CHK	96.63	95.54	97.37	99.20

	179.art	183.equake	188.ammp	301.apsi
RMAP	98.80	96.86	99.26	94.37
RMAP+WALK	98.80	96.85	99.30	94.45
EMR	98.80	96.82	99.31	94.46
EMR+WALK	98.80	96.83	99.30	94.42
UL_CHK	98.80	96.83	99.30	94.54

Table 6.3. CFP2000 Branch Prediction Accuracies(%)

6.4.3 Mis-predictions-under-Mis-predictions

This section evaluates the performance of EMR/+WALK when the number of allowed outstanding mis-predictions varies. In order to achieve a good trade-off between performance and the hardware cost associated with the mis-prediction checkpoints, EMR implementations need to choose a reasonable value for M. Figure 6.8 illustrates the respective performance of different EMR implementations where the number of mis-prediction maps is varied from $M = 1$ to $M = 16$. We only present the harmonic mean IPC

for the entire suite of SPEC2000 benchmarks and omit the data for individual benchmarks. As it can be seen from the figure, both performance lines of EMR and EMR+WALK have a steep gradient from $M = 1$ to $M = 2$, and at larger values of M, performance improvement levels off. Recall that the front-end is stalled when the i^{th} mis-prediction is detected in EMR/+WALK with $M = i$. When $M = 1$, EMR and EMR+WALK stall fetching new instructions until the current mis-prediction is recovered resulting in poor performance. As the value of M is increased, EMR and EMR+WALK can better hide the latency of state recovery.

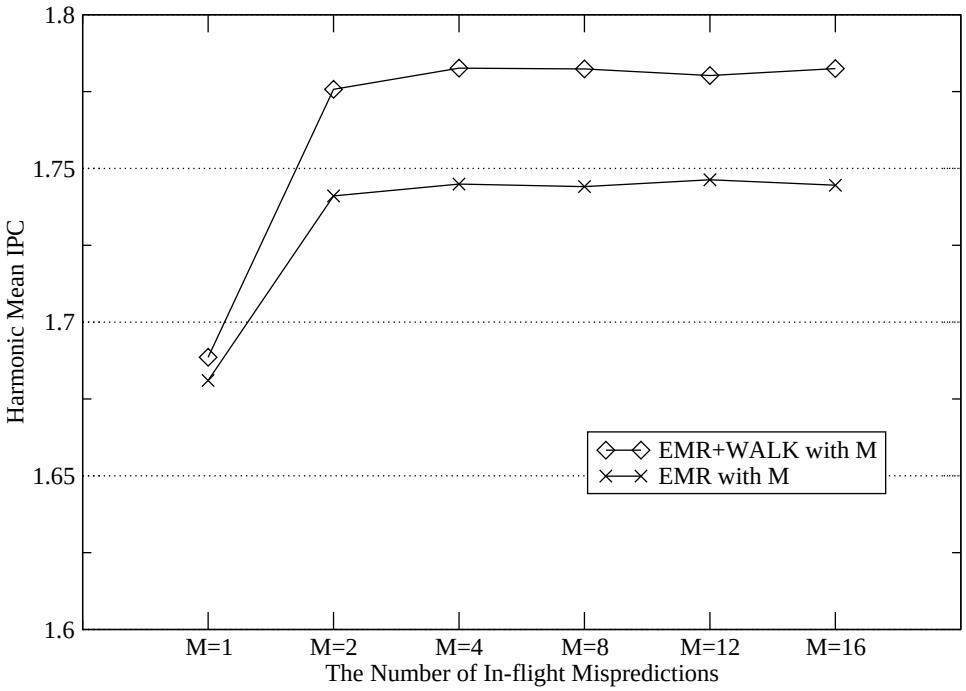


Figure 6.8. Performance of EMR/+WALK with Different M

With a highly accurate branch predictor, the probability of having many mis-predictions in succession diminishes. Under such circumstances allowing many mis-predictions to be in-flight will not provide significant performance improvements. Figure 6.8 shows that

selecting $M = 4$ provides the best trade-off between performance and hardware complexity for both recovery models.

Theoretical analysis verifies the experimental results. In both EMR models, checkpoints are created only upon mis-predictions. If the number of in-flight branches is B, then

$$M = B \times mis-prediction\ rate. \tag{6.4}$$

In our experimental models, the number of total in-flight instructions is 256. Given that a branch is encountered on an average every 3-5 instructions [34], B is around 50. Assume that a gshare predictor, utilized in the experiment, has less than 10% mis-prediction rate, then $M \approx 4$ based on Equation 6.4. Since UL_CHK would need about 50 checkpoints, EMR mechanism roughly requires *mis-prediction rate%* of the hardware cost of the ideal UL_CHK while capturing 99% performance.

6.4.4 Towards a Large Instruction Window

This section studies the performance variation of the five recovery methods when the scheduling window size and the reorder buffer size increase. Figure 6.9 shows the harmonic mean IPCs when the reorder buffer size varies from 64 to 512. To focus the performance study on the mis-prediction recovery mechanism exclusively, the physical register file size is kept idealized in this group of experiments.

As shown in Figure 6.9, all five models obtain performance improvement due to an increased instruction window size. However, the strides of the improvement are not equal. As it can be seen, the performance gap between RMAP and UL_CHK becomes larger as

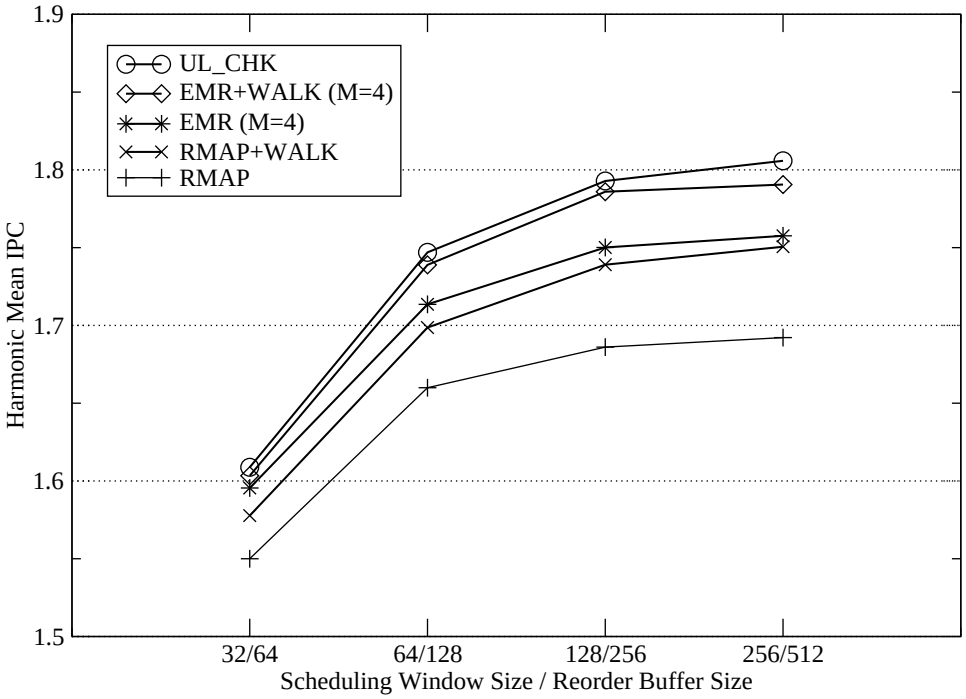


Figure 6.9. Performance of 5 Models with Different SW/ROB sizes

the instruction window size increases. The performance of RMAP reduces from 96% of the performance of UL_CHK down to 93% as the instruction window size is increased from 64 to 512. This phenomenon indicates that mis-prediction recovery becomes more critical for large instruction window processors. In contrast, EMR+WALK always achieves close to 99% of the performance of UL_CHK across all window sizes. We conclude that in general, FSP may lead to more scalable designs than coarse-grain state recovery methods.

6.5 Related Work

In [1, 2], Akkary *et al.* use selective checkpoints at low-confidence branches to recover from branch mis-predictions. Selective checkpointing provides better scalability as the instruction window becomes larger. However, as the size of the instruction

window is increased, the distance between a valid checkpoint and the current instruction pointer increases, which in-turn increases the possibility of re-executing already executed instructions since the confidence estimator cannot be perfect.

Gandhi *et al.* [16] propose Selective Branch Recovery (SBR) to reduce branch mis-prediction penalty by exploiting a frequently occurring type of control flow independence, called exact convergence. The results of some convergent instructions computed on the mis-predicted path can be reused. Thus, the recovery penalty is reduced since convergent instructions do not need to be fetched/renamed again. Non-convergent instructions on the mis-predicted path are re-issued as move operations. Each such move operation copies the value from the previous renaming physical register of its destination to its renaming physical register. Thus the correct value of each logical destination is restored one by one through the definition chain similar to EMR state recovery.

In [4], Aragon *et al.* analyze the performance loss due to branch mis-predictions. They break the mis-prediction penalty into three subcategories: pipeline-fill penalty, window-fill penalty, and serialization penalty. They propose a Dual Path Instruction Processing (DPIP) to reduce the pipeline-fill penalty. In DPIP, a low-confidence branch is forked and both paths are fetched and renamed, however, the alternative path is not executed. A checkpoint of the map table is created upon the low-confidence branch to support the dual path processing. Thus, when a mis-prediction happens, some instructions from the correct path have already been fetched and renamed in the pipeline. DPIP can only fork once since only two active paths are allowed at the same time.

A significant body of research has provided us with increasingly better branch prediction accuracies [59, 33, 50, 9, 23]. Although the type of branch predictor is orthogonal to the EMR technique, EMR will provide diminishing returns as the accuracy of branch prediction increases. Similarly, it provides significant performance benefits as branch predictor accuracy decreases. EMR may tend to blur the differences between different branch predictors and hence may favor less accurate but faster branch predictors.

Armstrong *et al.* [5] propose to reduce performance degradation caused by branch mis-prediction. They propose a mechanism to leverage wrong path events (WPEs), which occur during periods of mis-prediction, such as a NULL pointer memory access. WPEs can be used to detect whether a branch was mis-predicted before it is executed. Thus, the time for detecting mis-prediction is reduced. When a wrong path event occurs, mis-prediction recovery can be initiated early. Utilization of WPEs is orthogonal to EMR.

6.6 Summary of EMR

As the pipeline depth increases, branch mis-prediction becomes a primary bottleneck in obtaining high performance. Utilization of the FSP concept for branch mis-prediction recovery mechanism can significantly reduce the latency of branch mis-predictions by immediately starting to process instructions from the correct target without waiting for the processor state to be restored. Furthermore, the fine-grain processor state can be kept appropriately in checkpoints, and correct values can be forwarded to blocked instructions by using free functional units, resulting in a complexity-effective approach.

As an implementation of the FSP concept, EMR+WALK obtains an average performance

speedup of 9.0% over the traditional technique on CINT2000. Moreover, it achieves 99% of the performance obtained by an unlimited checkpoint recovery method using only 4 checkpoints.

Chapter 7

Fine-grain State Guided Runahead Execution

In this chapter, we apply the FSP concept in the field of value speculative execution by using Runahead execution. We apply the fine-grain state maintenance techniques in a multi-threaded environment and present Fine-grain State Guided Runahead Execution (FSG-RA), an SMT-like FSP model of runahead execution where the data values dependent on a missed load are treated as damaged values. These values are verified and recovered as necessary by an independent thread, while the original thread continues to execute new instructions.

We demonstrate that FSG-RA can improve the single-thread program's performance by exploiting the parallelism in the Runahead execution recovery in a multi-thread processor environment.

7.1 Introduction

Runahead execution was first proposed by Dundas and Mudge [14] for in-order processors and later applied to out-of-order processors by Mutlu *et al.* [37]. It increases the effective instruction window of a processor by continuing execution when the instruction

window is blocked by a long latency operation, *e.g.*, an L2-cache miss load. In this case, the processor enters the "runahead mode" by providing a bogus value for the blocking operation and pseudo-retiring it out of the instruction window. Under the "runahead mode", all the instructions following the blocking operation are fetched, executed, and pseudo-retired from the instruction window. Once the blocking operation completes, the processor rolls back to the point it entered the runahead mode and returns to the "normal mode". Though all instructions and results obtained during the "runahead mode" are discarded, the runahead execution warms up the data cache and significantly enhances the memory level parallelism.

Similar to the general value speculation example depicted in Chapter 5.2, one can envision the runahead execution as a value speculation during which part of the state is damaged. Consider the timelines shown in Figure 7.1. Timeline (b) represents the conventional runahead execution which uses a coarse-grain state recovery method to maintain the state. It has to roll back when the load miss is resolved. Timeline (c) illustrates a runahead processor that reuses the independent results generated during the "runahead". This Roll-back + Reuse method, however, obtains little improvement [35].

On the other hand, an ideal FSG-RA processor can achieve much better performance, because it only needs to re-execute miss-dependent instructions which updated the processor state with incorrect values during the runahead execution. Furthermore, it can execute these instructions in an arbitrary order (pending the dependencies among them) because it is able to continue executing new instructions with a partially correct state.

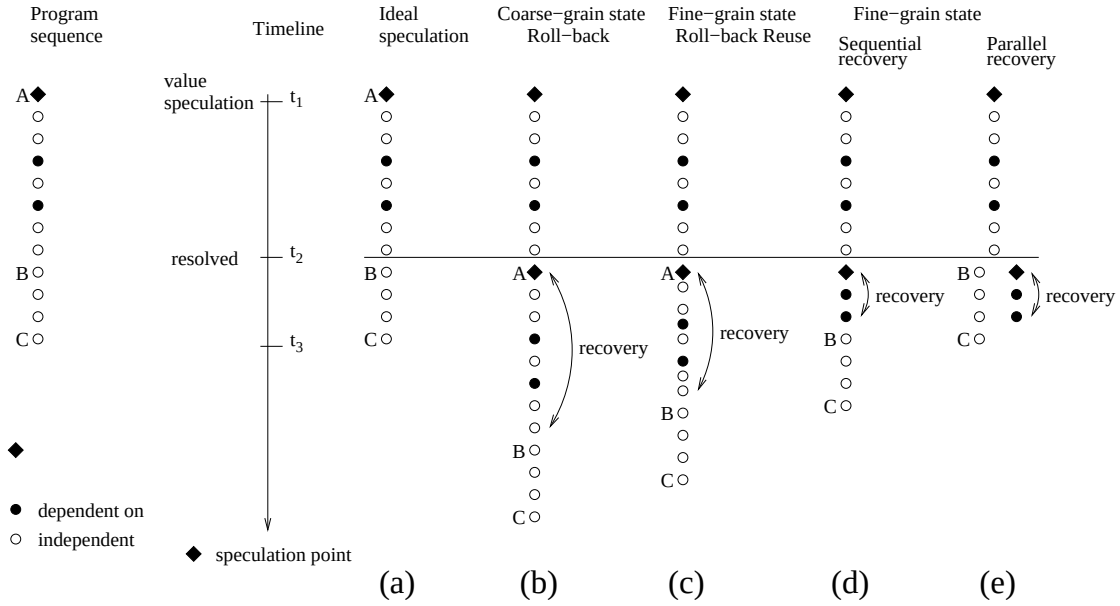


Figure 7.1. Value Speculation

This ability enables a typical *explicit parallel recovery* version shown in timeline (e) to completely overlap the recovery process with the execution of new instructions. In this approach, a second thread can be made responsible from repairing the state, and under favorable conditions, it can match the performance of the processor equipped with an ideal cache.

Notice that a similar effect is achieved by recent *Continual Flow Pipelines*(CFP) proposal [51]. It drains load-dependent instructions into a slice buffer, and then executes the slice to restore the correct state. However, CFP does not exploit the parallelism that is available during the speculation recovery. Therefore its operation corresponds to timeline (d) rather that timeline (e) in which the execution of new instructions is blocked until the recovery is done and the slice is executed. In effect, the recovery is performed sequentially.

A full-fledged implementation of FSG-RA is difficult due to the memory dependencies. Although the distinction of independent versus dependent instructions is clear in terms of register values and a simple *tag* propagation scheme suffices to identify the miss-independent instructions, such is not the case with the memory references. When a store instruction's address is miss-dependent, it is virtually impossible to know whether a subsequent load instruction is miss-dependent or not. Similarly, assuming independence for a following store instruction which references the same location results in a violation of output dependencies and the violation can't be detected until the cache miss is complete.

In order to handle the memory dependencies, we permit load speculation guided by a dependence predictor and re-execute memory operations for verification and correction. We extend the store-set [11] dependence prediction algorithm for a multi-threaded environment. This contrasts with *Continual Flow Pipelines* (CFP) [51] approach which needs to buffer all the memory instructions to ensure correct memory ordering.

In the next section we present an overview of an SMT-like FSP and show how the runahead concept can be implemented in such a FSP model.

7.2 SMT FSG-RA

In order to implement FSG-RA, we utilize a *resource replicating SMT* [55] design where most front-end resources such as the register file, the reorder buffer and the front-end pipelines are replicated (Figure 7.2). The two halves are organized such that the instructions retiring from one half can be sent to the reorder buffer of the other half. An optional FIFO called *Instruction Stream Queue* (ISQ) is placed between the two reorder buffers,

which enable further buffering of additional instructions when the destination reorder buffer becomes full. We refer to each of the halves as an *execution engine* (EE). Each EE is a fully out-of-order engine.

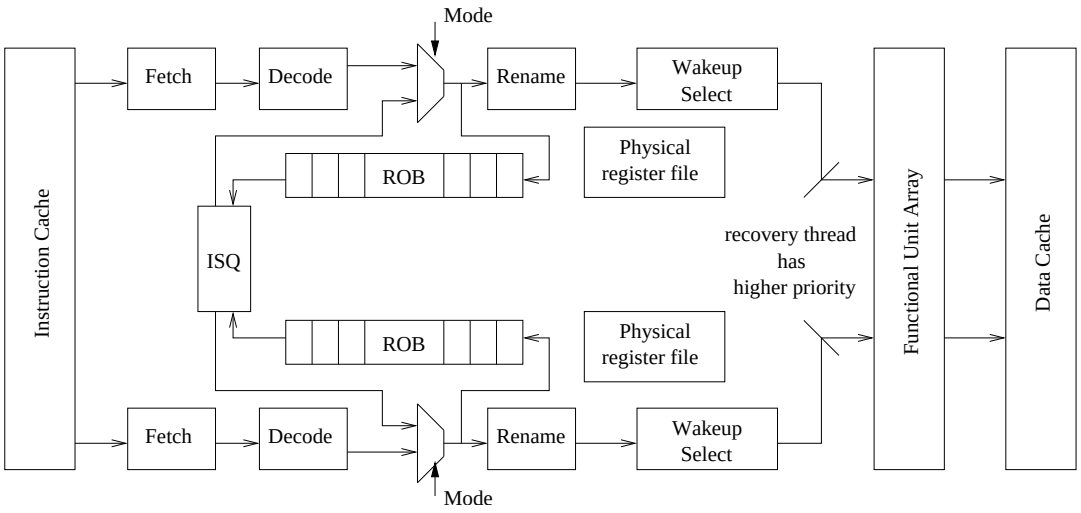


Figure 7.2. SMT FSG-RA machine model

We present two designs. In the first, the recovery thread simply re-executes all the pseudo-retired instructions from the main thread, including the miss-dependent and miss-independent instructions. We refer to this design as *FSG-RA-all*. The second design re-executes only miss-dependent instructions. We refer to this design as *FSG-RA-dep*. In case of *FSG-RA-all*, the ISQ FIFO carries the instruction stream and acts as a trace-cache. In case of *FSG-RA-dep*, only dependent instructions and memory operations are placed into the ISQ FIFO alongside their current available operand values. As discussed previously, in order to satisfy the correction property, *FSG-RA-dep* needs to buffer the current value of the operands which are miss-independent together with the instruction. Instruction flow into each EE is controlled by the current *mode* of that particular engine. There are three

modes of operation:

1. Normal Mode: The EE fetches instructions from the instruction cache and executes a particular thread, processing and retiring instructions normally. When both halves are in the *normal mode*, the processor exploits thread level parallelism. As long as there are no cache misses, the machine behaves like a conventional SMT processor.

2. Runahead Mode: The EE supplies *invalid* values to the result registers of missing loads and pseudo-retires instructions from its reorder buffer into the ISQ. When the ISQ is full, the retirement of that particular engine is stalled.

3. Recovery Mode: The EE retires and commits instructions in program order just as a normal processor, but retrieves its instructions from the ISQ instead of the instruction cache.

Improving the uni-thread performance requires both halves as follows. For simplicity, let us assume that the program is executing as a single thread on one of the halves referred to as the *main thread*. As long as the reorder buffer of the EE executing the *main thread* is not blocked by a long latency operation, the thread executes normally and retires and commits instructions. Once a load that missed in the L2 data cache reaches the head of the reorder buffer (ROB) of the *main thread*, the particular engine switches to the *runahead mode*. This load is called the *runahead trigger*:

Switching to the runahead mode: When the EE which is processing the *main thread* switches to the *runahead mode* the second EE is placed into the *recovery mode*. The execution of the *main thread* is carried out in a way similar to the original runahead proposal

[37]. The processor supplies an invalid value to the missing load and starts pseudo-retiring instructions which are fed to the *ISQ* and from there to the tail of the recovery thread's reorder buffer. In case of *FSG-RA-all*, the recovery thread executes instructions which are independent of the missing load value in parallel with the main thread until its reorder buffer becomes full. Once the recovery thread's ROB becomes full, the *ISQ* still continues to buffer instructions pseudo-retired from the main thread. Note that the state of the recovery thread is always behind the state of the main thread. During execution both threads can use any form of load speculation guided by a dependence predictor.

Cache Miss is Complete: After the L2-miss data is back, as opposed to the original runahead proposal, the main thread continues to run and does not roll back. Meanwhile, the recovery thread can move forward because the L2-miss is serviced. It can begin to verify, repair, and catch up with the main thread's state. Assuming that the validation of the main thread's state succeeds, the recovery thread eventually catches up with the main thread because it is given priority in the use of the execution resources. If the validation fails, the main thread is possibly executing on the wrong path, fetching data irrelevant to the current execution, or replacing useful data from the cache. In such cases, the main thread is stopped.

Validation Complete: Once the recovery thread finishes the validation (i.e., either detects an error, or catches the main thread) it is at the correct point in the program with a correct state. Once this state is reached, the recovery thread switches to the normal mode and continues its execution as the main thread. The other EE is now available to be used to

either improve uni-thread performance, or improve throughput via multi-threading.

This scheme improves performance regardless of whether the recovery thread executes all instructions (*FSG-RA-all*), or only the dependent instructions (*FSG-RA-dep*). The purpose of "running ahead" is to avoid the structural blockage caused by the cache miss and touch as many future cache misses as possible. Because the blocking load is discarded from the pipeline, during the L2 miss the main thread can run far ahead in the program path with a partially correct state to generate useful data cache prefetches. In case of *FSG-RA-all*, forking the recovery thread simultaneously with the switching to the runahead mode allows it to start with the same state as the main thread and follow it from behind. Instead of waiting for the L2-miss to be serviced, this early start allows the recovery thread to fill in the pipelines and execute all miss-independent instructions until its reorder buffer becomes full. Once the cache miss completes, the recovery thread rapidly repairs the state on an individual location basis. The timeline for this mode of operation is similar to the DCE proposal [60] during a *runahead*, but unlike DCE, instructions are not executed twice all the time. In the next section, we outline the fine-grain state maintenance that we implemented which is applicable to both models of FSG-RA.

7.3 State Maintenance

The *Identification Property* of FSP requires that the machine has the capability to classify individual locations with respect to the processor state. The fine-grain processor state is maintained for both threads by incorporating a set of *INV* bits with each physical register. Similarly, the rename map tables also incorporate a set of *INV* bits and a set of counters

(*CNT*) as shown in Figure 7.3. The *INV* bits serve the purpose of distinguishing the miss-dependent and miss-independent values and each store-queue entry also accommodates these bits. Note that *INV* bits are first set by the missing loads and then propagated by instructions which source those registers.

An instruction is ready to be issued if its operands are ready or the corresponding *INV* bits are set. A store instruction becomes a no operation (NOP) if its address is miss-data dependent. If its address is valid but the data to be written is invalid, the corresponding entry in the store queue is also marked as *INV*. Branch instructions which reference an invalid operand are not resolved and do not raise mis-predictions. Since the main thread is running speculatively, the store instructions with the valid address do not write values into the data cache. To prevent subsequent loads from getting stale values from the data cache, FSG-RA also incorporates a 4KB *runahead cache* [37]. Address-valid store instructions write their values with *INV*-bits into the runahead cache. Load instructions access the store buffer, the runahead cache, and the data cache simultaneously.

Attached counters on rename map table entries are used to track the fine-grain processor state between the two halves. During the runahead mode, the main thread increments the corresponding counter as each register definition is encountered. Similarly, the recovery thread increments its own counters as each definition is encountered. When the two counters are equal, all the definitions of a particular register name have been seen, and the processor state with respect to that register is consistent. At this point, the recovery thread either needs to verify the value, or repair it, depending on the setting of the *INV* bit

on the main thread's RMAP entry (*Correction Property*). Shown in Figure 7.3, the logical register *R*7 needs to be repaired and *R*9 needs to be verified.

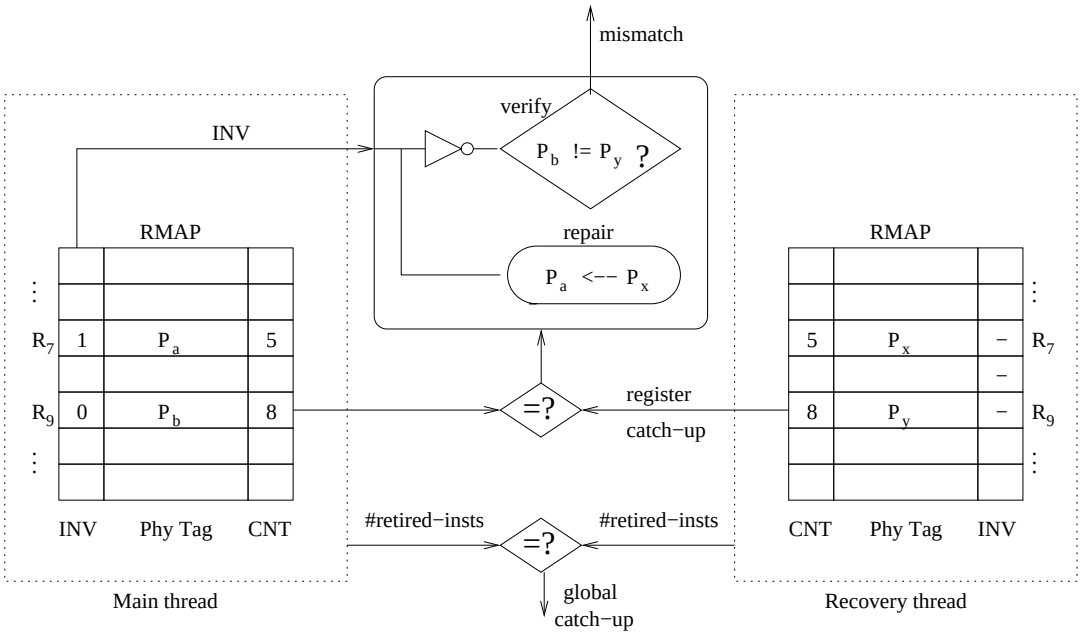


Figure 7.3. Fine grain state recovery

Instructions retire in-order as usual to maintain the execution. The in-order state of the main thread, though speculative, is presented in the in-order map table, RMAP [20]. All *INV* and *CNT* fields are reset at the beginning of runahead mode. When a producer instruction retires, it updates the RMAP as usual. The *INV* bit of its physical destination register is copied to the RMAP, and the *CNT* is incremented by one. When the *INV* bit of the RMAP in the main thread is set, it indicates the value is miss-dependent and it needs to be repaired. The correct value is copied from the *recovery thread*'s renamed register to the *main thread*'s renamed register: $P_a \leftarrow P_x$. Such a repair permits the main thread to use the correct value of the register in its future references during the recovery.

When the *INV* bit of an RMAP entry in the main thread is not set, it indicates the value is miss-independent. However, as previously discussed, we still need to verify its value because the main thread might have computed the wrong value because of load/store communication through the memory. For example, if a store instruction is dependent on the value of the *runahead trigger* to compute its address, and a subsequent miss-independent load references the same location, it will acquire the stale value from the data cache, and the corresponding *INV* bit won't be set. In this case, the *recovery thread* detects the mismatch and signals a validation failure if $P_b \neq P_y$, resulting in the main thread to be stopped.

Although one could detect that the state recovery is complete with respect to the register values when all the counters on both halves become equal, this approach won't detect the proper memory state. Therefore, in the case of *FSG-RA-all*, in addition to the above mechanisms, we incorporate two global counters on each side of the SMT. These count the number of retired instructions from the main thread and the recovery thread under the runahead mode and the recovery mode, respectively. These counters are initialized to zero when one EE enters the runahead mode and the other is put into the recovery mode. When the two global retired-instruction counters become equal, the recovery thread has caught the main thread and the recovery process is complete. Detection and correct handling of the memory state in case of *FSG-RA-dep* is more involved and is elaborated in Section 7.7.

7.4 Termination of Runahead Mode

The use of two threads to maintain the runahead execution gives excellent control over the runahead mode. Specifically, runahead mode can be held until it is detected that it is

not providing benefits and can also be exited upon appropriate state recovery, or when it is detected that the runahead mode is not providing benefits. There are three conditions under which the runahead execution is terminated: (1) The normal completion of the recovery after an L2 miss; (2) Detection of useless runahead; and (3) Detection of a control or data mis-speculation.

Upon normal completion, the main thread is ahead of the recovery thread, possibly has touched a few more useful misses, and the recovery thread has a known state with respect to the initial cache miss. Even though this mode could continue, there is little benefit in dual execution and the main thread is stopped. The recovery thread becomes the main thread and resumes normal mode. The EE running the old main thread is available to improve multi-thread performance, or to be used as the next recovery engine.

```
int
list_length (t)
{
  register tree tail;
  register int len = 0;

  for (tail = t; tail; tail = TREE_CHAIN(tail))
    len++;

  return len;
}
```

Listing 7.1. list_length in 176.gcc

When the trigger load is the head of a chain of pointers, runahead execution cannot generate useful prefetches. An interesting example is a C function of *176.gcc* in SPEC CINT2000 suite. Shown in Listing 7.1, the list_length function returns the length of a node

chain via TREE_CHAIN. This pointer-chasing of the loop always brings an L2-miss if the root node loading is an L2-miss. As a result, a conventional runahead processor enters the runahead mode for each loop iteration. However, no useful work can be found with running ahead.

In order to optimize the runahead behavior we have implemented a mechanism to detect such useless runahead. For this purpose, we associate a single *issue-bit* with each load instruction in the main thread under the runahead mode. If a load instruction is valid and issued into the data cache then its issue-bit is set. Otherwise, the issue-bit is reset. This issue-bit is fed to the recovery thread together with the instruction into the ISQ. Under the recovery mode, the recovery thread can tell a cache miss is a pointer-chasing miss if its issue-bit is zero. It indicates that it is dependent on the original runahead trigger miss and it was not issued in the main thread. In order to stop the runahead mode properly, we stop pseudo retirement in the main thread and let the recovery thread catch on the main thread with respect to the damaged state. When pseudo-retirement is stopped in the main thread, the thread is said to be in the *blocking mode*. Once the recovery is complete, we allow the main thread to continue normally, instead of allowing the recovery thread to continue as the main thread. This is because the main thread will be further ahead in the execution sequence than the recovery thread once its damaged state is repaired. Note that this technique effectively utilizes useful instructions executed during the recovery process without special micro-architecture mechanisms. Thus the latency of the recovery process is effectively hidden by the useful executions in the main thread, improving the ILP.

The outlined mechanisms provide easy detection and handling of control and data mis-speculations. Naturally, a branch mis-prediction by the runahead engine goes undetected when the branch instruction is load-miss dependent. In addition, when there is a dependence through memory where *INV* bits cannot be propagated, the main thread may be computing an incorrect value. Both of these cases are easily detected by the recovery thread. In case of a branch mis-prediction, the branch instruction is at the head of the ROB of the recovery thread; in case of a load value mis-speculation, the load instruction is at the head of the ROB of the recovery thread. Upon detection of the mis-speculation, the main thread is stopped and the EE is released. The recovery thread enters the normal mode of operation with the correct target (or from the load instruction with the correct value) and the execution is resumed.

7.5 Thread Memory Dependencies

For intra-thread memory dependencies, FSG-RA utilizes the store set [11] algorithm to predict the memory dependencies in the usual way: When a load instruction is decoded, it accesses the Store Set Identifier Table (SSIT) based on its PC and gets its store set identifier (SSID). If it has a valid SSID, it accesses the Last Fetched Store Table (LFST) and gets the ROB index of the most recently fetched store instruction on which it depends. If a dependence is predicted between a load and a previous store instruction belonging to the same thread, it is blocked until the dependent store instruction issues.

Handling of inter-thread dependencies requires an extension to the algorithm. Under the blocking mode, a load instruction executed by the main thread may be dependent on a

store instruction that will be issued by the recovery thread and it has to wait until that store is issued. Unfortunately, it is very difficult for a load instruction in the main thread to be aware of its dependence in the recovery thread. Even if the main thread's load instruction is allowed to access the recovery thread's LFST, it will not always get the correct information since it is quite possible the store instruction has not yet been fetched into the recovery thread's pipeline when the load instruction is decoded in the main thread.

Our solution extends the algorithm by sharing the SSIT table between the two threads and incorporating private LFSTs for each engine. We also include a new table called Store Set Counter Table (SSCT). The SSCT counts the number of pseudo-retired stores for each live store set in the main thread under the runahead mode. When a store in the main thread is pseudo-retired, the corresponding counter entry in the SSCT is incremented by one if it has a valid SSID. Note that all pseudo-retired instructions from the main thread are fetched and re-executed by the recovery thread. Thus, under the blocking mode, loads in the main thread can be aware of the memory dependence information in the recovery thread by accessing the SSCT, even before those stores belonging to the store sets appear in the pipeline. This algorithm is depicted in Figure 7.4.

Under the blocking mode, when a load is decoded in the main thread, it accesses the LFST and the SSCT in parallel, if it belongs to a store set. The LFST and the SSCT together provide the accurate dependence prediction for the load. There are three cases for LFST and SSCT values:

$LFST = 0, SSCT = 0$: There is no intra-thread or inter-thread dependence. The load is

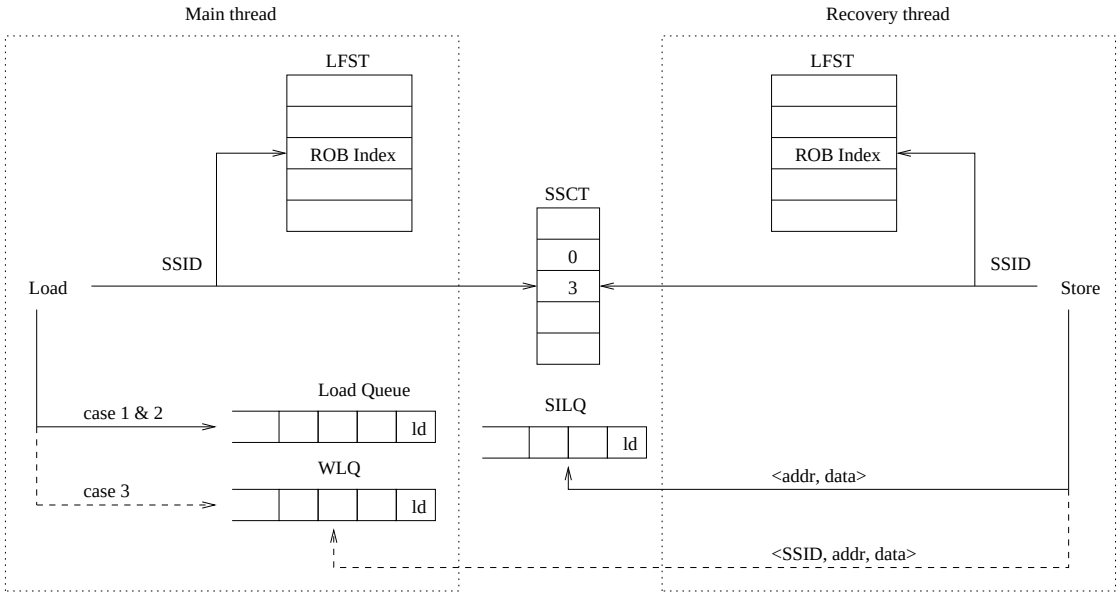


Figure 7.4. Memory Ordering in FSG-RA

inserted into the main thread's load queue and it is issued when it becomes ready.

$LFST > 0, SSCT = 0$ *or* $SSCT > 0$: There exists an intra-thread dependence. LFST holds the ROB index of the most recently fetched store instruction in the main thread on which the load depends. Since it is dependent on a store in the main thread, the load is inserted into the main thread's load queue and it is issued after that store is issued regardless of whether an inter-thread dependence exits in the recovery thread. The intra-thread dependence overrides the inter-thread dependence.

$LFST = 0, SSCT > 0$: The store set predicts that this load is not dependent on any store in the main thread, but it depends on some stores in the recovery thread. The load is inserted into a Waiting Load Queue (WLQ) where it waits until the latest store in the same store set commits the value to the data cache in the recovery thread. Each entry in the WLQ contains the load's address and SSID. It is implemented as a CAM structure which can be

associatively searched using the SSID as the key.

After the L2 cache miss which triggered the runahead execution is completed, the recovery thread can move forward. In the recovery thread, once a store instruction is retired, it commits its result into the data cache. Meanwhile, it decrements the counter in the SSCT if it has a valid SSID. If the counter becomes zero, it indicates it is the last store instruction in this store set. Then it sends the $< SSID, address, data >$ into the WLQ to forward the data to those loads belonging to the same store set. The SSID is used to associatively search the WLQ, if there is a match and both addresses are the same, the store's data is forwarded to that load. If there is a match but their addresses are different, then it indicates that load is not dependent on the store. It is then removed from the WLQ and inserted back into the load queue of the main thread. This technique effectively provides load forwarding between the two threads as well as reducing the load-queue pressure during the blocking mode.

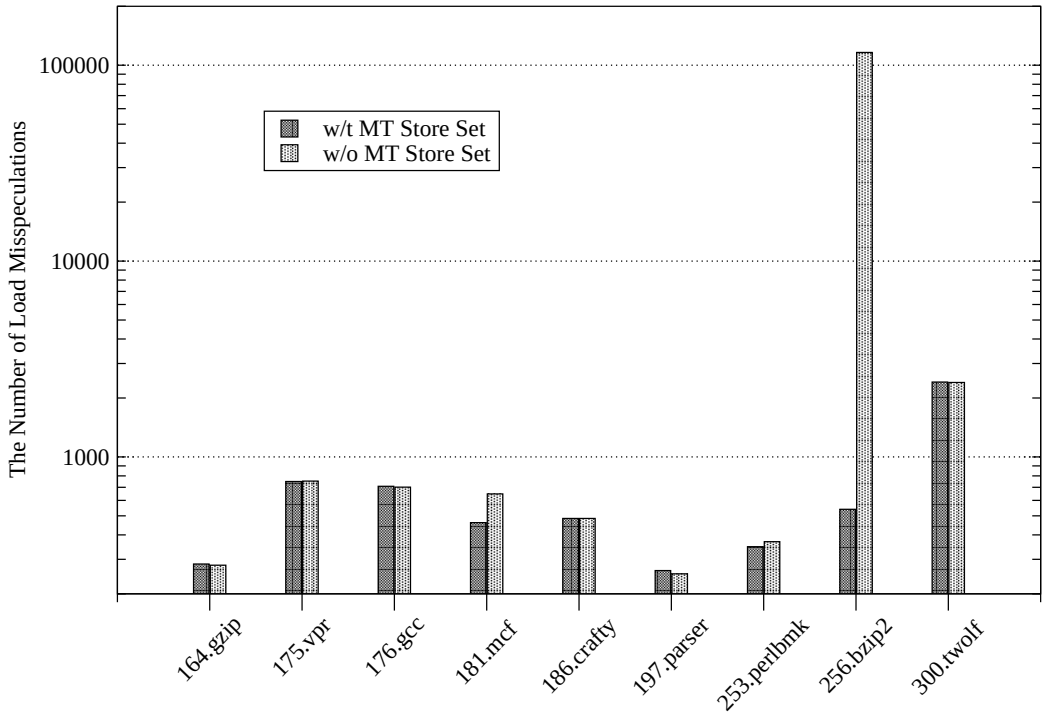
7.6 Detecting Memory Order Violations

The intra-thread memory violations are handled locally in the main thread and the recovery thread as usual. When a store instruction is issued, the local load queue CAM is associatively searched with its address. If there is a matching load that is incorrectly speculatively issued prior to the store instruction, it is marked as a mis-speculation.

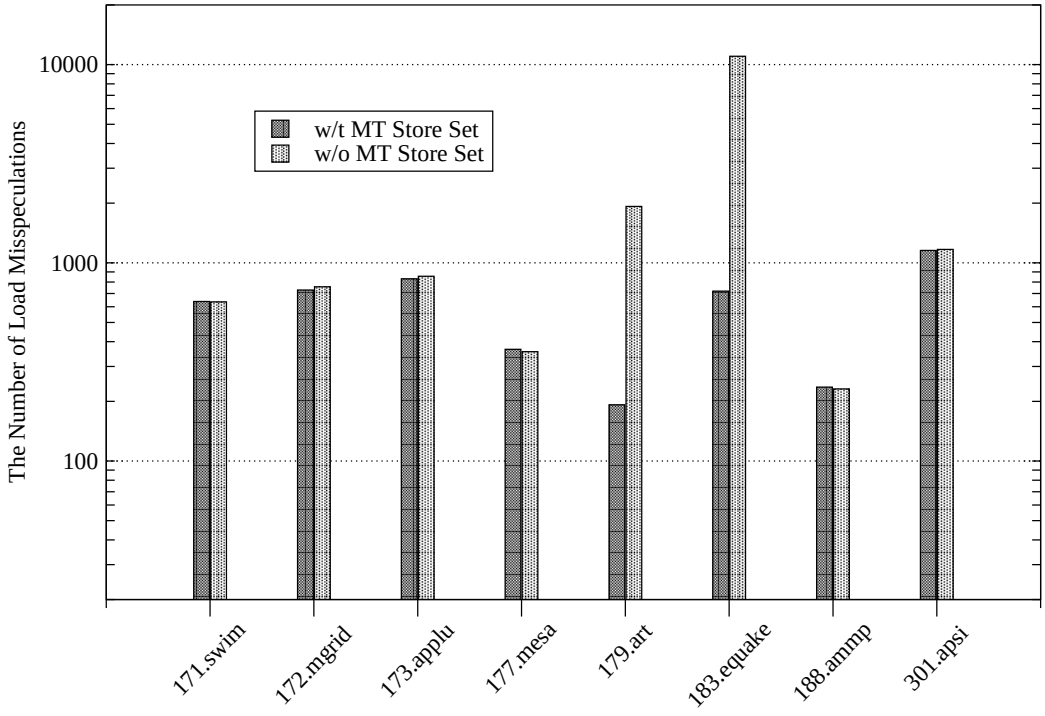
In order to detect the inter-thread violations between two threads, FSG-RA maintains a load queue called the Speculative Inter-thread Load Queue (SILQ). Under the blocking mode, when a load instruction is issued in the main thread, it is put into the SILQ if it

gets the value from the data cache or the forwarding data from the committing stores of the recovery thread. Note that the load is not put into the SILQ if it gets the forwarding data from the store queue of the main thread. This is the second type of case where the intra-thread dependence overrides the inter-thread dependence. When a store instruction is retired in the recovery thread, it associatively searches the SILQ with its address. If there is a match and their values are not equal, then the mis-speculation flag of the matching load is set. If there is a match and their values are equal, then the mis-speculation flag is reset if it has been set. This value-based violation detection algorithm is adopted from [39]. Note that this detection process can only set/reset the inter-thread violation flag. If a load instruction in the main thread is issued out-of-order before a previous store which is also in the main thread and writes to the same address, it is marked as an intra-thread mis-speculation locally and is not reset by the SILQ detection process.

If a memory ordering violation is detected, the thread's state needs to be restored. FSG-RA utilizes the conventional state recovery method to handle the memory mis-speculation exceptions. Since load mis-speculations do not change the dynamic program path, they can be handled locally. When a load instruction reaches the head of the reorder buffer, the recovery process is invoked if its mis-speculation flag is set. The thread's state is restored by copying the retirement map table to the front-end map table, and the thread restarts from the mis-speculated load instruction. This multi-thread memory ordering algorithm significantly reduces the number of memory order violations for a set of benchmarks (Figures 7.5(a) and 7.5(b)).



(a) CINT2000



(b) CFP2000

Figure 7.5. Mis-speculations Enhanced Store Set Algorithm

7.7 Re-Executing Only Dependent Instructions

We refer to the presented design where the recovery thread simply re-executes all pseudo-retired instructions from the main thread as *FSG-RA-all*. An alternative is to re-execute only miss-dependent instructions which we refer to as *FSG-RA-dep*. This policy effectively increases the efficiency of FSG-RA at the cost of increased hardware complexity. In FSG-RA-dep, the main thread drains only the miss-dependent instructions into the ISQ during the runahead mode. Obviously, each such instruction has at least one operand that is dependent on the runahead trigger. As previously discussed, the value of the independent operand must also be buffered in the corresponding entry and all memory operations need to be re-executed by the recovery thread. Revisiting these reasons, it is important to remember that store instructions only speculatively commit their values into the runahead cache, and they should commit into the cache/memory by the recovery thread in the precise program order. Moreover, it is possible that a miss-independent load instruction in the main thread may get a stale value from the memory during the runahead mode. Therefore the recovery thread needs to verify all miss-independent load instructions to detect such errors. Despite these complications, as it will be shown in the experimental section, the *FSG-RA-dep* design is quite favorable since it greatly boosts the performance with a manageable increase in complexity.

Recall that the recovery thread in FSG-RA-all is able to maintain a correct state at any point because it re-executes all instructions. However, in FSG-RA-dep, the recovery thread may not have a correct state when it finishes the validation because it will have correct

miss-dependent state, but not necessarily a correct miss-independent state.

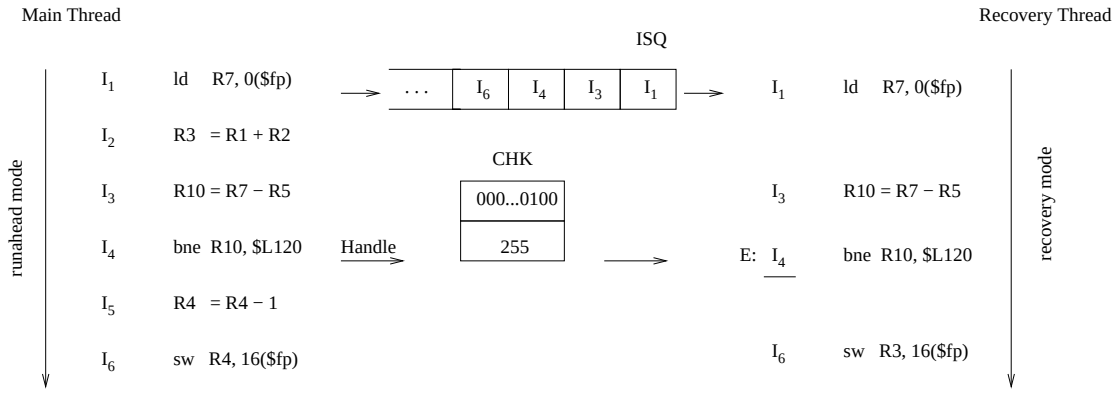


Figure 7.6. Example of FSG-RA-dep

Consider the example shown in Figure 7.6. Suppose the load instruction I_1 triggers the runahead execution. I_3 and I_4 are dependent on the trigger and instructions I_2, I_5 and I_6 are not. When they are pseudo-retired one by one from the main thread, I_1, I_3, I_4 and I_6 (it is a memory operation) are drained into the ISQ. Branch I_4 cannot be resolved because it is miss-dependent. When the recovery thread fetches these instructions from the ISQ and re-executes them after the L2-miss of I_1 in the main thread is serviced, I_1 in the recovery thread can get the data from the cache/memory, and the recovery thread can move forward to repair the state. Branch instruction I_4 might have been mis-predicted by the main thread and the main thread may be running along the wrong path. Unfortunately, at this point neither the recovery thread nor the main thread has the correct state at I_4.

To address this issue, FSG-RA-dep combines a *checkpoint* [21] scheme to repair the register state and a history buffer to repair the memory state. Both of these techniques are employed as differential techniques and using these techniques FSG-RA-dep is capable of

salvaging the independent work that has already been done in the main thread.

7.7.1 Handling the Register State

To handle the register state, the main thread creates the checkpoints of the miss-independent register state at instructions referred to as *handle* instructions. Suppose that the main thread creates a checkpoint at I_4. It contains the destination result of I_2 for the miss-independent register $R3$. When the recovery thread retires the same instruction I_4, it reads the checkpoint and writes the value into the in-order renaming register of $R3$. Thus, the recovery thread maintains a correct state at I_4. If an exception occurs, it can restart the execution from this point with a correct state.

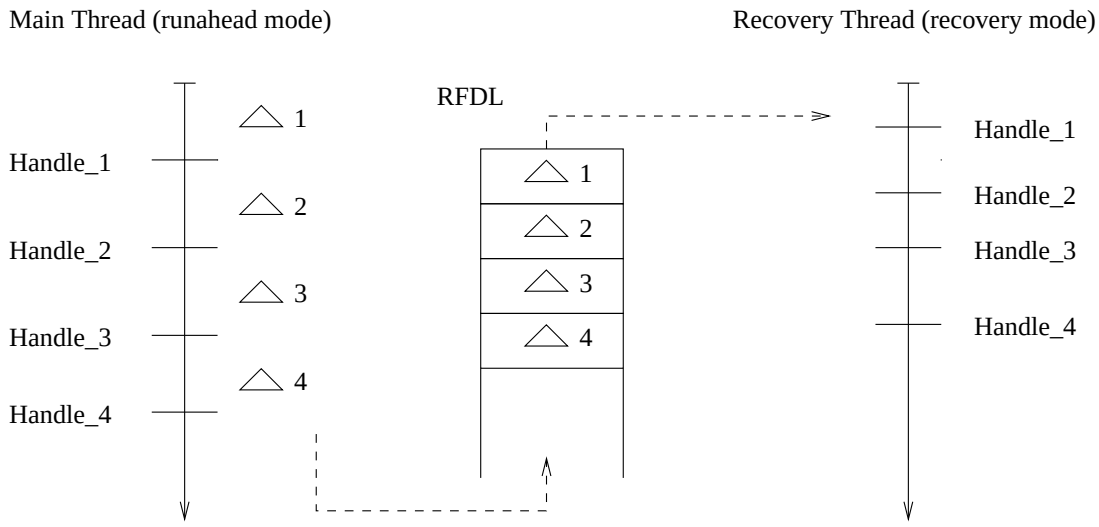


Figure 7.7. Multiple Handles

FSG-RA-dep implements the checkpoints using the difference technique introduced in [21]. Each checkpoint buffers the difference of the miss-independent register state from one handle to another. A register mask is used to record which miss-independent registers

are defined between two handles. In the above example, when I_2 is pseudo-retired, the corresponding bit of the mask is set. Later, if the main thread creates the first handle at I_4 when it retires, then the current register mask is inserted into the checkpoint. It indicates that $R3$ is miss-independent, has the difference value from I_1 to I_4, and provides the in-order state value of $R3$. The main thread creates multiple handles on its pseudo-retired instruction stream such that each forward difference δ between the two successive handles is inserted into a list in the program order. We refer to this FIFO list as the Register Forward Difference List (RFDL), shown in Figure 7.7,

Since the overhead of creating a handle at each instruction would be prohibitively high, FSG-RA-dep creates handles only at branch instructions. With this scheme, when an exception happens and the current point is not a handle, the execution is restarted at the previous handle. Unfortunately, there may be committed memory operations between the current point and the most recent handle. As a result, rolling back will not only discard some useful work, but the memory state will not be correct. In order to maintain the correct memory state, we employ the history buffer technique for memory operations.

7.7.2 Handling the Memory State

Similar to the miss-independent register state, the memory state can be repaired by using the difference technique. Before a store instruction is retired and committed into the cache/memory, the data in that location is read and inserted into a *History List*. When a handle instruction is retired, the history list is flushed. Thus, the history list always keeps the backward difference of the memory state from the latest retired handle point

to the current retirement point. If the recovery thread needs to roll back to the most recently retired handle point, the history list is used to restore the memory state. The history data saved in the list are stored back to undo all modifications to the memory introduced by the wrong speculative state. Thereby the in-order state of the memory at the latest retired handle is repaired. Note that the alternative to using a history-difference is to buffer the store instructions in the recovery thread until a handle is received without an exception. This alternative is more suitable when FSG-RA is employed in a multi-processor environment. It would require snooping by the main-thread into the store buffer kept for this purpose in the recovery thread.

It is also possible to extend the History List based algorithm to a multi-processor environment. In a multi-processor setting, the fact that a commit is speculative must be communicated to other processors, a problem which is similar to Thread Level Speculation (TLS) [17, 45]. We leave this to future work.

7.8 Experimental Evaluation

We evaluated and compared four machine models: the Baseline model, the Runahead CSP model, and the two FSG-RA FSP models. We kept the above four models identical in all aspects except the L2-miss latency tolerance scheme. Two EEs in FSG-RA are based on the baseline mode's configuration. Each has its own physical register file and internal queues. They share the functional units and the cache/memory system. We model a one-cycle delay when copying a register value between the two threads. The parameters of machine models are shown in Table 7.1.

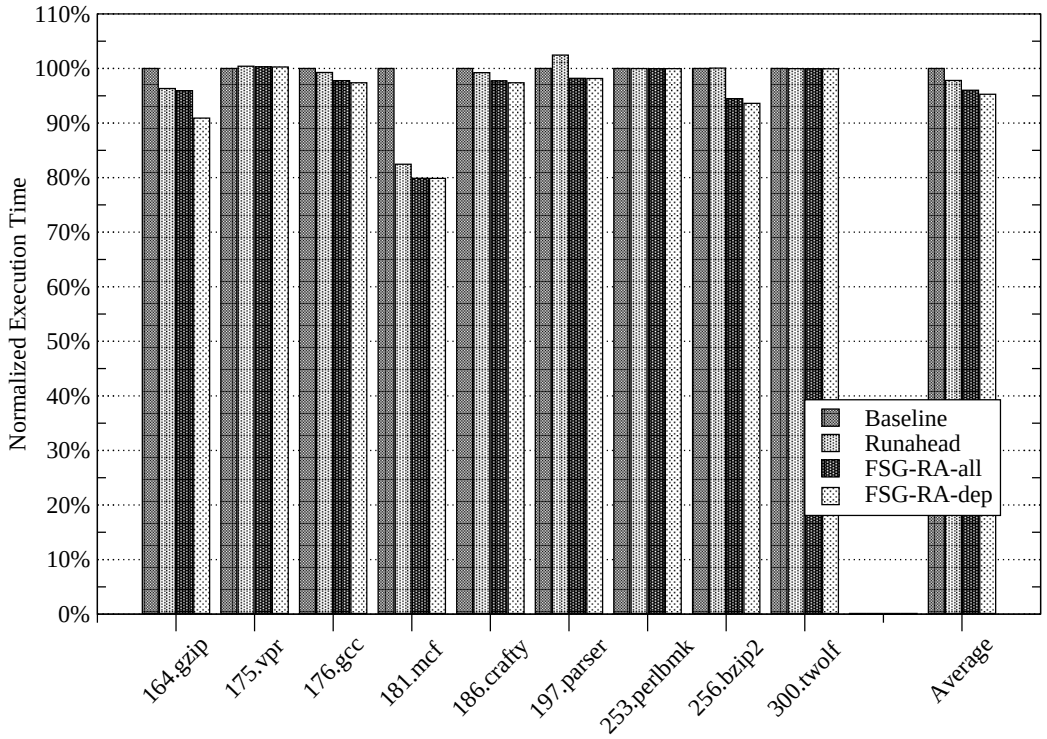
Parameter	Configuration
Issue/Fetch/Retire width	8/8/8
Scheduling window size	64
Reorder buffer size	128
Load/Store queue size	128
WLQ size	128
SILQ size	128
Register file entries	256
Functional units	Issue width Symmetric
Branch predictor	16K-bit gshare, 32K-entry BTB
ISQ size	1K-entry
Memory disambiguation	Store set
Runahead cache	4KB, 4-way, 8B/line, LRU
Split Data cache	L1: 32KB, 2-way, 64B/line, 2-cycles, LRU, 4-port, 128 MSHRs L2: 512KB, 4-way, 64B/line, 10-cycles, LRU, 1-port, 128 MSHRs
Memory	220 CPU cycles

Table 7.1. Machine's configurations

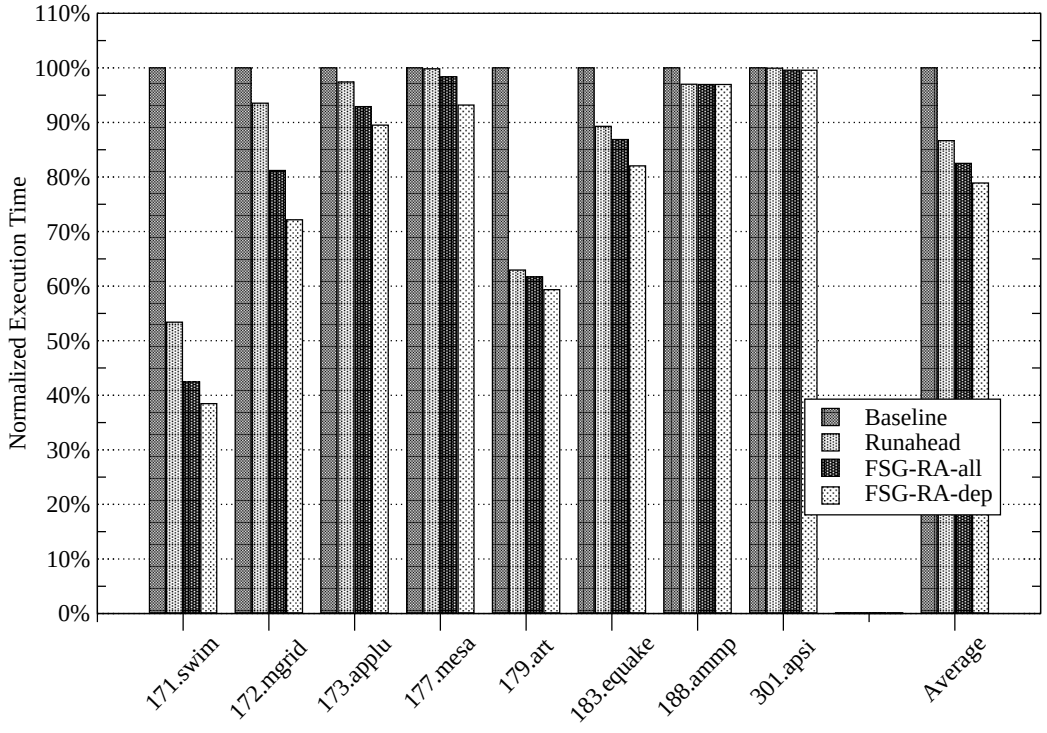
7.8.1 Performance Results

The normalized execution time for each program in the benchmark suite for each model is shown in Figures 7.8(a) and 7.8(b). The average bar shows the average of the normalized execution time, which is calculated as the arithmetic mean of each benchmark's normalized execution time. All evaluated machines achieve significant performance improvement over the baseline model and both FSG-RA models outperform the Runahead model across all CINT2000 and CFP2000 benchmarks.

The speedup for each benchmark is calculated from its normalized execution time shown in Figures 7.8(a) and 7.8(b). The average bar shows the mean of all individual speedups. Figures 7.9(a) and 7.9(b) illustrate the percent speedup over the baseline model. As it can

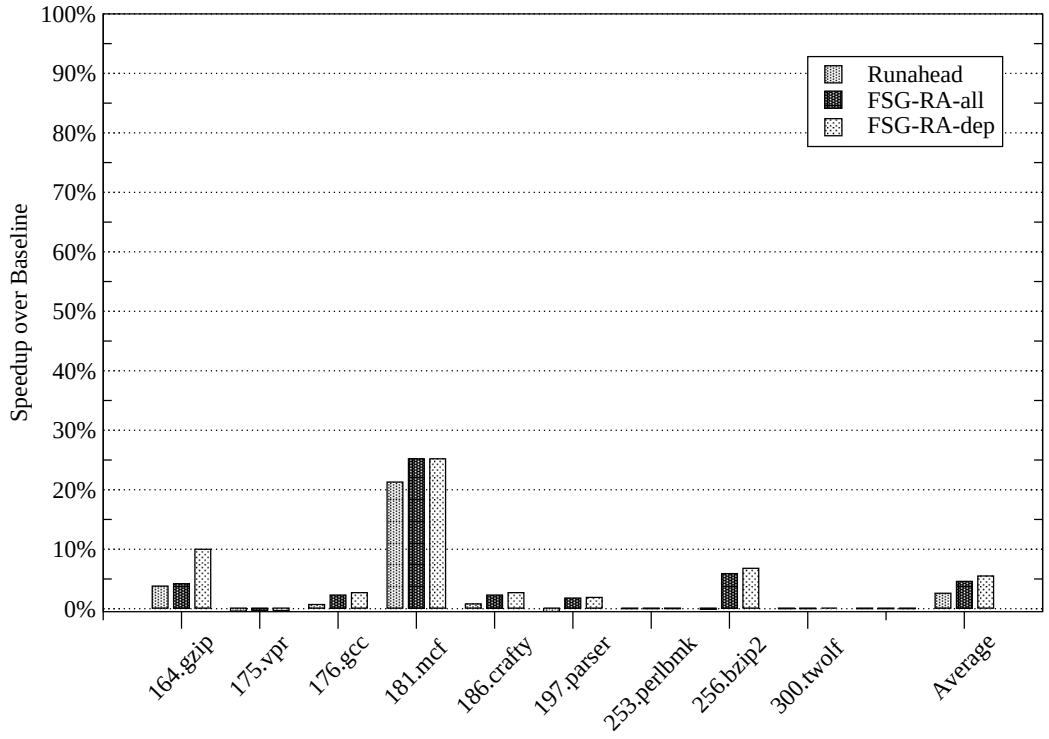


(a) CINT2000

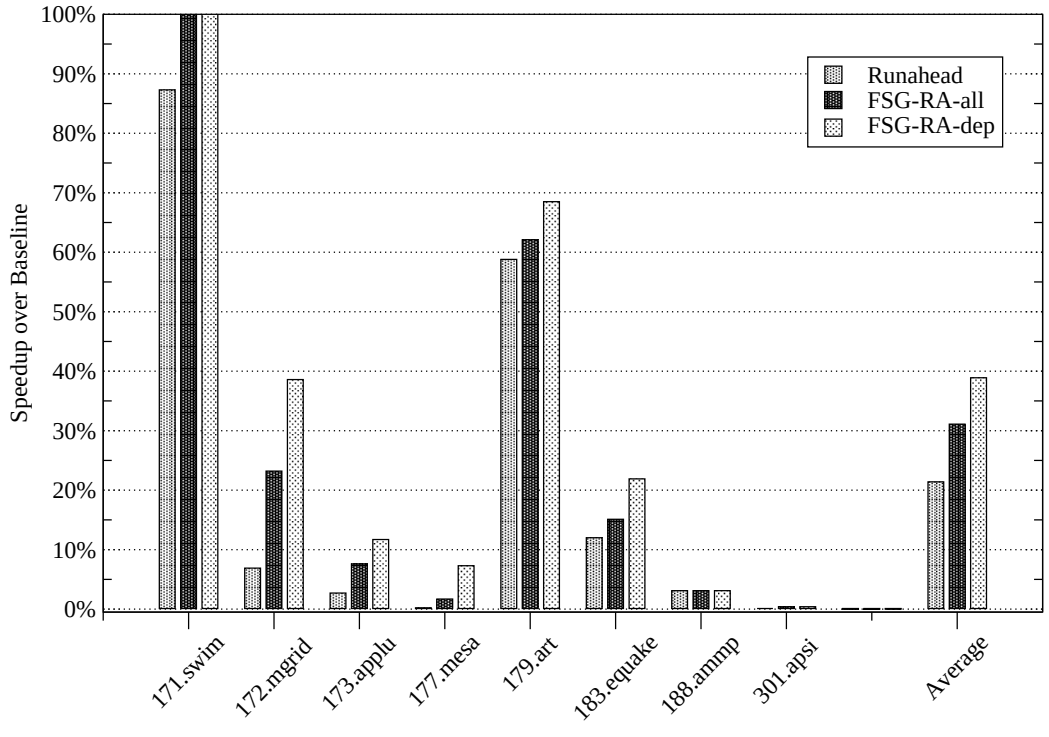


(b) CFP2000

Figure 7.8. Performance of 4 Models



(a) CINT2000



(b) CFP2000

Figure 7.9. Δ Performance

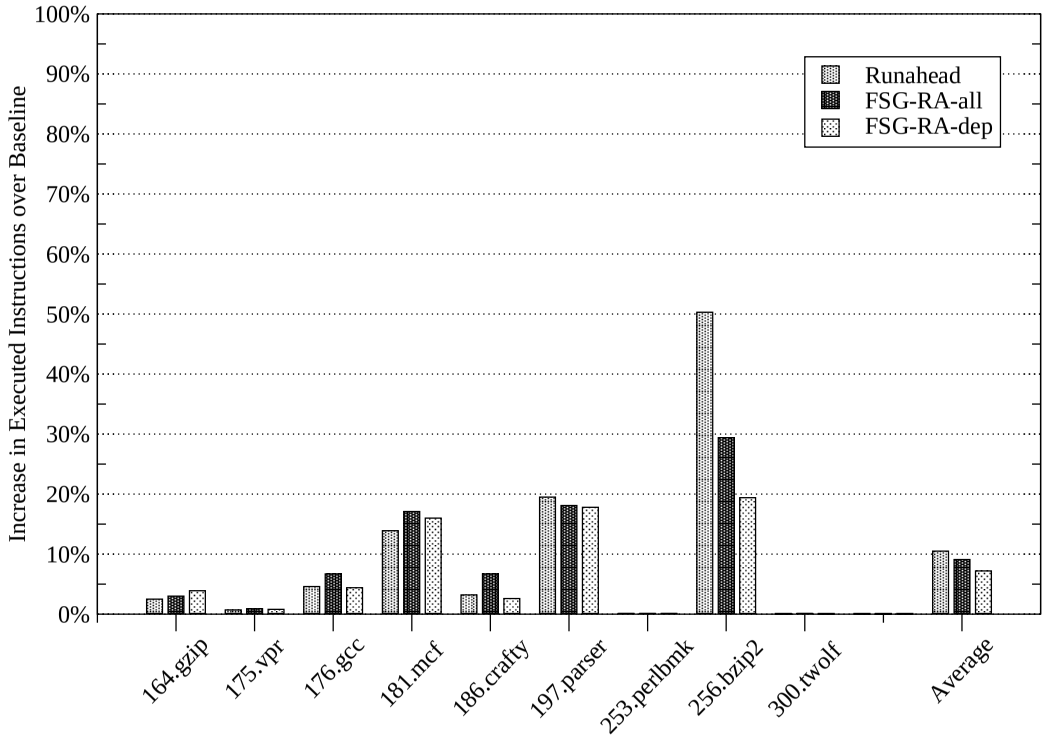
be seen from Figure 7.9(a) the three models obtain limited performance improvement for the integer benchmarks. The Runahead model achieves a 2.6% average speedup while the FSG-RA-all and the FSG-RA-dep obtain 4.6% and 5.5%, respectively. An exception is on 181.mcf, where the cache miss rate is relatively high. The three models outperform the baseline model on 181.mcf by 21.3%, 25.2% and 25.2%, respectively.

For floating point benchmarks (Figure 7.9(b)), the Runahead model obtains an average speedup of 21.4% and up to 87.3% (171.swim). Meanwhile, the FSG-RA-all model achieves a speedup over the baseline model on CFP2000 by an average of 31.1% with a maximum improvement of 135.6% on 171.swim. Since FSG-RA-dep model re-executes only miss-dependent instructions to recover the state, FSG-RA-dep gains an average speedup of 38.9% on CFP2000 and up to 160.0% (171.swim). Multi-threaded store set algorithm is very effective in terms of performance gains for applu and equake, boosting performance by 6-8 % compared to a FSG-RA-dep without the multi-threaded store set algorithm.

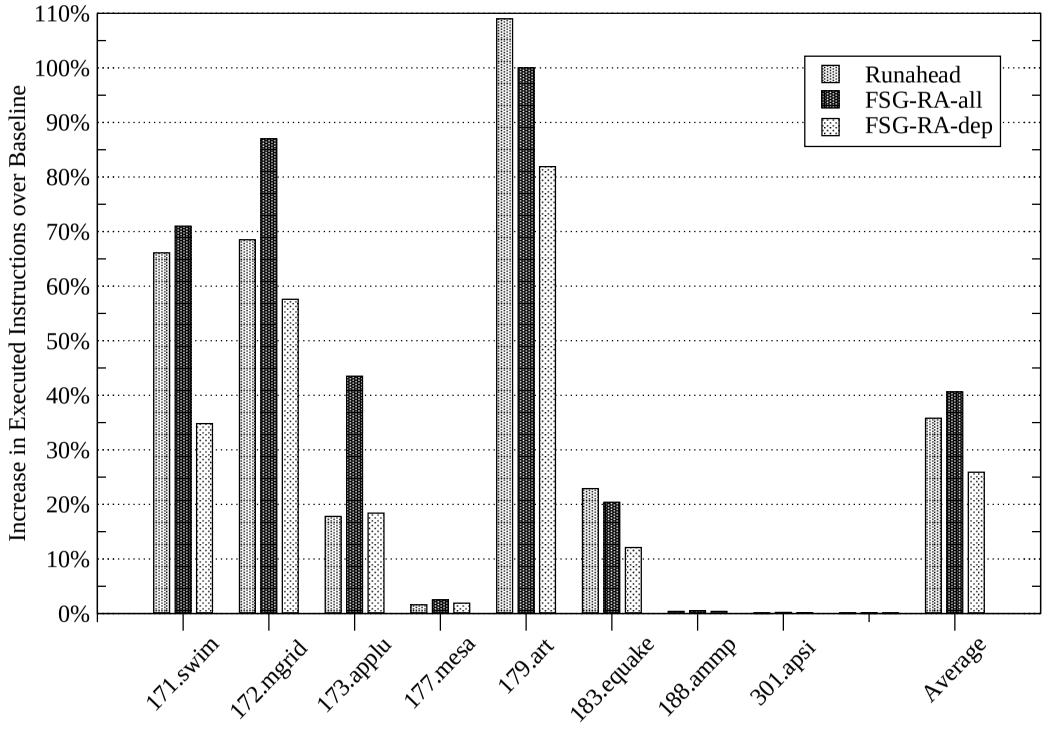
Figure 7.9(a) also shows that the three models do not achieve any improvement for certain benchmarks such as 253.perlbmk and 300.twolf because of few L2 misses. The Runahead model degrades performance for the benchmarks 175.vpr and 197.parser. The degradation is a result of short runahead periods [36].

7.8.2 Efficiency of FSG-RA

Both runahead execution and FSG-RA execute more instructions than the baseline processor. Using a recent analysis method [36], we evaluated the efficiency of FSG-RA



(a) CINT2000



(b) CFP2000

Figure 7.10. Δ Number of Instructions

algorithms. Following the definition of *Efficiency* in [36] given by:

$$Efficiency = \frac{Percent\ Increase\ In\ Performance}{Percent\ Increase\ In\ Executed\ Instructions} = \frac{\Delta\ Performance}{\Delta\ Instructions} \quad (7.1)$$

Note that, this is only an indirect measure of the dynamic power consumption of the processor. Given the additional structures that makes an SMT, there will be non-negligible increase in the static power consumption compared to a uni-processor. However, since we use an idle thread to speed-up a single-threaded program, we believe a fair comparison should be between an SMT that employs pure runahead and one that employs FSG-RA.

Figure 7.10(a) and Figure 7.10(b) show the increase in the number of executed instructions for the three models over the baseline model. As it can be seen from the figure, on an average, the Runahead model executes 35.8% more instructions than the Baseline model on CFP2000 and it achieves a 21.4% speedup. Meanwhile, FSG-RA-all model increases the number of executed instructions by an average of 40.6% in order to obtain the speedup of 31.1% on CFP2000. The FSG-RA-all model outperforms the Runahead model at the cost of executing 4.8% more instructions.

The FSG-RA-dep model, reduces the number of executed instructions over the Runahead model across all floating point benchmarks and outperforms it (Figure 7.10(b)). It executes 25.9% more instructions than the Baseline model.

Efficiencies of three models are listed in Table 7.2 and Table 7.3. On an average, the FSG-RA-dep model obtains the efficiencies of 0.84 and 3.0 in CINT2000 and CFP2000,

respectively. Meanwhile, the Runahead model achieves only 0.39 and 1.47, respectively. Shown in the table, the efficiencies of the FSG-RA-all model are also higher than that of the Runahead model. This is because the FSG-RA-all model has much better performance than the Runahead model.

	164.gzip	175.vpr	176.gcc	181.mcf	186.crafty
Runahead	1.52	-0.56	0.16	1.53	0.24
FSG-RA-all	1.39	-0.33	0.34	1.47	0.35
FSG-RA-dep	2.54	-0.35	0.61	1.57	1.03

	197.parser	253.perlbmk	256.bzip2	300.twolf	Average
Runahead	-0.12	0.30	0.00	0.47	0.39
FSG-RA-all	0.10	0.49	0.20	0.73	0.53
FSG-RA-dep	0.11	0.57	0.35	1.15	0.84

Table 7.2. SPEC CINT2000 Efficiencies

	171.swim	172.mgrid	173.applu	177.mesa
Runahead	1.32	0.10	0.15	0.11
FSG-RA-all	1.91	0.27	0.18	0.67
FSG-RA-dep	4.60	0.67	0.64	3.88

	179.art	183.equake	188.ammp	301.apsi	Average
Runahead	0.54	0.52	7.60	1.39	1.47
FSG-RA-all	0.62	0.74	6.98	2.04	1.67
FSG-RA-dep	0.84	1.80	7.51	4.08	3.00

Table 7.3. SPEC CFP2000 Efficiencies

7.8.3 Effect of Branches

By allowing the recovery thread to verify and repair the state while the main thread continues the execution, FSG-RA can fully utilize the multi-threading computing power for the uni-thread programs in the presence of a long L2-miss. However, it is possible

that the main thread is running on the wrong path because some branch instructions are dependent on the missing L2 data and cannot be resolved. If the main thread is on the wrong path, the data cache will be polluted and no useful miss-independent instructions will be executed. As discussed in Section 7.4, , if the recovery thread detects that the main thread is on the wrong path, FSG-RA will stop the main thread before the recovery thread's state catches up with the main thread's state.

Table 7.4 and Table 7.5 show the collected data of branch instructions under the runahead execution in the FSG-RA-all model. The first row shows the number of branch instructions per 1000 pseudo-retired instructions. The second row shows the percentage of these branch instructions which are dependent on the original L2 miss data. The higher this number, the more likely that the main thread will follow the wrong path. The last row of the table indicates the percentage of killed main threads due to the branch mis-predictions.

	171.swim	172.mgrid	173.applu	177.mesa
BR/1K pseudo-retired	10.7	14.0	43.4	190.1
% miss-dependent	0.1%	1.5%	0.2%	3.8%
% main thread killed due to branch-misp.	29.3%	43.2%	2.3%	100.0%

	179.art	183.equake	188.ammp	301.apsi	average
BR/1K pseudo-retired	127.1	80.7	199.1	61.4	34.0
% miss-dependent	15.1%	2.0%	15.6%	3.9%	0.3%
% main thread killed due to branch-misp.	59.2%	14.6%	97.5%	62.2%	13.1%

Table 7.4. CFP2000 Branch statistics in FSG-RA-all

As it can be seen from Table 7.4, on an average, there are 34.0 branches in every 1000 pseudo-retired instructions on CFP2000 benchmarks. Moreover, only 0.3% of them are

	164.gzip	175.vpr	176.gcc	181.mcf	186.crafty
BR/1K pseudo-retired	46.0	174.9	162.1	233.8	113.5
% miss-dependent	6.0%	21.7%	5.4%	48.0%	6.9%
% main thread killed due to branch-misp.	94.4%	97.8%	51.7%	87.5%	48.5%

	197.parser	253.perlbmk	256.bzip2	300.twolf	average
BR/1K pseudo-retired	235.9	145.6	95.7	144.0	119.6
% miss-dependent	57.2%	18.0%	22.6%	22.5%	12.4%
% main thread killed due to branch-misp.	98.8%	95.8%	73.9%	92.5%	76.9%

Table 7.5. CINT2000 Branch statistics in FSG-RA-all

dependent on the runahead trigger load's data. It is highly possible that the main thread can stay on the correct program path. Accordingly, under the two-thread running mode, on an average, only 13.1% main thread executions are stopped prematurely since the recovery thread detects a branch mis-prediction. On the other hand, it is relatively difficult for FSG-RA to keep the main thread running on the correct path on CINT2000 benchmarks, since there are 119.6 branches in every 1000 pseudo-retired instructions and 12.4% of them are miss-dependent. Shown in Table 7.5, 76.9% main thread executions are killed due to mis-predictions of miss-dependent branches. Mis-predicted branches which depend on the missing load is a main factor for terminating the runahead mode.

7.9 Related Work

Karkhanis and Smith [25] showed that the structural blockages due to a full ROB are the major reason behind performance loss with cache misses. If the structural resources are available, the processor can continuously issue instructions and overlap even very long cache miss latencies. However, they also concluded that mis-predicted branches

which depend on the missing load will inhibit performance in some cases because useful instruction issue stops immediately after the mis-predicted branch. Our work supports their observations and analyses.

Zhou proposed the dual-core execution (DCE) micro-architecture [60]. The front processor always runs far ahead to warm up the data cache. It commits a branch-fixed instruction stream to the back processor. In the term of processor states, the back processor is responsible for correcting the state of the missing load and its dependent instructions. However, in DCE, all instructions are executed twice, once by the front processor and once the back-end processor. Since *FSG-RA only re-executes instructions when the main thread is under the runahead mode*, there are significant power/energy differences between the two approaches.

Srinivasan *et al.* proposed the continual flow pipelines (CFP) [51]. Unlike CFP which employs a *sequential recovery* method and utilizes very large hierarchical load and store queues to buffer all in-flight load and store instructions, FSG-RA needs small load/store queues and forks a second thread to verify and maintain the processor state. FSG-RA is a fully parallel recovery FSP model.

The SlipStream paradigm [53, 42] uses the A-stream to reduce the length of a running program by dynamically skipping ineffective instructions. The R-stream uses the A-stream's outcomes only as predictions. FSG-RA utilizes multiple threads only when L2 misses occur.

7.10 Summary of FSG-RA

We have presented an exploration of fine-grain state processor. Our results indicate that by allowing a processor to continue execution with a partially correct state and repairing the state in parallel by using a second thread may prove valuable in addressing the speed gap between the memory and the high-performance processors of today.

Chapter 8

Conclusion

Processor states can be manipulated either at the coarse-grain level or at the fine-grain level. If appropriate mechanisms have been implemented to answer queries regarding the current state of data values at a finer granularity level, it is possible to salvage part of the work done during speculative execution after a mis-speculation, or to recover only the part of the state that has been damaged without a roll-back. Conventional fine-grain state guided speculation recovery methods are ad-hoc and fail to explore the full potential of fine-grain processor state handling, namely the parallelism in speculation recovery. This dissertation introduces the concept of FSP and provides a general FSP framework for handling the processor state at a fine-grain level and lays the foundations of parallel speculation recovery.

Under this general framework, a processor can continue execution past a mis-speculation resolution point before the state is fully recovered. In such an organization, newly fetched instructions which access incorrect speculative values are blocked until the correct data are restored; however, those instructions that access the correct values continue execution while the recovery occurs. In parallel with the recovery, the processor is capable of moving

forward seamlessly with a partially correct state. Thus, the long latency of mis-speculation recovery is overlapped with useful execution.

Based on the proposed framework, this dissertation has explored applications of FSP on a sophisticated uni-processor setting as well as a simple multi-core/multi-threaded organization. It has presented two detailed FSP models, EMR and FSG-RA, regarding control speculation and value speculation, respectively. Both models have demonstrated that the FSP technique handles processor states more efficiently and obtains much higher performance than the traditional CSP mechanism does.

In next section, the contributions of this dissertation are summarized, and the future research directions are briefly discussed.

8.1 Dissertation Contributions

This dissertation has made the following contributions:

1. The concept of *Fine-grain State Processors (FSP)*. FSP can utilize partially correct processor state upon an exception, which cannot be seen and used by *Coarse-grain State Processors (CSP)*.

2. A taxonomy of fine-grain state processors. The taxonomy summarizes and categorizes existing fine-grain state guided speculation recovery mechanisms.

3. A general FSP framework, including a novel concept of exploiting parallelism in speculation recovery. This framework is made of the following properties:

    - *Identification property.*

- *Block and shelve property.*
- *Correction property.*
- *Unblocking property.*
- *Parallelism-in-recovery property.*

4. An FSP model for control speculation, *Eager branch Misprediction (EMR)*. EMR obtains an average performance speedup of 9.0% over the traditional RMAP on CINT2000. Moreover, it achieves 99% of the performance obtained by an unlimited checkpoint recovery method using only 4 checkpoints.

5. An FSP model for value speculation, *Fine-grain State Guided Runahead execution (FSG-RA)* which investigates value speculation by using runahead execution. FSG-RA improves the single-thread program's performance by exploiting the parallelism in the Runahead execution recovery in a multi-thread processor environment. FSG-RA can obtain an average of 38.9% and up to 160.0% better performance than coarse-grain baseline processor on the SPEC CFP2000 benchmark suite.

6. An extension of the store set algorithm to detect the inter-thread memory dependences. This algorithm is applicable for not only the FSG-RA model, but also other general multi-threaded processor models.

7. Implementation of a cycle-accurate and function-driven cache hierarchy simulator using *ADL* and its integration into the *FAST* simulation system.

8.2 Future Directions for Research

This work has focused on a uni-processor configuration which has only a single core, and the processor states are private to this core. Under such a uni-processor environment, EMR and FSG-RA models have demonstrated that the FSP techniques can provide impressive speed-ups without using difficult to scale processor elements. With the fine-grain state concept, mis-speculation recovery essentially becomes free if there is enough independent work to do for the processor.

In the near future, we believe more and more processor cores will be integrated into a single die [3, 22]. In addition to increasing throughput, efficiently utilizing multi-threading or multi-core resources will play a crucial role to achieve high performance and to reduce the hardware complexity. Furthermore, aggressive speculation mechanisms become more feasible than ever by using the FSP techniques developed in this dissertation. In the following, future directions and possible extensions of the current FSP techniques are briefly discussed:

Architectural State Memory

In Chapter 7.7.2, FSG-RA-all model allows store instructions speculatively commit their values into the memory. Thus, the memory contains both in-order values and speculative values. Also, FSG-RA-all utilizes a *History List* as the recovery method. Before a store instruction is retired and committed into the memory, the data in that location is read and inserted into a History List. If it needs to roll back to the most recently retired handle point,

the history list is used to restore the in-order memory state.

This memory design is referred to as *Architectural State Memory (ASM)*. Under a uni-processor environment, the ASM method works well using the History List as the recovery method. However, under a multi-processor environment, a single uni-processor History List is not enough. Assuming a centralized shared memory is used, processors need to communicate with each other through the shared memory. When ASM is employed in such a configuration, a speculative commit into the memory must be known by other processors.

A simple way is to append a speculative bit with each memory line or even word to indicate this memory line (word) is speculative or not. It is also possible to extend the History List based algorithm to a multi-processor environment. If a processor needs to read the in-order value from a particular address, it needs to access ASM and the History Lists of other processors simultaneously.

Fine-grain State Multi-core Processors

Once a practical micro-architecture mechanism of the multi-core version ASM is designed, the concept of Fine-grain State Processors can be extended into *Fine-grain State Multi-core Processors (FSMP)*.

Like FSP, FSMP would have the ability to identify machine states on an individual value basis. Each single core of an FSMP should be able to precisely identify the fine-grain register state of its own, as well as the fine-grain shared memory state of the whole multi-core processor. Therefore, the concept of fine-grain state can be utilized in the context of not only the intra-core speculative executions but also the inter-core ones.

For example, when a speculation is detected as a miss, the core executing the program can continue running with a partially correct state, other idle cores can be used to repair the state. Similar to FSG-RA, which can improve a single-thread program's performance by exploiting the parallelism in the Runahead execution recovery in a multi-thread processor environment, such an FSMP model will be able to speed up a single-thread program's performance by exploiting the parallelism-in-recovery in a multi-core processor configuration.

Thread Level Speculation

An attractive speculative execution technique utilized in a multi-core environment is *Thread Level Speculation (TLS)*. With more cores integrated into a single die, different TLS methods [30, 18, 31, 12, 52, 32] have been proposed to utilize available hardware resources to optimistically execute non-analyzable serial programs in parallel to boost performance.

With TLS, a sequential program is divided into tasks which can be executed in parallel by different cores. There is one non-speculative main task which precedes all other speculative tasks. If any inter-task dependence violation is detected, either control dependence or data dependence, incorrect tasks need to squash and polluted processor states need to be repaired. Utilizing traditional TLS techniques, coarse-grain state multi-core processors have to squash tasks, repair states, and then re-execute tasks from speculation point, sequentially. An improvement mechanism [45] would re-execute only speculation dependent instructions, which is categorized as Sequential Recovery fine-grain state scheme introduced in Chapter 4.2.1.

In FSMP, each core would have the ability to precisely identify state at a single register or a single memory location basis. Thus, an FSMP would be able to apply the concept of fine-grain state in the context of TLS. Once a dependence violation occurs, the main task can continue execution, and can even fork new tasks with a partially correct processor state. Only dependent instructions will be re-executed to restore the correct state in parallel with the main task.

Dual Path Branch Execution.

In order to reduce branch mis-prediction penalty, several dual path branch execution techniques have been proposed in [54, 19, 27, 28, 56, 4]. Normally, a dual path execution mechanism uses a spare hardware context to process the alternative path of a hard-to-predict branch at the same time as the predicted path is being executed. Thus, mis-prediction penalty can be significantly reduced if the branch is mis-predicted. However, most proposed models belong to the CSP category and they have to maintain the complete set of states for each path. It makes the dual path execution quite complex.

Unlike CSP, an FSP or an FSMP should be able to identify individual data items belonging to the either path of an branch instruction. At a hard-to-predict branch instruction, an FSP or an FSMP can follow both paths and would not need to maintain the complete set of machine states for each path. It can even commit speculative memory values into ASM without utilizing a complex store buffer. Thus, the hardware complexity can be reduced and the dual path execution can be processed at deeper depth.

Moreover, passing the convergence point of branch, control-independent instructions

can be processed seamlessly, whether they are data- dependent or independent on any values along both paths of the predicted branch. Data-dependent instructions can be simply blocked and they can wait for the correct values to be resolved. Once the branch is resolved and the correct values are recognized, execution of these instructions can be resumed. At the same time, processor can execute data-independent instructions in parallel. Therefore, the *parallelism-in-resolution* can be achieved to boost performance in such a setting as well.

Bibliography

[1] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *Proceedings of the 36th International Symposium on Microarchitecture*, pages 423–434, December 2003.

[2] H. Akkary, R. Rajwar, and S. T. Srinivasan. An analysis of a resource efficient checkpoint architecture. *ACM Transactions on Architecture and Code Optimization*, Volume 1:418–444, December 2004.

[3] AMD. Six-core amd opteron processor. *http://www.amd.com*.

[4] J. L. Aragon, J. Gonzalez, A. Gonzalez, and J. E. Smith. Dual path instruction processing. In *Proceedings of the 2002 International Conference on Supercomputing*, pages 220–229, June 2002.

[5] D. N. Armstrong, H. Kim, O.Mutlu, and Y. N. Patt. Wrong path events: Exploiting unusual and illegal program behavior for early misprediction detection and recovery. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO-37)*, pages 119–128, Portland, Oregon, December 2004.

[6] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The m5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, 2006.

[7] D. C. Burger and T. M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report CS-TR-97-1342, Computer Science Department, University of Wisconsin Madison, 1997.

[8] C.-Y. Cher and T. N. Vijaykumar. Skipper: a microarchitecture for exploiting control-flow independence. In *MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 4–15, Washington, DC, USA, 2001. IEEE Computer Society.

[9] I.-C. K. Chih-Chieh Lee and T. N. Mudge. The bi-mode branch predictor. In *The 30th Annual IEEE-ACM International Symposium on Microarchitecture*, pages –, December 1997.

[10] Y. C. Chou, J. Fung, and J. P. Shen. Reducing branch misprediction penalties via dynamic control independence detection. In *Proceedings of the 13th ACM International Conference on Supercomputing*, pages 109–118, 1999.

[11] G. Z. Chrysos and J. S. Emer. Memory dependence prediction using store sets. In *Proceedings of the 25th International Conference on Computer Architecture*, pages 142–153, June 1998.

[12] M. Cintra, J. F. Martínez, and J. Torrellas. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 13–24, New York, NY, USA, 2000. ACM.

[13] COMPAQ. Alpha 21264 microprocessor hardware reference manual. July 1999.

[14] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the 1997 ACM International Conference on Supercomputing*, pages 68–75, Vienna, Austria, July 1997.

[15] J. Edler and M. D. Hill. Dinero iv trace-driven uniprocessor cache simulator. *http://pages.cs.wisc.edu/ markhill/DineroIV*.

[16] A. Gandhi, H. Akkary, and S. T. Srinivasan. Reducing branch misprediction penalty via selective branch recovery. *Proceedings of the 10th International Symposium on High-Performance Computer Architecture*, pages 254–264, February 2004.

[17] M. J. Garzarán, M. Prvulovic, J. M. Llabería, V. Vi nals, L. Rauchwerger, and J. Torrellas. Tradeoffs in buffering memory state for thread-level speculation in multiprocessors. In *HPCA*

'03: Proceedings of the 9th International Symposium on High-Performance Computer Architecture, page 191, Washington, DC, USA, 2003. IEEE Computer Society.

[18] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. *SIGOPS Oper. Syst. Rev.*, 32(5):58–69, 1998.

[19] T. H. Heil and J. E. Smith. Selective dual path execution. Technical report, University of Wisconsin Technical Report, 1997.

[20] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the pentium 4 processor. In *Intel Technology Journal*, February 2001.

[21] W. W. Hwu and Y. N. Patt. Checkpoint repair for out-of-order execution machines. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 18–26, June 1987.

[22] Intel. Intel xeon 'nehalem-ex' processor. *http://www.intel.com*.

[23] D. A. Jimenez and C. Lin. Dynamic branch prediction with perceptrons. *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, pages 197–206, January 2001.

[24] M. Johnson. *Superscalar Microprocessor Design*. Prentice Hall, 1991.

[25] T. Karkhanis and J. E. Smith. A day in the life of a data cache miss. In *Workshop on Memory Performance Issues*, Anchorage, AK, May 2002.

[26] R. Kessler, E. McLellan, and D. Webb. The alpha 21264 microprocessor architecture. In *International Conference on Computer Design*, December 1998.

[27] A. Klauser. *Reducing branch misprediction penalty through multipath execution*. PhD thesis, Boulder, CO, USA, 1999. Director-Grunwald, Dirk.

[28] A. Klauser, A. Paithankar, and D. Grunwald. Selective eager execution on the polypath architecture. In *ISCA '98: Proceedings of the 25th annual international symposium on Computer architecture*, pages 250–259, Washington, DC, USA, 1998. IEEE Computer Society.

[29] A. KleinOsowski and D. J. Lilja. Minnespec: A new spec benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, Volume 1, June 2002.

[30] V. Krishnan and J. Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Trans. Comput.*, 48(9):866–880, 1999.

[31] P. Marcuello and A. González. Clustered speculative multithreaded processors. In *ICS '99: Proceedings of the 13th international conference on Supercomputing*, pages 365–372, New York, NY, USA, 1999. ACM.

[32] J. F. Martínez and J. Torrellas. Speculative synchronization: applying thread-level speculation to explicitly parallel applications. *SIGPLAN Not.*, 37(10):18–29, 2002.

[33] S. McFarling. Combining branch predictors. Technical Report WRL-TN-36, Digital Western Research Laboratory, 1993.

[34] S. McFarling and J. Hennesey. Reducing the cost of branches. *SIGARCH Comput. Archit. News*, 14(2):396–403, 1986.

[35] O. Mutlu, H. Kim, and Y. N. Patt. On reusing the results of pre-executed instructions in a runahead execution processor. In *Computer Architecture Letters*, January 2005.

[36] O. Mutlu, H. Kim, and Y. N. Patt. Techniques for efficient processing in runahead execution engines. In *Proceedings of the 32st International Symposium on Computer Architecture*, pages 370–381, Madison, WI, June 2005.

[37] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An effective alternative to large instruction windows. *IEEE Micro*, 23(6):20–25, 2003.

[38] S. Önder and R. Gupta. Automatic generation of microarchitecture simulators. In *IEEE International Conference on Computer Languages*, pages 80–89, Chicago, May 1998.

[39] S. Önder and R. Gupta. Dynamic memory disambiguation in the presence of out-of-order store issuing. In *32nd Annual IEEE-ACM International Symposium on Microarchitecture*, pages 170 – 176, November 1999.

[40] S. Palacharla, N. P. Jouppi, and J. E. Smith. Quantifying the complexity of superscalar processors. Technical Report CS-TR-96-1328, University of Wisconsin Technical Report, 1996.

[41] C. Price. *MIPS IV Instruction Set Revision 3.2*. MIPS Technologies Inc., September 1995.

[42] Z. Purser, K. Sundaramoorthy, , and E. Rotenberg. A study of slipstream processors. In *Proceedings of the 33th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 269–280, Monterey, CA, December 2000.

[43] E. Rotenberg, Q. Jacobson, and J. Smith. A study of control independence in superscalar processors. In *HPCA '99: Proceedings of the 5th International Symposium on High Performance Computer Architecture*, page 115, Washington, DC, USA, 1999. IEEE Computer Society.

[44] A. Roth and G. S. Sohi. Register integration: a simple and efficient implementation of squash reuse. In *Proceedings of the 33th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 223–234, Monterey, CA, December 2000.

[45] S. R. Sarangi, W. Liu, J. Torrellas, and Y. Zhou. Reslice: Selective re-execution of long-retired misspeculated instructions using forward slicing. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*, Barcelona, Spain, November 2005. IEEE Computer Society.

[46] D. Sima, T. Fountain, and P. Kacsuk. *Advanced Computer Architectures, A Design Space Approach*. ADDISON-WESLEY, 1997.

[47] J. E. Smith and A. R. Pleszkun. Implementing precise interrupts in pipelined processors. *IEEE Trans. Computers*, 37(5):562–573, 1988.

[48] A. Sodani and G. S. Sohi. Dynamic instruction reuse. In *Proceedings of the 24th International Conference on Computer Architecture*, 1997.

[49] SPEC. http://www.spec.org/cpu2000/.

[50] E. Sprangle, R. S. Chappell, M. Alsup, and Y. N. Patt. The agree predictor:a mechanism for reducing negative branch history interference. In *Proceedings of the 24th International Conference on Computer Architecture*, pages 284–291, 1997.

[51] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual flow pipelines. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 107–119, New York, NY, USA, 2004. ACM Press.

[52] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. *SIGARCH Comput. Archit. News*, 28(2):1–12, 2000.

[53] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream processors: Improving both performance and fault tolerance. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, 2000.

[54] A. K. Uht, V. Sindagi, and K. Hall. Disjoint eager execution: an optimal form of speculative execution. In *MICRO 28: Proceedings of the 28th annual international symposium on Microarchitecture*, pages 313–325, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press.

[55] T. Ungerer, B. Robic, and J. Silc. A survey of processors with explicit multithreading. In *ACM Computing Surveys*, volume 35, pages 29–63. ACM, March 2003.

[56] S. Wallace, B. Calder, and D. M. Tullsen. Threaded multiple path execution. In *ISCA '98: Proceedings of the 25th annual international symposium on Computer architecture*, pages 238–249, Washington, DC, USA, 1998. IEEE Computer Society.

[57] M. Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[58] K. C. Yeager. The mips r10000 superscalar microprocessor. In *IEEE Micro*, pages 28–44, April 1996.

[59] T.-Y. Yeh and Y. N. Patt. Alternative implementations of two-level adaptive branch prediction. In *Proceedings of the 19th International Conference on Computer Architecture*, pages 124–134, 1992.

[60] H. Zhou. Dual-core execution: Building a highly scalable single-thread instruction window. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, pages 231–242, Washington, DC, 2005. IEEE Computer Society.

[61] P. Zhou and S. Önder. Improving single-thread performance with fine-grain state maintenance. In *CF '08: Proceedings of the 2008 conference on Computing frontiers*, pages 251–260, New York, NY, USA, 2008. ACM.

[62] P. Zhou, S. Önder, and S. Carr. Fast branch misprediction recovery in out-of-order superscalar processors. In *Proceedings of the 2005 ACM International Conference on Supercomputing*, pages 41–50, Boston, MA, June 2005.