The Dissertation Committee for Rajagopalan Desikan
certifies that this is the approved version of the following dissertation:

Distributed Selective Re-Execution for EDGE Architectures

Committee:

Douglas C. Burger, Supervisor

Lizy Kurian John

Stephen W. Keckler

Craig M. Chase

Donald S. Fussell

Kathryn S. McKinley

Distributed Selective Re-Execution for EDGE Architectures

by

Rajagopalan Desikan, B.E., M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2005

Acknowledgments

"Gratitude bestows reverence, allowing us to encounter everyday epiphanies, those transcendent moments of awe that change forever how we experience life and the world."
–John Milton

Numerous people have played an important role in helping me tide over the rigors of graduate school. I would like to first thank my advisor, Doug Burger, for his advice, guidance, and training that helped me get to this point in my graduate career. Doug has been an excellent mentor, guide, and friend to me, and the things I have learned from him will stay with me for life.

Steve Keckler, as the the co-leader of the CART group, has also played an important part in my professional development. I am thankful for the many opportunities I have had to interact with him, and for his numerous insightful comments and suggestions.

I would like to acknowledge the moral and technical support that I have received from the students in my research group, CART. Karu Sankaralingam and Ramdas Nagarajan wrote the original GPA simulator that I used for my initial evaluation. I would like to thank them for their patience in answering my innumerable questions. I would also like thank everybody who helped design and implement the TRIPS prototype simulator, and Bill Yoder for swiftly dealing with any compiler related issues that I happened to come across

during my research. My thanks also go out to Robert McDonald for designing the TRIPS prototype simulator framework, and for the *first-store* dependence predictor proposal. The constructive criticism provided by the students in the CART group during my different practice talks have greatly helped me hone my presentation skills, and I am grateful to them for that. Finally, I would like to thank my roommate for the last 5 years, Rubin Sidhu, and everyone on the ACES 3^{rd} floor, who made it such a fun and exciting place to work.

I would also like to acknowledge the institutions that helped support my research in graduate school: the Intel Research Council for my fellowship and equipment, National Science Foundation for the initial grants that funded me, and DARPA for funding the TRIPS-related aspect of my research.

Last, but not the least, I would like thank my family for helping me reach this milestone in my life. My sisters and my parents provided me with the constant encouragement necessary for success in graduate school, and I could not have done it without them.

Distributed Selective Re-Execution for EDGE Architectures

Publication No. _____

Rajagopalan Desikan, Ph.D.
The University of Texas at Austin, 2005

Supervisor: Douglas C. Burger

Speculation is a key technique that modern processors use to achieve high performance. Traditionally, speculation meant control speculation, in which the processor predicts the outcome of control instructions when they are fetched, and validates the prediction when the instructions are executed. More recently, processors have adopted another form of speculation called data speculation to improve performance. Data speculation involves the prediction of the data values produced by instructions, and forwarding the predicted values to consumers in the data-flow graph. For both control and data speculation, mis-speculation recovery is required when the speculation is incorrect.

The conventional mechanism for mis-speculation recovery consists of flushing the processor pipeline of all incorrect state and restarting execution from the corrected state. However, pipeline flushes have become increasingly

expensive in modern microprocessors with large instruction windows and deep pipelines. Selective re-execution is a technique that can reduce the penalty of mis-speculation recovery by re-executing only instructions that received incorrect values due to the mis-speculation. Conventional mechanisms to implement selective re-execution have had limited success because of the enormous complexity involved in the implementation.

In this dissertation, we introduce a new selective re-execution mechanism that exploits the properties of a dataflow-like Explicit Data Graph Execution (EDGE) architecture to support efficient mis-speculation recovery, while scaling to large window sizes. This Distributed Selective Re-Execution (DSRE) mechanism permits multiple speculative waves of computation to traverse a dataflow graph simultaneously. The mechanism has no centralized state, and uses simple state bits to determine instructions to re-fire on a mis-speculation, thus reducing the complexity of selective re-execution.

We evaluate DSRE as a recovery mechanism for load-store dependence mis-speculation on a high-level EDGE architecture simulator, the Grid Processor Architecture (GPA) simulator, and on the more detailed TRIPS prototype processor simulator. DSRE provides 17% and 4.2% speedup, respectively, over dependence prediction, on the two simulators. Our results show that DSRE needs to be used in conjunction with pipeline flushing to achieve high performance. Predictors need to be aware of the the costs associated with each mechanism, and use the appropriate recovery mechanism for each speculation.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Speculation has become an increasingly important technique in processors to achieve high performance. Modern processors like the Alpha 21264 and Pentium IV already have more than half a dozen predictors in various pipeline stages. This trend is likely to continue, as growing wire delays in current and future technologies will force microarchitectural structures within the processor to make decisions with incomplete information, thus requiring more speculative techniques to achieve high performance.

However, the trend towards large instruction windows will result in large mis-speculation penalties, both in terms of performance and power, using the conventional mis-speculation recovery mechanism of pipeline flushing. Pipeline flushing results in the processor re-executing all instructions after a mis-speculating instruction, irrespective of whether the instruction executed correctly the first time.

Thus, the number of predictors, the number of mispredictions, and the cost of each misprediction are all likely to increase, forcing future processors to spend larger fractions of execution time recovering from mispredictions. Hence, to achieve high performance using aggressive speculation, future processors will

need a mechanism for efficient, low-cost recovery from mis-speculations. This need is evident from Figure 1.1.

Figure 1.1 shows the performance of a high IPC processor, with different mis-speculation costs, as a function of the number of mis-speculations per thousand instructions. The data represented in the graph is not empirical, and assumes an ideal machine in which the only source for performance degradation is pipeline flushes due to mis-speculations. The solid lines represent future machines that have a peak throughput of 8 instructions per cycle, with varying mis-speculation costs. The dashed line represents a present day superscalar processor that has a peak IPC of 4.

We see from Figure 1.1 that with increasing mis-speculation costs, even a small number of mis-speculations can result in a significant drop in performance. Since the graph is not an experimental curve and represents worst case scenario for a particular issue width, the actual drop in performance due to pipeline flushes will likely be lower, because of other performance constraining factors in the processor. However, Figure 1.1 does illustrate the general future trend of increasing mis-speculation cost.

1.1 Efficient Mis-speculation Recovery

There are a number of ways of to alleviate the performance loss due to mis-speculations. Processors can have fewer predictors in their pipelines thus reducing the number of predictions, and consequently, mis-speculations. Processors can tradeoff area and power for complex predictors that are more

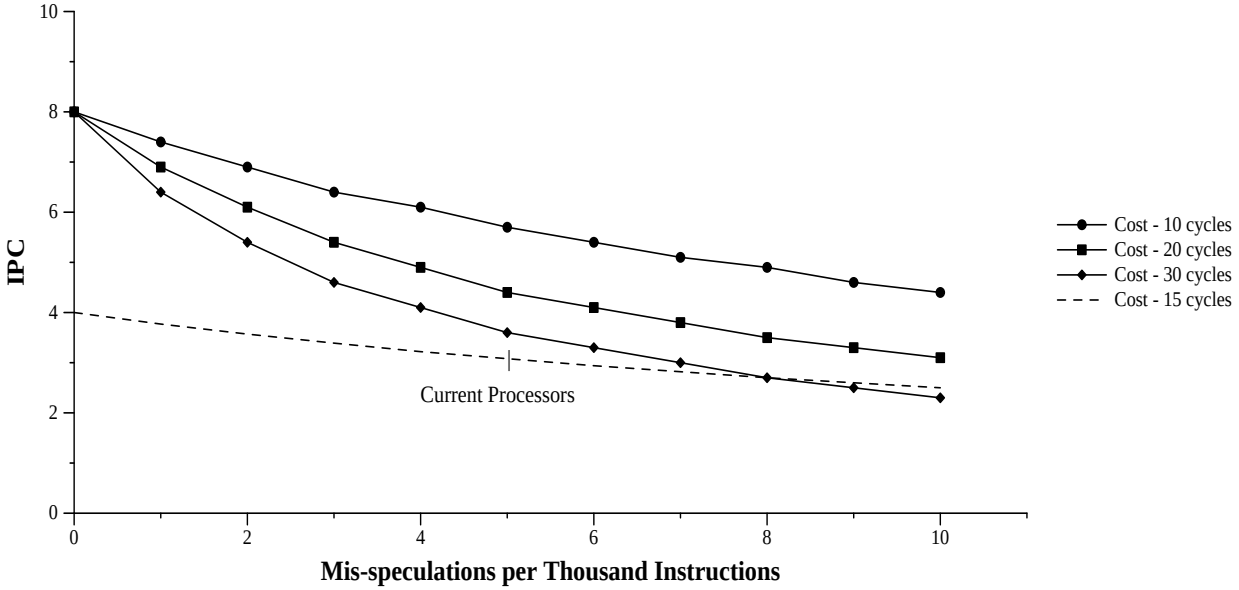

Figure 1.1: Variation in IPC with mis-speculation cost and rate

accurate, and hence incur fewer mis-speculations. Finally, more efficient mis-speculation recovery mechanisms can reduce performance losses. In this dissertation, we look at efficient mis-speculation recovery to reduce the performance loss due to mis-speculations.

One mechanism for efficient data mis-speculation recovery involves selectively re-executing only those instructions that produced incorrect values, by identifying executed instructions in the processor that are part of the data flow graph (DFG) of the mis-speculating instruction. This technique, called selective re-execution (SRE), is implemented in a limited fashion in modern processors. For example, both the Alpha 21264 [32] and the Pentium 4 [24] use scheduling speculation to schedule the consumers of loads, and use a limited

form of SRE to recover from mis-speculations arising from an incorrect schedule. However, making SRE more general in future conventional processors will become progressively more difficult due to the following three challenges:

1. Tracking and maintaining dependences between large amounts of in-flight state.

2. The increasing physical distance between the distributed detection of violations and the centralized recovery control.

3. The complexity of having many predictions in flight from multiple, distributed heterogeneous predictors.

In a recent study of various current and proposed re-execution schemes, Kim and Lipasti [33] conclude that "universal selective replay, where an instruction can cause a recovery event at any point during its lifetime, is barely feasible for current-generation designs, and does not scale to wider machines or additional types of speculation." These challenges need to be overcome to support aggressive speculation in future processors.

1.2 Explicit Data Graph Execution (EDGE): A New Architecture Model

Explicit Data Graph Execution (EDGE) instruction set architectures (ISA) are a new class of instruction set architectures that are designed to scale to high performance in future communication-dominated technologies [3, 49].

The two main characteristics of EDGE ISA are block atomic execution model, wherein blocks of instructions are fetched and committed atomically and direct instruction communication, wherein an instruction sends its results directly to its consumers without using a shared namespace. Using direct instruction communication, EDGE architectures overcome the first difficulty encountered in implementing selective re-execution in conventional architectures. EDGE architectures permit limited dataflow execution within defined program regions, and conventional execution across those regions, with sequential memory, exception, and register semantics, and a conventional programming model. The explicit representation of the dataflow graph in the EDGE ISA obviates the dynamic reconstruction of data dependences in the processor.

1.2.1 The Tera-op, Reliable, Intelligently adaptive Processing System (TRIPS): An EDGE Architecture Implementation

The TRIPS processor is one particular initial implementation of the EDGE architecture [30, 60]. We use the EDGE-based TRIPS architecture as the platform for implementing distributed selective re-execution. The EDGE ISA specifies a block atomic execution model. Hence in the TRIPS processor, we compile programs into sets of instructions called hyperblocks. During program execution, the processor fetches the hyperblocks atomically and places the instructions within a block in the reservation stations of the arithmetic logic units (ALUs) in the processor. Instructions execute in dataflow fashion, firing when they receive all their input operands, and forwarding results to their consumers.

The processor commits a block atomically when all the block has produced all its outputs—the register writes, the stores, and a branch. Since the instructions in a block remain in the reservation stations until the block is committed, instruction refiring can be initiated by simply sending new values of the input operands. Thus, recovery from mis-speculation can be accomplished in a completely distributed fashion. The block atomic execution model overcomes the second difficulty encountered in implementing selective re-execution in conventional architectures, where the recovery needs to be centralized. The proposed selective re-execution mechanism takes advantage of this characteristic of TRIPS processors.

1.2.2 Distributed Selective Re-Execution (DSRE) for TRIPS

The explicit representation of the dataflow graph and the block atomic execution model in the EDGE ISA lend themselves to the efficient implementation of a distributed, selective re-execution mechanism (DSRE). The proposed mechanism provides a simple technique that multiple heterogeneous predictors can use for mis-speculation recovery, while scaling to both increasing instruction window sizes and wire delays. The general nature of the recovery mechanism overcomes the last difficulty encountered in implementing selective re-execution in conventional architectures, by decoupling the underlying recovery mechanism from the type of data speculation that uses it for recovery.

The DSRE mechanism enables multiple "waves" of speculative execution to traverse the dataflow graph simultaneously. However, we need a

mechanism to detect the non-speculative value for each operand. The solution we explore in this dissertation involves associating a commit bit with each operand in the machine. This bit is set for an operand when it becomes non-data speculative. Thus, to ensure that the right answer is eventually produced and committed, a "commit wave" traverses the DFG behind the waves of speculative execution, and ensures that the correct results are eventually saved. In this dissertation, we propose two techniques to accelerate the commit wave, which can become the bottleneck in this scheme. We use DSRE to increase the performance with two types of data speculation, load-store dependence speculation and last-value prediction.

The DSRE mechanism, which enables lightweight recovery from load/store order violations, is not limited to dependence prediction recovery. The same mechanism can be used to recover from any data value mis-speculation—including other types of value predictors, such as last-value prediction, stride prediction, predicate prediction, and coherence speculation—as well as recovery from soft errors. Since the DSRE mechanism we propose uses only point-to-point messages to implement recovery, it is ideal for distributed microarchitectures built in future technologies, and may be an enabling technology that supports new types of speculation or execution on highly unreliable computational substrates.

The DSRE mechanism described in this dissertation has significant limitations. First, it increases the contention within the processor, both for the operand network and for the ALUs. Second, the serial nature of speculation

validation can result in the commit wave falling significantly behind the execution wave. Third, having speculative multiple versions of an operand can result in unexpected race conditions within the processor. We validated the mechanism on a low-level prototype simulator to delineate some of these issues. However, our simulation infrastructure is limitated due to its slow simulation speed, and the compiled benchmarks have sub-optimal code with redundant loads and stores. Future work in DSRE can involve tackling those issues.

1.3 Thesis Statement

This dissertation proposes Distributed Selective Re-Execution(DSRE) as an alternative, low-cost mechanism for recovering from data mis-speculations. We evaluate the overhead associated with DSRE, and suggest potential uses in future, large instruction window processors.

1.4 Dissertation Contributions

This dissertation makes the following contributions:

1. Identify the importance of efficient mis-speculation recovery in large instruction window machines.

2. Enumerate the drawbacks of current schemes for mis-speculation recovery.

3. Explain the features of a new instruction set architecture (EDGE), and

its one particular implementation (TRIPS) that make it amenable to implementation of efficient mis-speculation recovery.

4. Propose a selective re-execution scheme, Distributed Selective Re-Execution (DSRE) that is simple, distributed, and supports many different forms of data speculation.

5. Evaluate DSRE on a research simulator for one type of data speculation, load-store dependence speculation. Identify bottlenecks in DSRE and evaluate methods for overcoming these bottlenecks.

6. Validate DSRE on a simulator that faithfully models the TRIPS prototype processor. Expose challenges involved with implementing selective re-execution on a simulator that models the low-level hardware details, and propose mechanisms for overcoming these challenges.

7. Explain implementation complexity of DSRE.

8. Identify the importance of using the appropriate recovery scheme for each type of speculation in future processors.

9. Suggest additional uses for DSRE.

1.5 Background and Related Work

Mis-speculation recovery has been a subject of active research ever since control speculation was first introduced in processors. Control speculation,

also known as branch prediction, is one class of speculation that has been extensively researched in computer architecture. Branch misprediction recovery in conventional processors is done by pipeline flushing. Due to the high cost of mis-speculation recovery, researchers have examined ways to reduce mis-speculation by improving branch predictor accuracy. Thus, branch predictors have evolved from simple 2-bit tables to complex, two-level predictors that track multiple histories [35, 36, 43, 64, 71, 72]. More recently, researchers have looked at neural branch predictors to increase predictor accuracy [1, 26, 27, 41].

Data speculation is another class of speculation that is increasingly important in modern processors. Data speculation involves predicting the values of operands before they are computed, using the predicted values to accelerate program execution. Lipasti et al. [40] introduced the notion of value locality and described methods to capture it to perform load value prediction. The authors propose microarchitectural enhancements (load value prediction) to PowerPC 620 and Alpha 21164 to predict 32 and 64 bit register values, and get 3% and 6% average improvement in performance, respectively. Sazeides and Smith defined two predictor models for value prediction—computational predictors and context predictors [61]. The authors perform simulations with unbounded prediction tables, and find highly predictable data values in SPEC95 benchmarks.

Wang and Franklin [68] investigated a variety of data value prediction schemes including stride-based and pattern-based two-level predictors, and a hybrid predictor combining the two schemes. The authors find that the hybrid

predictor was able to correctly predict 50-80% of register-result producing instructions, with the percentage of mispredictions ranging from 5-18%. A number of other papers have examined various aspects of data value speculation in uni-processors [4, 5, 19, 37, 39, 42, 55, 69]. Researchers have also proposed a number of data speculation techniques for improving the performance of multi-processors [28, 29, 34, 47, 52].

With the increasing number of predictors resulting in increasing mis-speculations, many researchers have explored and are exploring selective re-execution to defray growing mis-speculation costs. Some of the earliest work in selective re-execution was done by Rotenberg et al. [57], who discussed applying selective re-execution to both control and data mis-speculations recovery for Trace Processors. Selective re-execution for control prediction exploits control independence [58], and can be used for techniques like out-of-order fetch [66]. Researchers have also looked at techniques like dynamic instruction reuse for dynamically reusing the results of instructions with the same inputs, and squash reuse to reuse values in the register file that did not change due to the mis-speculation [59, 65]. More recently, researchers have looked at exact convergence to reuse the results computed after the convergence of control paths [21].

Calder et al. [7, 8] showed that selective re-execution coupled with dependence prediction can—in a centralized microarchitecture with small issue windows—approach the performance of a perfect dependence predictor. Those techniques are insufficient to provide the same gain on distributed microarchi-

tectures with much bigger (1000+ entry) instruction windows, which is the problem that we address.

Despite its potential benefit, implementation complexities prevent current selective re-execution schemes from being used as a single unified recovery mechanism for multiple types of data value speculation. Recent patents from AMD [31], Sun Microsystems [50], and Intel [44, 45] propose selective re-execution for recovering only from load scheduling speculation, using signals from the lower-level cache [50] or circular queues [44, 45] to facilitate the re-execution schemes. Multiple disparate recovery modes are used due to the design complexity introduced by interaction among distinct types of speculation, complexity which is exacerbated by slowing global wires. Slowing communication is causing multi-cycle delays between misprediction detection and reporting, which will grow progressively worse if speculation resolution remains centralized. Ernst et al. [15] also made this observation in their recent work. The selective re-execution scheme that we propose in this dissertation does not suffer from these challenges, since it provides a single, distributed framework for handling potentially many types of speculation simultaneously.

Zhou et al. [74] identify the challenges associated with implementing aggressive selective re-execution on a conventional superscalar processor, which include retention of issued instructions that may be re-executed, the reissue mechanism itself, and the data-dependence driven identification of the set of instructions to be re-executed. Of the solutions they describe, the re-order buffer (ROB) augmentation that holds instructions in the window until committed

is most similar to the DSRE mechanism proposed in this work. However, their approach is not scalable to larger windows and distributed microarchitectures, nor does it eliminate the performance losses associated with their proposed solutions to the other two challenges—the complexity of the reissue mechanism and the data-dependence driven identification of the set of instructions to re-execute. Ernst et al. [16] present a mechanism for a dynamic scheduler that uses selective replay without using broadcast communication. The selective replay mechanism presented in their work is specific to recovering from scheduling speculation.

Finally, Kim and Lipasti [33] recently looked at various replay schemes for conventional superscalar processors. The paper finds that current and proposed replay schemes do not scale well to future, large instruction window machines. The authors propose a token-based selective replay scheme that reduces complexity of replay by moving the dependence tracking information out of the scheduler at the expense of marginal degradation in IPC. The authors look at ways for early termination of an incorrect speculation wave, and the techniques proposed in the paper primarily addresses scheduling speculation. The approach taken in this dissertation is more general, and can be used for any type of data speculation.

1.6 Dissertation Organization

This dissertation focuses on efficient recovery from mis-speculations in future processors. Chapter 2 describes speculation in modern processors,

along with its importance for high performance. The chapter also lists the various types of speculation and their growing importance. We discuss one particular type of data speculation, load-store dependence speculation, for the selective re-execution mechanism described in later chapters. The chapter then describes mis-speculation recovery, and explains the two mechanisms that are available for recovery.

Chapter 3 explains the features of a new class of instruction set architectures called Explicit Data Graph Execution (EDGE) architectures. The chapter also describes an initial implementation of an EDGE architecture, the TRIPS processor that we use to evaluate the performance of the proposed selective re-execution scheme. We use two different simulators in our study. The first simulator is a high-level simulator that approximately models the key architectural features processor. We use this simulator to perform an initial study of the DSRE mechanism. We then implemented DSRE on a more detailed simulator that accurately models the TRIPS prototype processor. This chapter describes these simulators, along with the benchmarks we used on the two simulators.

In Chapter 4, we describe the distributed selective re-execution mechanism and its implementation on EDGE architectures. We discuss the necessary enhancements in the architecture to support selective re-execution, and discusses the additional state needed to fulfill this requirement. The chapter also analyses the performance of the proposed mechanism for load-store dependence prediction. In Chapter 5, we describe two methods to improve the

performance of the base mechanism. These methods primarily involve ways to accelerate the commit wave. Even with the enhancements to the base DSRE mechanism, there is a significant difference in performance between the DSRE mechanism and an oracle policy that does perfect load-store prediction, and we discuss the reason for this performance gap in this chapter.

DSRE is designed as a recovery mechanism for any type of data speculation. In Chapter 6, we perform a brief evaluation of another data speculation mechanism, last-value prediction, to show that DSRE can concurrently support multiple speculation engines. The chapter also discusses how DSRE affects energy expended in the processor, and suggests ways for using DSRE to support recovery from soft errors to enhance reliability.

Chapter 7 discusses the implementation complexity of DSRE. We identify the challenges involved with implementing DSRE on a low-level simulator that models the TRIPS prototype processor. We propose and evaluate mechanisms for overcoming these challenges.

Finally, Chapter 8 talks about future directions for DSRE. The chapter discusses techniques for closing the performance difference still remaining between DSRE and an oracle policy, and as well as other potential speculation mechanisms that can benefit from DSRE.

Chapter 2

Efficient Speculation Recovery

Microprocessors have evolved from simple, non-pipelined, single issue machines to out-of-order, superscalar processors, capable of executing multiple instructions concurrently. Out-of-order execution of multiple instructions exploits the instruction level parallelism (ILP) in the program by allowing the processor to execute independent instructions concurrently. To exploit the ILP in a program and achieve high performance, processors use a technique called speculative execution. Speculative execution involves predicting values in hardware, and using the predicted values, to execute instructions further down in the instruction stream. There are many different types of speculation. Some of these include :

- Control speculation [18, 64]
- Scheduling speculation [24, 32]
- Load-store dependence speculation [12, 46, 73]
- Data-value speculation [40]
- Predicate prediction [13]

- Coherence speculation [9, 25]

Control speculation, also known as branch prediction, is a well known speculation technique for predicting the outcome of control instructions during instruction fetch [18, 64]. The processor uses the predicted outcome to fetch and execute instructions speculatively, and validates the speculation when it executes the corresponding control instruction. Over the years, researchers have proposed and implemented a number of different branch predictors in processors to improve branch prediction accuracy [1, 26, 27, 35, 36, 41, 43, 64, 71, 72]. Recovery from control mis-speculation involves discarding instructions that were fetched down the wrong path, and restarting execution with instructions from the right path.

Modern processors with deep pipelines and large issue widths are capable of managing a large number of instructions in flight. To keep the instruction window full, a number of other speculation mechanisms have been implemented in processors. These mechanisms include set and way prediction for caches [32], scheduling speculation for loads [24, 32], and load-store dependence speculation [32]. Set and way prediction involves predicting the next set and way in a set-associative instruction cache for fetching. In scheduling speculation, the consumers of load instructions are scheduled assuming a load hit, while load-store speculation tries to predict if loads conflict with earlier in-flight stores. We discuss load-store dependence speculation in detail in the next section, as it is the type of speculation that we use to evaluate distributed selective re-execution.

2.1 Load-store Dependence Speculation

In this dissertation, we evaluate the use of DSRE to reduce the misspeculation penalty for load-store dependence speculation. Issuing loads out of order with respect to stores is necessary for high ILP in current and future machines. Current machines use load-store dependence prediction to facilitate early issue of loads. However, effective load speculation is growing more difficult for several reasons. First, larger instruction windows mean that more conflicting load/store pairs will exist in the window, putting more pressure on the dependence predictors. Second, the cost of flushing the pipeline upon a misprediction is increasing as the in-flight state increases and control becomes more distributed. Third, the performance losses due to dependence mispredictions become more of a bottleneck as ILP elsewhere is increased. Fourth, since wire delays will force partitioning in future architectures, dependence predictors are likely to be distributed along with cache banks, reducing their accuracy.

2.1.1 Maintaining Sequential Memory Semantics

In out-of-order processors, sequential memory semantics must still be maintained. Program-earlier stores must forward their values to later loads for correct execution. The *conservative* policy for addressing this issue is to prevent a load from issuing until all earlier stores with unresolved addresses have issued. The ideal policy is an *oracle*, in which loads that do not conflict with earlier stores access the cache to retrieve data but wait for the latest

conflicting store in program order before the load if a conflict exists.

Researchers have investigated compiler-assisted approaches for efficient memory disambiguation. Gallagher et al. [20] proposed the memory conflict buffer for memory disambiguation. In their approach, the compiler aggressively hoists load instructions above store instructions and inserts correction code to provide recovery when there is an address conflict. The memory conflict buffer detects these conflicts. This scheme relies on a centralized issue queue for initiating recovery, and is thus unsuitable for distributed architectures. Cheng et al. [11] investigate early load address computation using compiler support. However, their approach requires changes to the ISA so that the microarchitectures can differentiate between the various types of loads in the system.

Microarchitects have tried to approach oracle performance by providing dependence predictors, which allow some loads to issue in the presence of earlier unresolved stores, speculating that they will not be dependent, and flushing the pipeline if incorrect. Loads incorrectly predicted to be dependent on an earlier in-flight store do not cause a pipeline flush—they merely lose an opportunity for higher performance by issuing late, after previous stores are resolved even though they could safely issue earlier.

A few examples of simple dependence predictors include those proposed by Moshovos et al. [46], which used the load's program counter to index into a table of saturating counters that specified whether or not a load should issue speculatively. Store-wait tables, as implemented in the Alpha 21264 [32], is

another PC-based dependence prediction mechanism, where a load predicted conflicting waits for all previous stores to resolve. Store sets [12] are a more complex proposal that attempts to match up loads with specific stores, so that potentially dependent loads do not have to wait for *all* previous stores to resolve, just the ones likely to conflict. Finally, Yoaz et al. [73] propose a predictor that uses distance estimations to approximate store set capabilities with reduced complexity. The proposed dependence predictors need to be modified to work in a distributed environment. In this dissertation, we simulate three types of dependence predictors that work in a distributed environment, *all-stores*, *one-store*, and *first-store*.

All-stores: This strategy is similar to the predictor organization of Moshovos et al. [46]. This predictor uses the PC to index a 1-bit table that is set on a conflict, and only predicts no conflict for a load when the counter value is not set. When *all-stores* predicts a conflict, a load waits until all prior stores complete before issuing safely. The table is cleared unconditionally every 10,000 blocks. PC-indexing outperformed the other indexing functions we measured, including address and PC-address hybrid indices, as well as the less aggressive store-wait tables of the Alpha 21264. This predictor matches the predictor in the TRIPS prototype processor implementation [3].

One-store: The second type of predictor we simulate is a modified variant of store sets [12]. Modifications were necessary because the distributed architecture that we simulate cannot enforce issue-order among stores due to the distributed fetch mechanism. Stores in this model fire in dataflow fashion

when they receive their address and data. Thus, we modified the predictor, which we call *one-store*, to force a load to wait for exactly one store, rather than a set of stores as in the original proposal. The next paragraph explains the differences between the *one-store* scheme and store sets.

The *one-store* predictor uses a PC-indexed Store Set Identifier Table (SSIT) to maintain a common tag for each load and store pair. These tags are called Store Set Identifiers (SSID). The predictor uses a Last Fetched Store Table (LFST) to store the SSIDs. The implementation is described in detail in the original paper [12]. Initially, these tables are empty and all loads are predicted as non-conflicting. When the processor detects a load-store ordering violation, it allocates an SSID index to the violating load-store pair, and also creates an entry in the SSIT for the load and the store containing this SSID. During block dispatch, all stores in the dispatched block access the SSIT table to check for a valid LFST entry. If a store finds a valid entry, it inserts its instruction identifier in the corresponding entry in the LFST table. Loads in a block also index the SSIT table during dispatch. When a load finds a valid SSIT entry, it checks the LFST table for a valid store entry. If a load finds a valid store in the SSID table, it marks itself as being dependent on the store. When a load resolves and reaches the memory interface, it checks to see if it has been marked as being dependent on a store. If the load is marked dependent, it sends data back to its consumers only after the pertinent store arrives at the memory interface. In our experiments, we used a 4K entry SSIT table and a 128 entry SSID table. The SSIT table was indexed using the last

12 bits of the PC and was unconditionally cleared every million cycles.

The main difference between the *one-store* predictor and the store sets scheme is that in the *one-store* predictor a load is marked as being dependent on only one store. The store sets scheme is able to track loads that are dependent on multiple stores by examining the LFST table during store issue, and replacing the store in the table with a matching store that is later in program-order. Due to the decentralized nature of store issue in the TRIPS processor, the *one-store* predictor is unable to track multiple stores that conflict with a load. This *one-store* predictor was used in the Trimaran/TRIPS-based high-level GPA simulator.

First-store: The *first-store* predictor is an alternative dependence predictor for the TRIPS prototype processor. The predictor is not implemented in the TRIPS ASIC prototype processor, but might be included in subsequent designs. The *first-store* predictor uses a table of 2-bit up-down saturating counters. The training of the predictor is entirely distributed and happens at the memory interface. The processor indexes the table using the load's PC. The counter corresponding to a load is incremented on a load violation and decremented when a load executes without a conflict. The predictor is discussed in more detail in the next chapter. The predictor can predict a load to be non-conflicting, conflicting-one-store, or conflicting-all-store. If the predictor predicts a load as non-conflicting, the load can send its reply without waiting for prior stores to resolve. If a load is predicted conflicting-all-store, it waits for all prior stores to resolve before sending a reply. If a load is predicted

conflicting-one-store, it sends its reply when the first matching store arrives at the memory interface or when all prior stores resolve, whichever happens earlier.

We used a Trimaran/TRIPS-based simulation environment for initially evaluating DSRE performance. This simulation environment modeled an EDGE architecture at a higher level of detail for initial evaluation of the architecture. We subsequently validated the protocol on a simulator that models the low-level details of a processor implementation. The simulated processor is described in greater detail in the next chapter. Figure 2.1 shows the performance of the simulated GPA processor using conservative load/store issue, *all-stores*, *one-store*, and oracular prediction. These experiments assumed the TRIPS prototype configuration of 64 frames, which corresponds to a 1K issue window across the 16 ALUs. The graph confirms prior results that the conservative policy performs poorly with respect to the oracle policy, which is 2.37 times faster on average. The *all-stores* dependence predictor improves performance significantly over the conservative approach, but only by 46%, which is approximately only one third of the additional performance improvement obtained by the oracle policy. The more aggressive *one-store* predictor performs much better (66 %) but still achieves only a fraction of what is possible with *oracle*, due to the performance lost by flushing the pipeline on a mis-speculation.

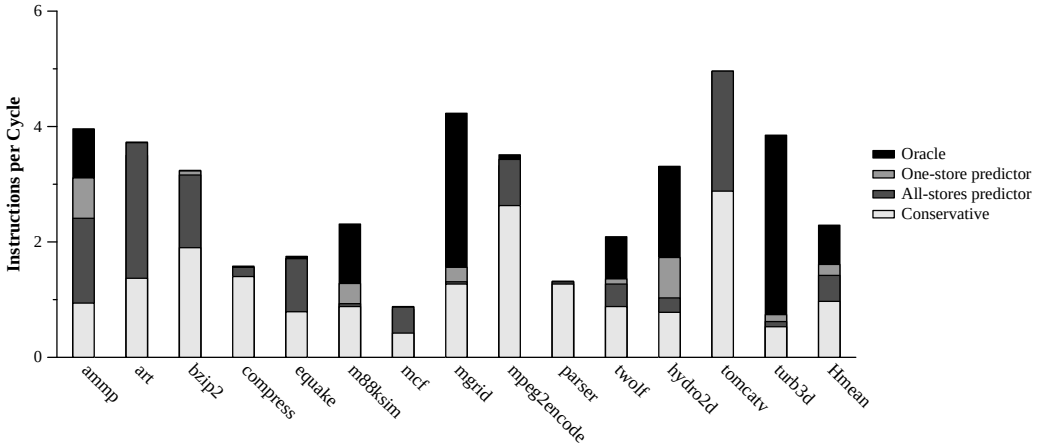

Figure 2.1: Performance effects of load/store ordering policies with a 1K window

2.1.2 Memory Speculation for Large Windows

As issue windows grow larger, from hundreds to thousands and eventually tens of thousands of instructions, the number of potential conflicts (stores followed by loads to the same address) grows. This growth threatens to limit the parallelism that can be exploited in future high-ILP machines.

Window size	**IPC**			
	conservative	**oracle**	**all-stores**	**one-store**
1K	1.06	3.14	1.63	2.03
2K	1.06	3.66	1.67	2.07
4K	1.05	3.82	1.68	2.05
8K	1.07	3.85	1.70	2.08

Table 2.1: Performance characterization with memory dependence prediction for a 4K store sets predictor

Window size	Potential conflict	Dependence predictor performance (% Accesses)			
		Predicted Dependent Executed Dependent (PD:ED)	Predicted Dependent Executed Independent (PD:EI)	Predicted Independent Executed Independent (PI:EI)	Predicted Independent Executed Dependent (PI:ED)
1K	12.17	17.04	12.17	71.57	1.03
2K	14.58	19.19	13.64	68.26	0.92
4K	17.10	21.04	15.02	65.35	0.76
8K	19.37	21.61	15.85	64.21	0.55

Table 2.2: Conflict breakdown with increasing window size

Table 2.1 lists the load/store conflict behavior for four different window sizes (1K-8K instructions). These experiments assume a perfect branch predictor, so the window is always filled with useful work unless the processor pipeline is being flushed from a load/store mis-speculation. The three IPC columns show the average instruction throughput for three of the four ordering schemes from Figure 2.1. From Table 2.1, we see that performance saturates with a conservative load issue policy when the window size is increased. However, with an oracle policy, performance continues to increase as the window size is increased until 4K instructions. Performance with the *one-store* predictor increases only marginally, because of the increasing number of load-store conflicts with a larger window size.

Table 2.2 shows a breakdown of the dependence predictor performance with increasing window size. The column labeled *Potential Conflict* shows the fraction of loads in the instruction window that reference the same address as a store that is also in the window, when using oracle load/store dependence prediction. Note that a conflict will actually occur only if the load issues out of order from the store. Not surprisingly, the fraction of potentially conflicting loads increases with window size, and combined with a larger number of loads in the window, results in a much larger total possibly conflicting loads. The remaining columns show the behavior of the *one-store* predictor. The first column, labeled Predicted Dependent Executed Dependent (PD:ED), shows the percentage of loads that are predicted dependent and actually end up being dependent during the execution. The second column, labeled Predicted Dependent Executed Independent (PD:EI), shows the percentage of loads that are predicted dependent but actually end up being independent. The third column, labeled Predicted Independent Executed Independent (PI:EI), shows the percentage of loads that are predicted independent correctly while the last column, labeled Predicted Independent Executed Dependent (PI:ED), shows loads that are predicted independent but end up conflicting with a prior store during execution.

For large instruction windows (8K), on average 15% of the predicted accesses are predicted as dependent (PD) and actually end up independent at execution time (EI), thus increasing the load latency for these accesses. Fewer than 0.6% of the accesses are predicted independent (PI) and are actually

dependent (ED), requiring a rollback recovery. The remaining 85% of the loads have their dependence predicted correctly and incur no penalty. As the window size is increased, we have a greater fraction of loads that conflict with earlier unresolved stores, resulting in the predictor becoming more conservative in its prediction. From column 4 in Table 2.2, we can see that the predictor becomes increasingly conservative as window size increases, and a greater percentage of the loads are incorrectly predicted to conflict, unnecessarily forcing them to wait. This class of loads stands to benefit greatly from DSRE.

With the larger in-flight state reducing the accuracy of predictors, efficient mis-speculation recovery will become increasingly important in future processors. The number of potential conflicts can be somewhat reduced with more aggressive compiler optimizations. For example, the compiler can register allocate load-store pairs that are likely to conflict. The compiler can also increase the distance, in terms of instructions, between a load and store, to prevent them from being in the instruction window at the same time. Even with aggressive optimizations, a compiler cannot completely eliminate load-store conflicts as these dependences are not always known at compile time. Hence, the number of potential conflicts is likely to increase with larger instruction windows.

2.2 Mis-speculation Recovery

Mis-speculation recovery involves discarding the incorrectly computed values resulting from the mis-speculation and computing the correct values.

The traditional method to deal with mis-speculation recovery, called a pipeline flush, involves purging the pipeline of all instructions after the mis-speculating instruction and re-executing them with correct values. This approach is inefficient as it re-executes instructions not control or data dependent on the mis-speculating instruction. A more efficient approach to mis-speculation recovery, called selective re-execution, re-executes only those instructions that executed incorrectly the first time. However, this approach involves tracking data dependences between in-flight instructions, and is complex to implement in current processors. We discuss both these approaches, along with examples, in the next section.

2.2.1 Pipeline Flush

The traditional method to recovering from mis-speculation involves flushing the pipeline of all instructions after the mis-speculating instruction in program order, and re-executing them. This approach has a high performance penalty as instructions that are independent of the mis-speculating instructions also get re-executed. To illustrate the pipeline flush solution, Figure 2.2 shows part of the data flow graph (DFG) from the SPEC CPU2000 benchmark, bzip2. The program order in the DFG is represented by traversing the graph top to bottom and left to right.

In one particular execution of this graph, let us assume that the load instruction *lws*—shown in gray in Figure 2.3—mis-speculates, and gets incorrect value from the cache instead of an earlier matching store. A pipeline flush

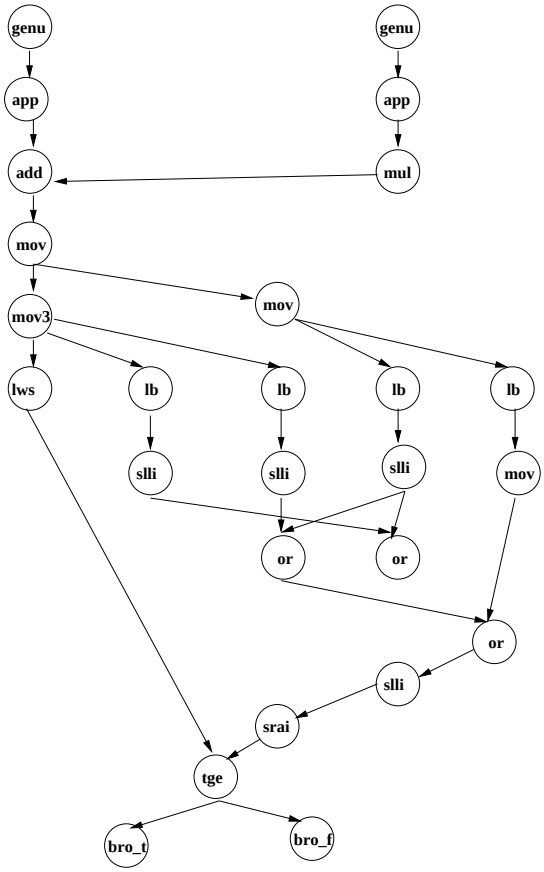

Figure 2.2: Part of the DFG from bzip2

involves flushing the *lws* instruction, as well as all instructions after the *lws* in program order, and re-executing them after computing the correct value of *lws*. This graph is shown in Figure 2.4, where the gray instructions are re-executed. This solution is inefficient as instructions that are not part of the DFG of the *lws* instruction are re-executed. The advantage of pipeline flushing is its simplicity—the processor flushes all instructions after the misspeculating instruction and re-executes them, making recovery simpler to im-

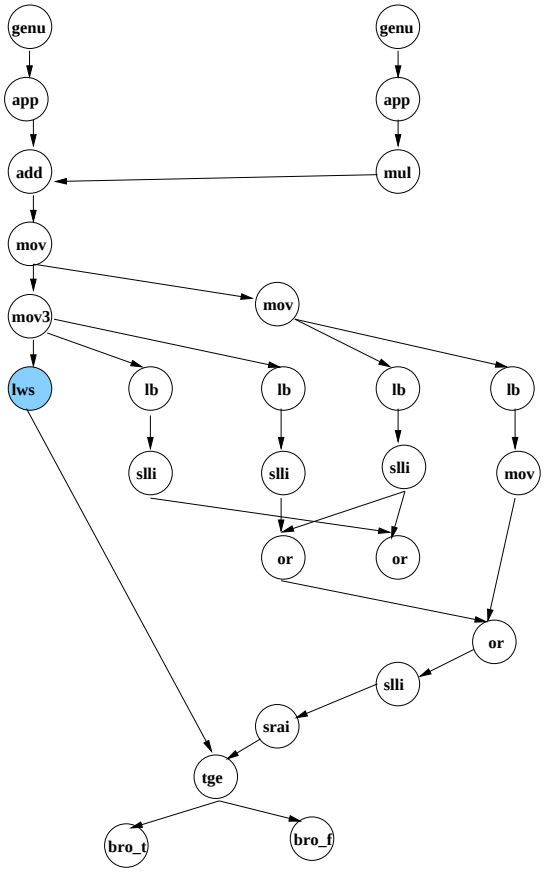

Figure 2.3: Load instruction *lws* mis-speculates

plement. However, flushing the pipeline is becoming increasingly expensive, both in terms of performance and energy, for processors that have a large number of instructions in flight because of the cost to refill the pipeline.

2.2.2 Selective Re-execution

Figure 2.5 shows another method to recover from mis-speculations, selective re-execution. In this method, only instructions that received incorrect

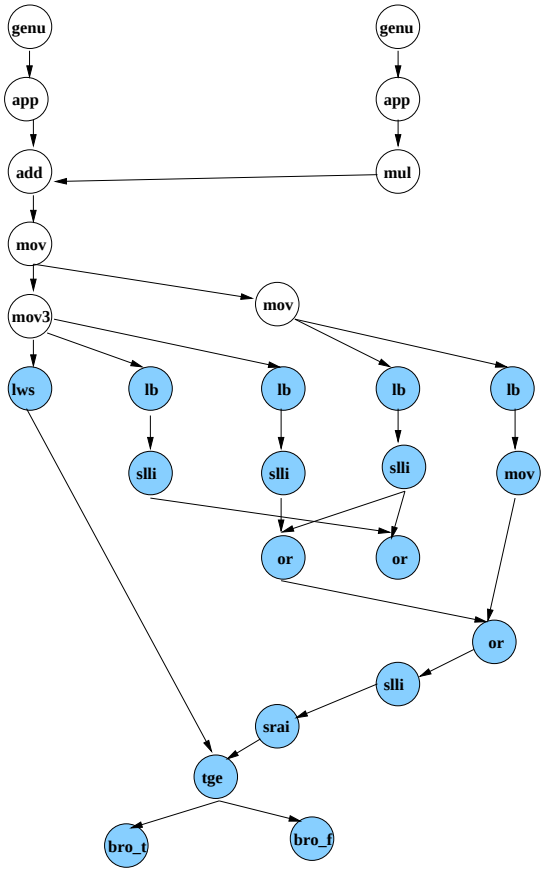

Figure 2.4: Traditional flush

---

values from the mis-speculating instruction are re-executed. Thus, selective re-execution prevents unnecessary re-execution of independent instructions on a mis-speculation. However, selective re-execution in current superscalar processors is implemented in only a limited fashion for two reasons:

1. The dynamic tracking of data dependences between all instructions that are in-flight in a conventional ISA involves an enormous amount of

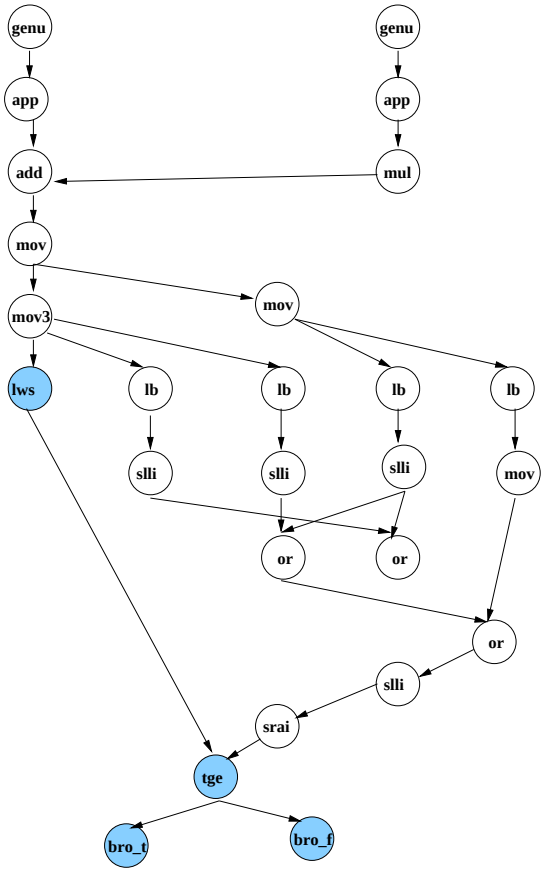

Figure 2.5: Selective re-execution

---

state [33]

2. Re-execution of a subset of instructions that have executed already results in scheduler complexity [15]

In this dissertation, we present a selective re-execution protocol that overcomes the drawbacks of traditional approaches, thus enabling aggressive data speculation. Using the features of Explicit Data Graph Execution

(EDGE) architectures and an example implementation, the TRIPS processor, we implement a distributed selective re-execution protocol that uses simple local state, thus scaling to future communication dominated architectures. The protocol permits multiple data-speculative values for an operand to exist concurrently in the processor. We propose mechanisms to determine when an operand becomes non-data speculative, and is safe to commit to the architectural state. We evaluate the performance of this basic selective re-execution protocol, and identify the bottlenecks to performance. We then suggest two mechanisms for improving the performance with selective re-execution. We evaluate the protocol on the high-level GPA simulator, and validate it using the TRIPS prototype simulator, thus exposing some of the constraints encountered with a real implementation.

Chapter 3

Methodology

In this chapter, we describe Explicit Data Graph Execution (EDGE) instruction set architectures and an EDGE implementation, the TRIPS processor that we use to study the performance benefits of DSRE. The EDGE architecture and the TRIPS processor is a result of the collaborative effort of a number of researchers at UT Austin. This chapter describes only those features of the architecture that are exploited by DSRE for correctness and performance. The architecture is described in more detail elsewhere [3, 30, 48, 49, 53, 60]. We also describe the benchmarks that we used in our study. It is important to note that even though the DSRE mechanism takes advantage of some features of the TRIPS implementation, the basic DSRE mechanism relies on fundamental characteristics of an EDGE ISA that can be applied to any implementation.

3.1 EDGE Architectures

Explicit Data Graph Architecture (EDGE) instruction set architectures are a new class of architectures designed for processors in future, communication dominated technologies. The two main characteristics of an EDGE ISA are:

1. *Block-atomic execution*, in which the compiler compiles a program into blocks of instructions. These blocks are fetched and committed atomically by the processor during execution. In this model, a block must be committed in its entirety and a fraction of a block may not be committed.

2. *Direct instruction communication* between instructions within a block, where the hardware delivers an instruction's output directly to its consumers. An EDGE ISA specifies the consumers of an instruction in the producing instruction. Hence, there is no need for a shared namespace like the register file to communicate values between instructions within a block.

Instructions in an EDGE ISA execute in dataflow order, with each instruction firing when it receives all its input operands, and forwarding its output to consumers. Thus, in an EDGE ISA, a producer with multiple consumers would specify these consumers explicitly in the ISA using a software fan-out tree. In a conventional ISA, instructions write their outputs to the register file that is subsequently read by multiple consumers.

Figure 3.1 shows an example of how instructions are specified in one EDGE ISA implementation, the TRIPS processor. On the left side, we have shown a conventional reduced instruction set computer (RISC) code snippet, and the on the right side, we have shown the corresponding TRIPS code. Unlike RISC instructions that specify both inputs and outputs using a shared namespace (register file), TRIPS instructions do not specify inputs within a

RISC Code	EDGE Code
`genu R1, 16 ; R1 = 16`	`#1 genu 16, #[3,0]`
`genu R2, 3 ; R2 = 3`	`#2 genu 3, #[3.1]`
`Add R3, R1, R2; R3 = R1 + R2`	`#3 add #[4,0]`
`muli R4, R3, 2; R4 = R3*2`	`#4 muli 2, #[5,0]`
`slli R1, R4, 3; R1 = R4 << 3`	`#5 slli 3, #[6,0]`

Figure 3.1: RISC code and corresponding EDGE code

block, and specify outputs as target instructions that are explicitly encoded in the producing instruction. The explicit specification of targets solves the problem of global broadcast required to propagate values in conventional RISC architectures. For example, if we take the *add* instruction in Figure 3.1, the RISC instruction specifies architectural register R1 and R2 as inputs to the instruction, and register R3 as the output of the instruction. The corresponding TRIPS instruction specifies no inputs, and specifies instruction #4 as the target of the *add* instruction. The *add* instruction in this example will fire when it receives its two inputs from instructions #1 and #2, and will send its result to instruction #4.

The explicit representation of the DFG in an EDGE ISA facilitates selective re-execution by obviating the need to dynamically track data dependences between in-flight instructions, thus overcoming one of the drawbacks to implementing selective re-execution in modern RISC processors. Selective

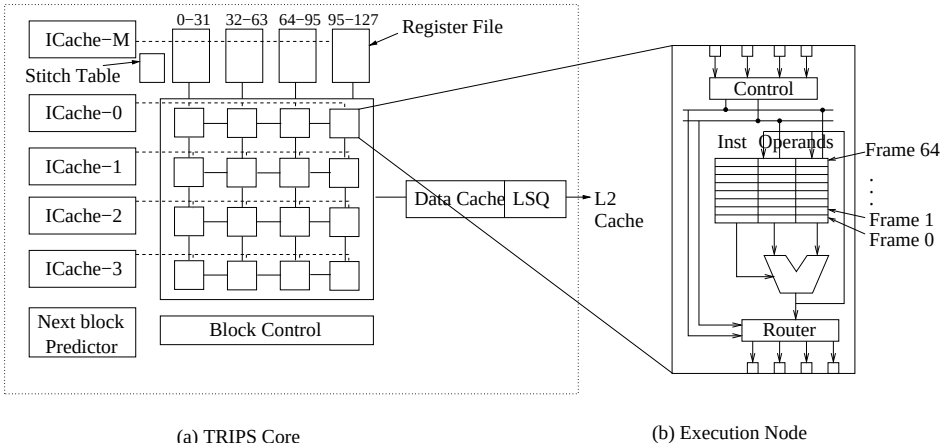

Figure 3.2: Simulated 4x4 grid processor

re-execution in an EDGE ISA can be implemented by sending correct operand values to the consumers of a mis-speculating instruction, that can in turn forward new results to their consumers. The next two sections describe two implementations of EDGE architectures that we used for evaluating selective re-execution.

3.2 Grid Processor

The grid processor was an early simulator implementation of an EDGE ISA to evaluate the feasibility of the architecture. The Grid Processor Architecture (GPA) simulator used the Trimaran infrastructure to simulate an EDGE ISA. The Trimaran infrastructure consists of a compiler, which generates machine instructions for the Trimaran ISA from C code. The GPA simulator maps these instructions to an explicit data graph ISA and executes

them. This mapping is an approximation of an actual implementation that helped with initial performance evaluation of this class of architectures.

In the grid implementation of an EDGE ISA, the compiler compiler conventional code written in C, C++, and FORTRAN into blocks of instructions called *hyperblocks*. During execution, the processor fetches, executes, and commits instructions belonging to a hyperblock atomically. Consequently, flushes in this model occur at a block granularity, with entire blocks being flushed on a trap or mis-speculation.

The microarchitecture preserves sequential semantics at the block level. Thus each block behaves like a "megainstruction" that is executed one after the other in the order specified by the program. Inside the blocks, instructions execute using a fine-grained dataflow model specified by an EDGE ISA. The grid processor supports conventional programming semantics by preserving the memory order among instructions within the block, and across blocks using a load-store queue that forwards earlier program-order store values to later loads.

The simulated grid processor consists of a grid of ALUs as shown in Figure 3.2. The ALUs have reservation stations for holding instructions and their operands. The compiler statically maps the instructions in a hyperblock onto the reservation stations in the ALUs. The instructions, however, fire dynamically when they receive their input operands. The processor concurrently fetches and maps the instructions for each row in a block.

The point-to-point communication among instructions within a block eliminates the need for a fully shared structure like a register file. However, we do need a register file for communication across blocks. The only shared structure that is still required is the load-store queue, to preserve correct ordering among loads and stores. There are different ways to implement the load-store queue and the data cache. We can have a centralized design that has a large area and timing overhead. We can have a physically partitioned, logically centralized design that results in design simplicity. However, this approach has a large area overhead due to replication, and also has similar timing issues to the centralized approach because of the large structures. We can have a logically and physically partitioned design that is both area and timing efficient. However, this approach significantly increases the design complexity, since we need to deal with overflow in the load-store queues. We used a centralized load-store queue and data cache in the GPA simulator. The TRIPS prototype processor replicates the load-store queue, resulting in a logically centralized, physically distributed design.

The grid processor has a lightweight network connecting the set of execution nodes. There are 4 register file banks on the top, along with an instruction cache bank for holding "read" and "write" instructions. The instruction cache banks are situated on one side and the centralized data cache is located on the other side. The global control logic, responsible for issuing fetch and commit commands, is situated at the bottom. Execution in the grid processor happens at a block granularity, with the processor fetching, mapping, and

committing blocks atomically [49].

Since the GPA simulator was an early high-level simulator implementation of an EDGE ISA, we made the following assumptions to aid its implementation:

1. The number of instructions in each hyperblock is not fixed, and there is no upper limit on the number of instructions per block.
2. There is no limit on the number of input and output registers per block.
3. The data cache and load-store queue is centralized.
4. There is no limit on the size of the load-store queue.

Having these approximations kept the simulator simple and flexible, and helped us do the scalability study reported in Chapter 5. Also, not having strict limits on the composition of blocks helped us achieve high performance on the GPA simulator without aggressive compiler optimizations. The TRIPS simulator performance is more sensitive to compiler optimizations because of the implementation constraints imposed on the architecture. Despite these differences, the GPA simulator is a good representation of an example EDGE architecture for initial performance evaluation.

3.3 TRIPS Processor

The TRIPS processor is a prototype hardware implementation of an EDGE architecture [3]. The TRIPS processor consists of 16 ALUs in a 4x4

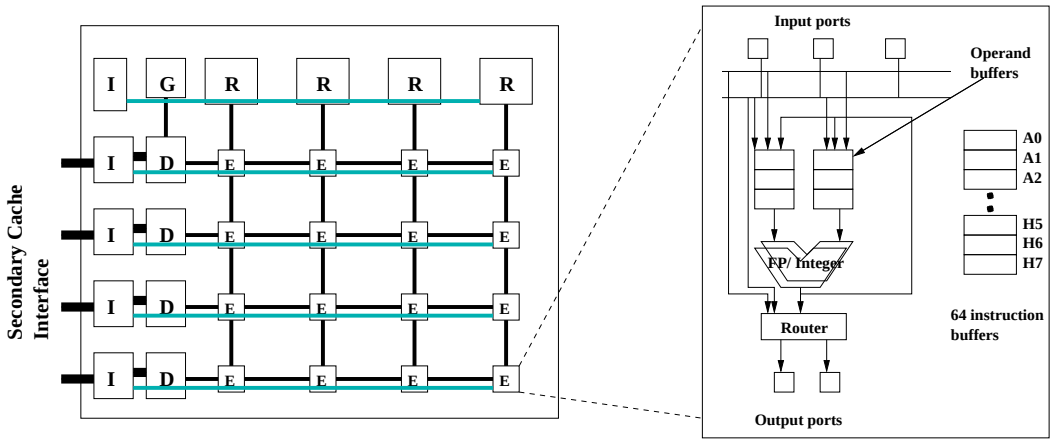

Figure 3.3: 4x4 TRIPS prototype processor

grid that are connected by a routed operand network (OPN) as shown in Figure 3.3. The processor core consists of five major types of units or tiles called global control tile (GT), instruction tile (IT), execution tile (ET), register tile RT), and the data tile (DT). The processor has a number of special networks connecting the different tiles.

3.3.1 TRIPS Microarchitecture

Figure 3.4 shows the various steps in the execution of a TRIPS block. The global tile initiates the fetching and mapping of instructions on the processor. After a block is fetched and mapped, instructions within the block execute in dataflow fashion, firing and forwarding values to their consumers. The placement of instructions on the processor is statically assigned by the compiler/scheduler, but they are issued dynamically by the issue logic at each node—the processor uses a Static Placement, Dynamically Issue (SPDI)

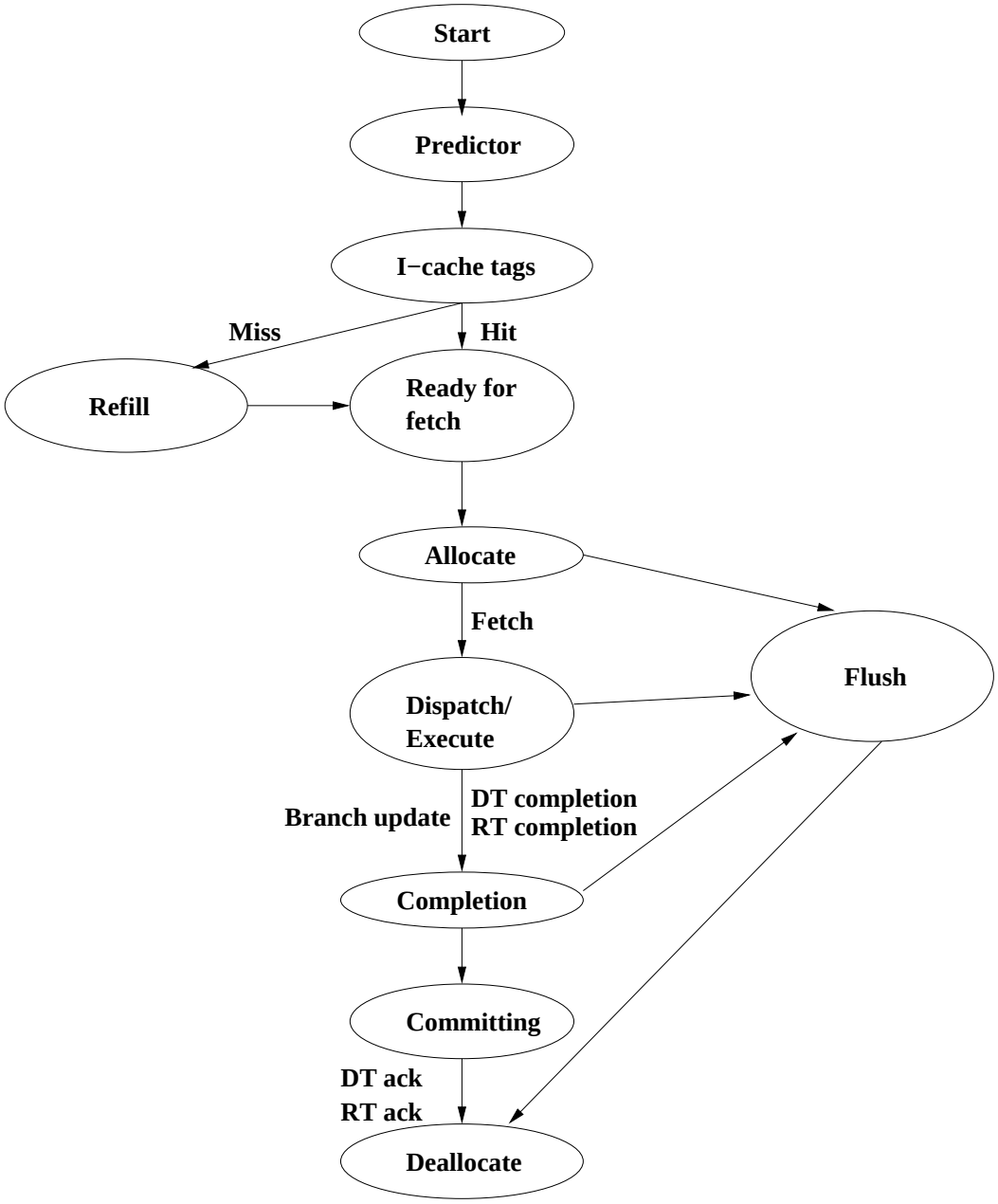

Figure 3.4: Life cycle of a TRIPS block

model [48].

The execution tile consists of the ALUs (integer and floating point), reservation stations for holding instructions and their operands, and logic for issuing instructions. The ET has a 4-stage pipeline consisting of the select stage, the read stage, the execute stage, and the writeback stage. The TRIPS processor uses a 2-packet operand network for sending operands between two tiles.

The register tiles handle reads and writes to the architectural register file. The RT has two pipelines—the read pipeline and the write pipeline—for sending register values to a block. The read pipeline in the RT is responsible for sending the register inputs from the architectural register file to the consuming instructions within a block. The write pipeline in the RT handles forwarding of register values from an older block to a newer block.

The data tiles handle the loads and stores within a block. The load-store queue (LSQ) in the data tile tracks the loads and stores that are in flight in the processor. The LSQ in the TRIPS processor has a physically distributed, logically centralized design. The DT has a main load-store pipeline for sending load replies and handling store arrivals. Incoming loads check for matching prior stores in the LSQ, and retrieve their data from the LSQ on a match. The loads get their data from the cache if they do not find a matching prior store. The DT has a dependence predictor for dynamically tracking load-store dependences. The DT also has a reissue pipeline for reissuing loads that are deferred by the dependence predictor.

Once the register tiles receive all the register outputs for a block, they send a completion signal to the GT on the Global Status Network (GSN). Similarly, the DTs send a completion signal to the GT when all the stores for a block have arrived at the data tiles. The GT also receives exactly one branch update message on the OPN from the ET that points to the next block to fetch. Once the GT receives all the three completion signals for a block, it sends a commit message on the Global Commit Network (GCN) to all the tiles. The commit message results in the ETs invalidating all the state corresponding to the committed block, the RTs writing the block outputs to the architectural state, and the DTs committing the stores in the block to the memory system. The RTs and DTs send a commit acknowledgment signal to the GT on the GSN after committing the register writes and stores to the architectural state. Upon receiving this signal, the GT can now free the resources allocated to this block and reassign it to another block.

To achieve high instruction level parallelism, the TRIPS processor can have up to 8 blocks in execution concurrently. The processor uses a variation of a 2-level predictor to predict the next block to fetch [53]. On a mis-speculation, the global tile sends a flush signal to all the other tiles. The TRIPS processor uses rolling flushes; blocks are flushed as soon as a mis-speculation is detected.

3.3.2 Dependence Prediction in the DT

The dependence predictor in the DT is used to predict if in-flight loads are independent of previous unresolved stores. Each DT has a 1024-entry 1-bit

dependence predictor that is indexed by exclusive or-ing (*xor-ing*) the top 5 bits of the block address, and the reversed load-store identifier (LSID) of the load. The LSID is a 5-bit identifier that the compiler assigns to each load or store in a block. The predictor is accessed by loads in the first stage of the load-store pipeline. If the bit corresponding to a load is not set, the predictor predicts no conflict and the load sends its reply immediately. However, if the bit in the predictor is set for a load, the DT defers the load. The reissue logic in the DT subsequently selects the load for execution after all prior stores have resolved.

When stores arrive at the DT, they check the LSQ to ensure that no newer load that matches the store address has replied incorrectly. If the store finds a mis-speculating load, it marks the block containing the load as violating and the bit corresponding to the load is set in the predictor. The next time the load executes, it will find this bit set and will send its reply only after all prior stores have resolved. The predictor is cleared unconditionally by the DT after the execution of 10,000 blocks.

The predictor described above is the one implemented in the TRIPS prototype. In this dissertation, we also evaluate an extension to the above scheme. This scheme, called the *first-store* scheme, uses an adaptive 2-bit predictor to track dependences between loads and stores. The predictor is again indexed by the block address and the load's LSID. The 2-bit value stored in the predictor identifies the load as non-conflicting, conflicting-one-store, or conflicting-all-store. The non-conflicting and conflicting-all-store states cor-

respond to the states of the 1-bit predictor. When the predictor predicts conflicting-one-store, the load reply is sent when the first matching store arrives at the DT.

To implement conflicting-one-store, the LSQ was enhanced with a *ready* bit for each load. The DT defers loads that are predicted conflicting-one-store by the predictor. When a later store arrives and its address matches with a load that has been predicted conflicting-one-store, it sets the ready bit for the load. The load reissue logic is modified to also issue deferred loads that have their ready bit set. If there is no store match, the deferred load is issued after all prior stores have resolved.

Since the predictor used for this scheme has a 2-bit up-down counter, clearing the entire predictor is not necessary. The counter is incremented when a load results in a violation. The counter is decremented when a load is held back unnecessarily. Identifying loads that were incorrectly predicted independent is easy, as it results in a pipeline flush. Loads that were incorrectly predicted dependent are difficult to identify. To identify this case, every load in the LSQ has a counter that stores the number of stores that forwarded values to this load. Stores increment this value for a load every time they forward a value. The reissue logic checks this counter for every load, after all prior stores have resolved. If the counter is zero and the prediction for this load was conflicting-one-store or conflicting-all-store, the load prediction is identified as an incorrect prediction to the dependence predictor by the load-store queue. If the counter is one and the prediction is conflicting-all-store,

again it is identified as an incorrect prediction. This information is used by the DT for training the predictor.

3.3.3 TRIPS Software Model

The TRIPS processor executes binaries generated by using the *tcc* compiler. The compiler takes code written in conventional programming languages, C and FORTRAN, and compiles it into blocks of machine instructions. These blocks, called hyperblocks, are single entry, multiple exit blocks. The high-level code is first compiled into an intermediate form called TRIPS Intermediate Language (TIL). TIL code resembles the assembly code for conventional processors and is in a human readable form. The scheduling phase of the compiler operates on the TIL code and converts it into TRIPS assembly (TASL) code. The linker links the TASL code into a TRIPS binary. For the TRIPS prototype processor, the compiler builds 128-instruction fixed size blocks.

The compiler assigns LSIDs to all the load and store instructions to establish the memory order within a block. The compiler also creates a 32-bit store mask that the DT uses to identify the stores in the block. The compiler needs to ensure that all the stores specified in the store mask reach the DT, for the DTs to signal completion.

3.3.4 TRIPS Cycle-accurate Simulator

The TRIPS simulator was developed by the TRIPS team at the University of Texas at Austin to model the TRIPS prototype processor. The purpose of the simulator was two-fold:

- To perform functional validation of the processor to ensure correctness of the design.
- To validate the performance of the processor, and explore techniques to improve performance.

The simulator models all the different tiles found in the prototype in great detail. The tiles are connected by the operand network that also matches the implementation. All the different pipelines in the various tiles of the prototype are faithfully modeled in the simulator. The simulator also models the various predictors found in the prototype, and has been matched with the processor RTL to within 5% accuracy on a wide array of microbenchmarks and random-tests. Thus, the simulator provides a good setting for the evaluation of DSRE in an actual processor implementation.

Due to the closeness of the simulator to the actual prototype processor, it is not as flexible as the GPA simulator used for initial evaluation of the DSRE mechanism. Also, because of its detailed modeling, the TRIPS simulator is considerably slower than the GPA simulator. However, the TRIPS simulator exposes some of the issues with the practical implementation of the mechanism,

and hence gives valuable insights into the difficulties encountered in an actual hardware implementation.

3.4 Benchmarks

We used a set of benchmarks from the SPEC CPU95, SPEC CPU2000, and the MediaBench suite with the GPA simulator. The modified Trimaran infrastructure we used could compile only a subset of benchmarks from this suite, and we used these in our initial study. The simulated benchmarks are listed in Table 3.1. For each benchmark, we fast-forwarded through the initialization phase and simulated 100 million instructions.

Benchmark	**Fast-forward count**
compress	6000000000
hydro2d	1000000000
tomcatv	1000000000
turb3d	1000000000
m88ksim	1000000000
ammp	3000000000
art	1000000000
equake	400000000
bzip2	1000000000
mcf	1000000000
mgrid	1000000000
parser	1000000000
twolf	1000000000
mpeg2encode	1000000000

Table 3.1: GPA simulator benchmarks

The higher instruction throughput of the GPA simulator allowed us to execute longer benchmarks. On a 1.7 GHz Intel machine, the GPA simulator can simulate 10,000 cycles/second on an average. The TRIPS simulator has a lower instruction throughput of around 600 cycles/second, because it models the low-level details of the processor. Hence, we were forced us to use smaller benchmarks with the TRIPS simulator.

The TRIPS prototype simulator was validated using a set of benchmarks that comprised a regression suite. This suite consists of a number of handwritten assembly programs designed to test all the functionality of the processor. The same suite was used initially to identify many of the bugs associated with the implementation of DSRE on the TRIPS processor.

After initial validation of the correctness of the DSRE mechanism, we used a set of benchmarks from the EEMBC suite for performance analysis. The self-checking nature of these benchmarks also helped us identify corner cases that showed up only after the execution of thousands of instructions. These programs were compiled with the *tcc* compiler. The EEMBC benchmarks are loop-based benchmarks that execute for a user-specified number of iterations. To skip the initialization phase, we found the number of blocks that need to be executed for each benchmark for a single iteration. We fast forwarded that many number of blocks for each benchmark to skip the initialization phase. The EEMBC benchmark main loops have widely varying instruction counts. Due to the slow simulation speed of the TRIPS prototype simulator, we fixed the number of iterations to simulate for each benchmark between

50 and 5000, to get reasonable simulation time. The number of blocks fast-forwarded, along with the number of committed instructions simulated, is listed for each benchmark in Table 3.2.

Benchmark	Blocks Fast-forwarded	Instructions committed
a2time01	14219	5495586
aifftr01	70038	64456170
aifirf01	14616	5209529
aiifft01	59718	59152208
autcor00	24519	1122004
basefp01	14138	827732
bezier01	19372	42770766
bitmnp01	22274	5998634
cacheb01	16991	1777907
canrdr01	14483	2131544
conven00	497864	3180085
fft00	270339	26850943
idctrn01	17282	2626483
iirflt01	14574	1520803
ospf	36289	11386322
pktflow	685311	12467818
pntrch01	15866	5691186
puwmod01	14337	3425108
routelookup	60449	26801639
rspeed01	14149	2393817
tblook01	14199	2534138
ttsprk01	14399	4466450
viterb00	53384	15315283

Table 3.2: EEMBC benchmarks fast-forward and simulation count for the TRIPS prototype simulator

---

In the next chapter, we describe and evaluate the proposed DSRE mech-

anism on the GPA and TRIPS simulator.

Chapter 4

Distributed Selective Re-Execution

In this chapter, we describe the selective re-execution mechanism and its initial evaluation on the high-level GPA simulator. We also compare these results to those obtained using the more detailed TRIPS prototype simulator. We first start by explaining how to implement selective re-execution on the EDGE based TRIPS architecture using a code snippet. We then describe the extra state necessary for correct execution with selective re-execution. Finally, we do a performance analysis of the proposed scheme using the GPA and the TRIPS prototype simulator.

4.1 DSRE for EDGE Architectures

In the TRIPS instantiation of an EDGE ISA, instructions and their operands are buffered as they arrive at the reservation stations. When an operand arrives, its tag indicates the reservation station and instruction operand to which it corresponds. When an instruction receives all its operands, it fires, executes, and sends the result to its consumers, which are specified using the target fields in the just-issued instruction.

The multiple hyperblocks in flight effectively form a large dataflow

graph (DFG). Within hyperblocks, the DFG is a statically constructed graph, with arcs going from ALU to ALU. Cross-block arcs are instantiated through register names; each block reads from and writes to a subset of the architectural registers. If a hyperblock produces an output allocated to R3, and the subsequent hyperblock requires an input read from R3, the processor will forward the value of R3 from the older to the younger block as soon as it is produced. Thus, the large DFG is a collection of smaller, statically produced DFGs stitched together by dynamically resolved cross-block arcs through the register file and inter-block and intra-block arcs through memory.

Figure 4.1 shows the C-code for a simple loop, along with the corresponding EDGE assembly code. We also show the data flow graph for 3 different iterations of the main loop body in the figure. The code snippet has two loop-carried dependences, one through memory and one through the register file (register 0). The loads in successive iterations of the loop depend upon the store in the previous iteration. The loop-carried dependence through memory is shown by dashed lines. As shown in the figure, this dependence is enforced by the load-store queue. If a load mis-speculates, the DFG sub-tree of the load gets incorrect values. These nodes are shown shaded in the figure. In a conventional implementation without selective re-execution, a mis-speculation will trigger a flush of all instructions after the violating load.

To initiate re-execution of an instruction that has computed with a wrong value, the correct value is sent to the incorrect instruction's node with the same tag as the original instruction. As shown in Figure 4.1, the instruction

C−Code

```
for (j=1; j < 10; j++)
   a[0] = a[0]+j;
```

TRIPS Assembly

```
loop_body:
  Read G[0] N[0, 0] ;   Read j
  Read G[1] N[2,0] N[4,0] ;  Read a[0] address
  N[0] addi 1 N[1,0] ;  j = j+1
  N[1] mov N[3,0] N[5,0]
  N[2] lw 0 N[3,1] ; Load a[0]
  N[3] add N[4,1] ; a[0] = a[0] + j
  N[4] sw 0 ; Store a[0]
  N[5] mov N[6,0] W[0] ; write j to register
  N[6] addi 1 N[7] ; Increment loop count
  N[7] teqi 10 N[8] ; Check for loop terminati
  N[8] mov N[9,P] N[10,P] ; Move predicte bits
  N[9] bro_f  loop_body ; if false, jump to lo
  N[10] bro_t exit ; Exit if true
```

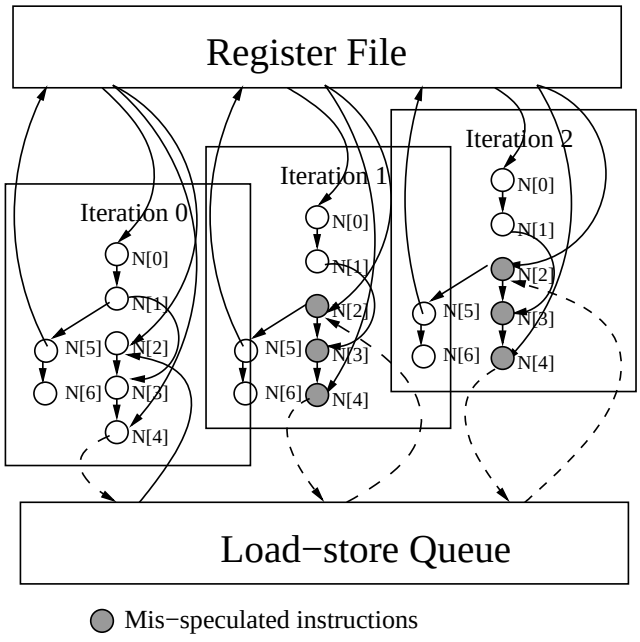

Figure 4.1: Re-execution on EDGE architectures

re-fires, sending a new output value to its dependent children. The children subsequently re-fire, and so on, eventually re-executing the entire DFG that is subtree data dependent on the faulting instruction. A re-fired instruction producing a value that crosses hyperblock boundaries sends a newer version of its result to the target hyperblock, which will cause additional instructions to re-fire. Note that this model permits selective re-execution without having to reissue any instructions not dependent on the erroneous instruction, nor do any instructions need to be re-fetched, re-dispatched, or moved. If the ALUs and the network can support the extra traffic, re-execution in an EDGE ISA is always beneficial, since re-executing an operation and everything dependent on it is no worse than having waited for the actual correct value rather than speculating.

When multiple speculative versions of an operand are allowed in the system, we need a mechanism to identify the non-speculative value of the operand. Selective re-execution can result in speculative sub-graphs for parts of the dataflow graph that the processor is executing. When a speculative operand finally resolves, it needs to communicate this resolution to all its children that are speculative. Thus, the problem becomes one of identifying when an operand is non-speculative, identifying the nodes in the speculative sub-graph of this operand, and communicating the speculation resolution to all the nodes in this sub-graph.

There are different ways to approach this problem. One solution is to have a centralized manager in hardware that keeps track of the dataflow

graph under execution. When a speculative operand resolves, it can check the manager to identify the nodes that depend on this operand, and mark them as non-speculative. However, this approach does not work in a distributed environment. Also, multiple independent speculative values might be traversing a dataflow sub-graph simultaneously, and it is difficult to isolate the value that becomes non-speculative.

Another solution to this problem is to have each node maintain its own dataflow sub-graph. The node can then identify all its children when it becomes non-speculative, and communicate the speculation resolution. The problem with this approach is the large amount of state that will be required at each node to maintain the dataflow sub-graph. Also, it is difficult to create the sub-graph when multiple speculative data values are allowed to overlap in the dataflow graph.

A third solution to this problem is to have a token that is propagated to the consumers of a speculating node, when the node becomes non-speculative. The children in turn propagate this token to their consumers when all their input operands become non-speculative. In this approach, we can imagine the set of tokens as a wave that follows execution and marks operands as non-speculative. This is the solution we explore in this dissertation.

4.1.1 Detecting Block Completion with Commit Waves

In the TRIPS processor, when each hyperblock completes, it is removed from the array and its instructions are all committed at once; hyperblocks log-

ically commit atomically. The processor writes the stores in a hyperblock back to the memory system in order during hyperblock commit. Since the TRIPS processor is a distributed microarchitecture, detecting completion of the oldest hyperblock is accomplished with point-to-point messages. The block header of each hyperblock contains a count of all hyperblock outputs (register writes, stores, and a single branch). When an output instruction in a hyperblock reaches the register banks or caches, the output counter for that hyperblock is decremented. When it reaches zero, all of the outputs of the hyperblock have fired, meaning that the block is safe to commit if it is the oldest hyperblock.

However, this scheme for detecting block completion cannot work without modification with selective re-execution. Since multiple waves of execution may be traversing the hyperblock's DFG simultaneously, the output instructions may receive their source operands multiple times, and thus do not know when it is safe to signal the completion logic that they have completed non-speculatively.

The solution we explore in this work is to add a *commit bit* to each valid bit at the instructions' operand buffers. When a commit bit is set, it signals that its operand is no longer speculative, and none of that operand's parents in the DFG may be speculative. The commit bit is only zero if there are still unresolved data speculations among its parents. Note that the control speculation (branch prediction) mechanisms are separate from these data speculation techniques. For a block to commit, it must be the control non-speculative block, and all of its register and store outputs must be non-speculative.

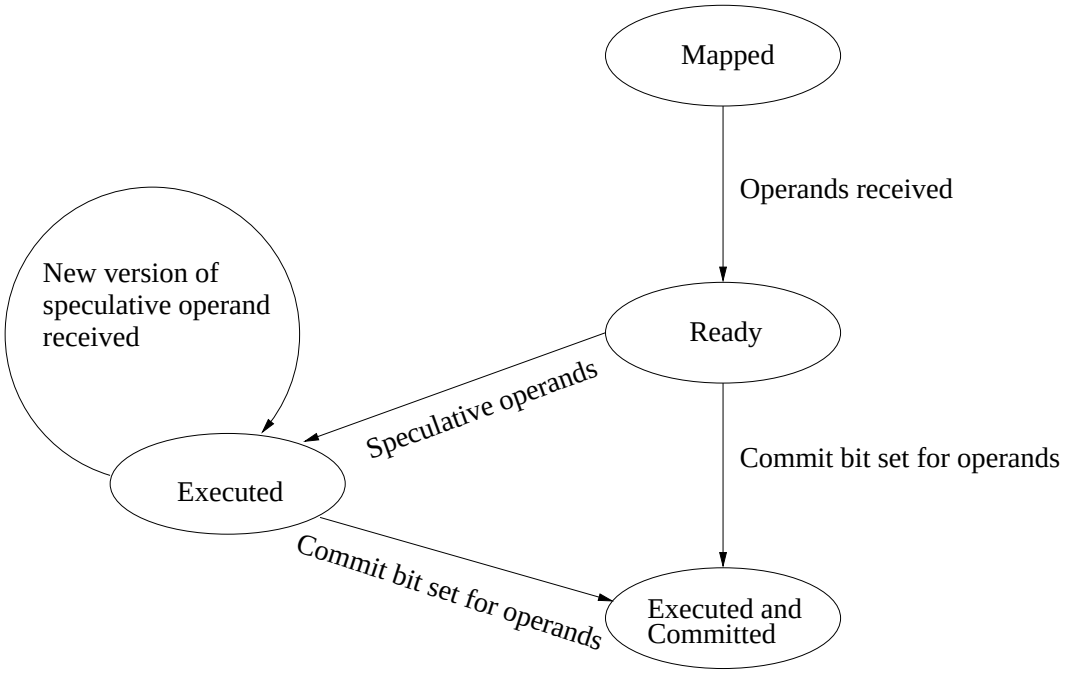

Figure 4.2: Instruction states with re-execution

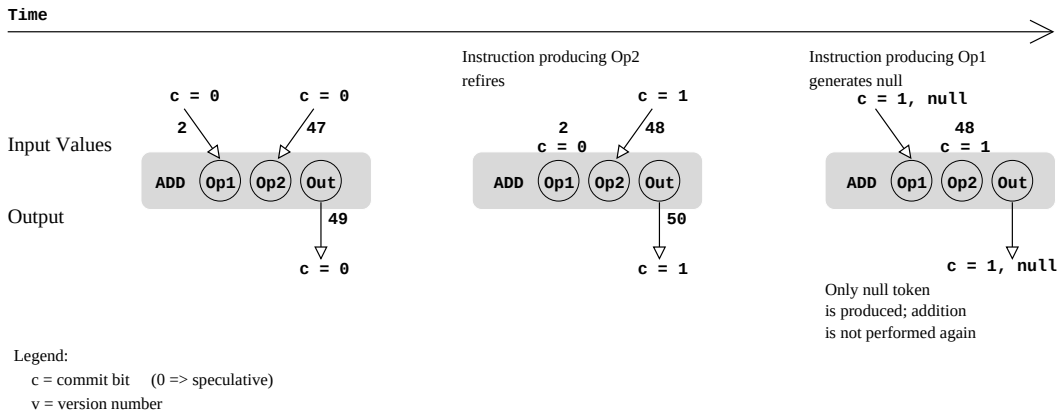

Figure 4.3: Illustration of commit messages

When all of an instruction's operands have received their commit bit, then the result computed using those operands is also non-data-speculative. Figure 4.2 shows the different instruction states with re-execution. When an instruction receives all its input operands, it goes into the *ready* state. The execution unit can now issue the instruction to a functional unit for execution. Once the instruction is issued to a functional unit, it executes and goes into the *executed* state. If any of the operands of the instruction is speculative, the instruction sends a speculative result to its consumers. The instruction will re-execute every time it receives a new version of any of its operands. The instruction remains in the *executed* state until it receives commit bits for all of its input operands. Once the instruction receives commit bits for all its inputs, it goes into the *committed* state and sends a commit bit to its consumers.

We simulate two types of commit bit messaging, shown in Figure 4.3. First, if an operand arrives at an ALU with its commit bit set, and the in-

struction's other operands are also non-speculative, then the instruction fires (or re-fires) and sends its result to its consumers with its commit bit set in the message control header. The second case occurs when an instruction has already fired—and has already sent its result with a zero commit bit—but is later determined to have been correct and becomes non-speculative. In this case, if the other operands of the instruction are non-speculative, a *null commit message* is sent to the consumers of that instruction, signaling that the operand previously sent is now non-speculative. However, if the the other operands of the instruction are still speculative, the operand is buffered and is marked as a null commit message operand. If the other speculative operands resolve correctly, a null commit message is sent after their resolution. If multiple parents target the same operand, only one parent is guaranteed to send the non-speculative value for the operand. Once a tile receives the non-speculative value for an operand, it ignores all other values for that operand.

An instruction may send a null commit message for several reasons. An arithmetic operation may compare its speculative, buffered operand with the receipt of a committed operand, and if they are the same, the result need not be recomputed, and the instruction can send a null commit message. More commonly, a load may have issued speculatively, in the presence of earlier unresolved stores (with a zero commit bit). When the load's address has received its commit bit, and *all* earlier stores have also received their commit bits—and if there was no address conflict with a store—then the load becomes non-speculative and a null commit message may be sent. A node can send

a null commit bit if the non-speculative output it produces matches the last speculative output that it produced.

A block is thus safe to commit when it is the oldest block (guaranteeing that there is no more control speculation) and when the block has received the commit bits of all its outputs. In terms of the DFG, the process of detecting completion can be thought of as a "commit wave" traversing the DFG behind the dataflow execution, and signaling completion when traversal of a hyperblock's portion of the DFG is complete.

4.1.2 Version Numbers: Out-of-Order Messaging

The scheme described thus far allows multiple, partially or fully overlapping waves of speculative execution traversing the DFG, succeeded by a "clean-up" commit wave. This model is simple so long as multiple speculative versions of an operand are always injected into the inter-ALU network in order *and* the network supports in-order delivery of messages. If either of those requirements are not met, then the possibility of overwriting the correct computation with later-arriving mis-speculative data arises. For example, assume that an instruction fires twice and produces versions A and B, where B is later determined to be correct, so is followed by a null commit message. If A and B are re-ordered (either by injection into the network or by the network itself), then the instruction's consumers will receive B, fire correctly, then receive A, fire incorrectly, and then receive the null commit message saving the incorrect result computed with A.

This case would occur if the network re-ordered messages, although the network we simulate does not exhibit this problem, because routing is deterministic and messages are never dropped. However, another window of vulnerability is opened by the possibility of injecting speculations in the wrong order. For example, assume a load with earlier unresolved stores accessed the cache but missed. Subsequently, a program-earlier store to the same address issued, so the store value is forwarded to the load and sent to the load's consumers (version B). The mis-speculated cache access eventually returns a value that could be forwarded to the load's consumers (version A) overwriting the correct computation triggered by version B. This case could be avoided by adding extra support to the memory system, but serves as an illustrative example.

We handle out-of-order messaging by augmenting transmitted operands with version numbers as well as commit bits. Each arc of the DFG can be traversed multiple times, with its version number increasing with each of its source operands. For example, if an ADD instruction fired three times, the version numbers of the operands sent to its consumers would be 0, 1, and 2, in order, regardless of what the version numbers of the ADD's source operands were. The highest version number is always the correct operand, and null commit messages are tagged with the version number that they are committing.

This scheme permits operands to be re-ordered and still function correctly, since version numbers are buffered with the operands and commit bits at the consuming instruction's reservation station. With version numbers, we

guarantee that the highest version number for an operand is always the correct value. If an ADD instruction has fired twice because it received two values of its left operand, which arrived with version numbers 0 and 2, and then later a version of the left operand arrives with version number 1, the last message is discarded because that operand has already received a higher version number, guaranteeing that the lower one is incorrect. If a null commit message arrived with a version number 3 for the left operand, the instruction would wait to receive the actual operand tagged with version number 3 before re-firing and propagating the result to the consumers with the commit bit set. This policy permits operands and commit bits (whether arriving with an operand or as null commit messages) to arrive in any order but still produce correct execution and guaranteed completion.

The simple version number scheme outlined above needs to be augmented if we allow multiple parents to target the same operand. In this case, we need to add an instruction identifier field to the version number to identify the particular parent of the operand. The highest version number, corresponding to a particular parent that generated the commit bit, is then guaranteed to be the right value.

We illustrate the role of version numbers with an example in Figure 4.4. The instruction shown in the example is a two input *add* instruction. The example shows one particular sequence of events related to the reception of operands for the instruction. Initially, the *add* instruction receives the first version of one of its operands. This operand is data speculative as its commit

INCOMING MESSAGE	STATE O1/V/C	O2/V/C	ACTION
01=2, v=0, c=0	2/0/0	−	No action − only one operand has arrived.
02=42,v=0, c=0	2/0/0	42/0/0	Add inputs and send output message (out=44; v = 0, c= 0). The output is speculative because inputs are speculative.
02=49, v=3, c=0	2/0/0	49/3/0	The execution unit re−executes the add instruction and a new result is sent out with a new version number. The results are still speculative (out=51, v =1, c=0).
01=null, v=0, c=1	2/0/1	49/3/0	The execution unit marks operand 1 as non−speculative. New output is not generated because the operand values have not changed.
02=null, v=4, c=1	2/0/1	49/3/1	Generate null token; both inputs are non−speculative
02=45, v=2, c=0	−	−	The execution unit drops the message because its version number is lower than the last version number received for this operand.

Table Legend:
STATE: Output value/Version number (V)/Commit bit (C)

Figure 4.4: Version number example

bit is not set, and is buffered in the reservation station entry corresponding to the *add* instruction. The instruction then receives a speculative version of its second operand. The instruction can now fire and send a speculative result to its consumers.

Next the instruction receives a newer, speculative version of its second operand. The instruction fires and sends another speculative result to its consumers. The instruction then receives a null commit message for its first operand. Since the operand value has not changed, and the other operand is still speculative, the instruction does not re-fire and operand 1 is marked as non-speculative. Next the instruction receives a null commit bit for the second operand. Since all the operands of the instruction have received their commit bit and their values haven't changed, the instruction can now fire and send a null commit bit to its consumers. Finally, we see that the instruction receives an incorrect message for its second operand. This message is dropped and not acted upon by the instruction.

With commit bits, completion of distributed selective re-execution can be detected, and with version numbers, the computation will still be correct in the presence of reordered messages. While we have used load/store dependence prediction as the driving example for this mechanism, *any* data speculation scheme may use this underlying framework for low-overhead recovery, so long as it obeys the rules of the mechanism: The last version sent is always the correct one (with no versioning support), or the highest version number is always the correct one (with versioning support). Thus, many types of data

value speculation may make use of this common framework for low-overhead recovery.

Version numbers also provide a convenient mechanism to throttle speculation. ALUs can be prevented from firing speculatively when the version number of their result reaches a certain maximum value. This throttling can be done for several reasons. Having a large number of speculative firings can result in extra traffic in the network, and extra contention in the ALUs, potentially reducing performance. Also, speculative firing of ALUs consumes dynamic power. Hence, speculative execution of ALUs can be throttled using version numbers to save energy. In the next subsection, we look at the impact of speculative firing of ALUs on performance using the GPA simulator and the TRIPS prototype simulator.

4.1.2.1 Impact of Speculative Execution—GPA Simulator

To find the impact on performance for various maximum values of version numbers, we ran experiments using the GPA simulator varying the number of times an instructions is allowed to fire speculatively. Table 4.1 shows the performance for the various benchmarks, when we vary the maximum number of speculative executions for an instruction. Loads that arrive at the memory interface send speculative values to their consumers in the presence of earlier unresolved stores. Later arriving stores that match the address of the loads, and are earlier in program order, are allowed to wakeup these loads. The number of load replies is also bound by the maximum version number allowed.

Maximum speculative firing	1	2	3	4	5	6
ammp	1.52	1.52	1.52	1.51	1.51	1.51
art	1.89	1.74	1.74	1.74	1.74	1.88
bzip2	2.14	2.13	2.13	2.13	2.13	2.13
compress	1.55	1.55	1.55	1.55	1.55	1.55
equake	1.20	1.24	1.31	1.36	1.36	1.37
m88ksim	1.10	1.08	1.08	1.07	1.06	1.06
mcf	0.79	0.80	0.79	0.77	0.75	0.73
mgrid	1.68	1.60	1.62	1.62	1.58	1.55
mpeg2encode	3.12	3.13	3.13	3.13	3.14	3.14
parser	1.30	1.30	1.30	1.30	1.30	1.29
twolf	1.11	1.13	1.13	1.13	1.12	1.11
hydro2d	1.34	1.32	1.28	1.23	1.20	1.19
tomcatv	3.82	3.82	3.82	3.82	3.82	3.82
turb3d	0.72	0.72	0.70	0.69	0.68	0.68
Mean	1.40	1.40	1.39	1.38	1.37	1.36

Table 4.1: DSRE IPC variation with increasing maximum speculative firing on the GPA simulator

The loads send their commit bits only after all previous stores have resolved.

From Table 4.1, we see that benchmarks like *bzip2, compress, parser, mpeg2encode, twolf*, and *tomcatv* are relatively unaffected when we increase the number of times ALUs are allowed to fire speculatively. In these benchmarks, either there are very few load-store dependences, or these dependences are satisfied by the first arriving store. Figure 4.5 plots number of load replies for different maximum version number, normalized to the number of loads executed when we allow only one speculative value per operand. For these benchmarks, we see that the number of loads executed is relatively unchanged with maximum allowed speculative execution for an operand.

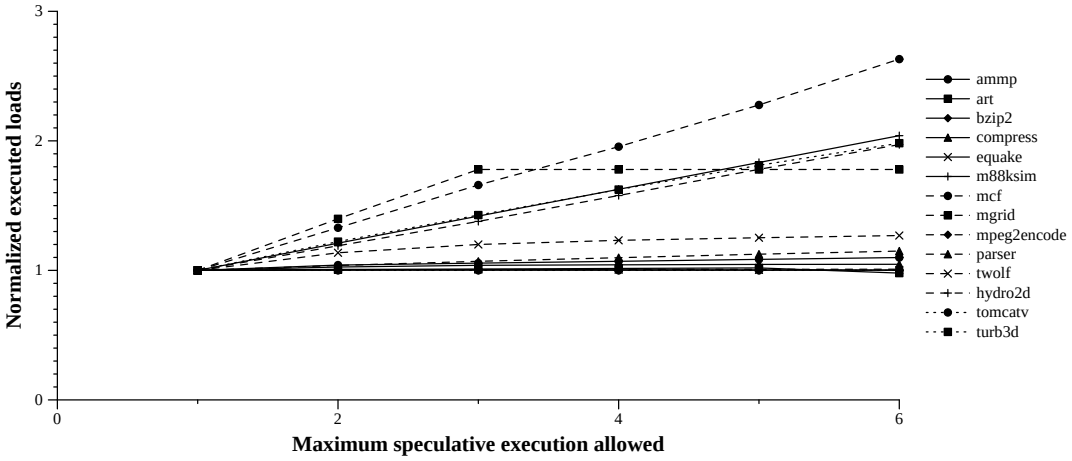

Figure 4.5: Normalized executed loads for various maximum speculative execution allowed

---

The performance of *art* decreases with increasing maximum version number allowed, and then increases when the maximum speculative execution is set to 6. With selective re-execution, later arriving stores are not allowed to wakeup a load, if the load has already received its value from a store that is later in program order than the arriving store. *art* has some loads that match with multiple earlier stores. These stores result in multiple speculative load executions when we increase the maximum version number, thus decreasing the performance. However, when the maximum speculative execution is set to 6, the number of load executions due to matching stores is actually reduced, because the store that is later in program order arrives at the memory interface before a matching store that is earlier in program order. Both these stores match the address of a later program order load, but only one store is allowed to wakeup the load, resulting in fewer load replies. The same phenomenon is

seen in *equake*, which shows increasing performance with increasing number of allowed maximum executions. From Figure 4.5, we see that the number of load replies increases as we increase the maximum allowed version number for *art*, and decreases when the maximum number of allowed speculative executions is 6. For *equake* the number of load replies decreases as we increase the maximum allowed version number for an operand.

ammp, m88ksim, mcf, mgrid, hydro2d, and *turb3d* show a reduction in performance when we increase the maximum allowed version number. For these benchmarks, as seen from Figure 4.5, the number of load replies increases as we increase the maximum allowed version number. In *ammp, mcf*, and *mgrid*, increasing the maximum allowed version number results in more versions of the same load reaching the memory interface. The speculative loads result in a larger number of speculative load replies. In *m88ksim*, increasing the maximum allowed version number results in a higher number of load replies because multiple speculative versions of a store end up waking up the same load. *hydro2d* and *turn3d* exhibit both the above mentioned behavior, and hence suffer reduction in performance.

Figure 4.6 plots the number of arithmetic instructions executed when we increase the maximum allowed version number, normalized to the total number number of arithmetic instructions executed when we allow each instruction to execute at most once speculatively. From Figure 4.6, we see that the number of arithmetic instructions executed increases as we increase the maximum allowed version number for all the benchmarks.

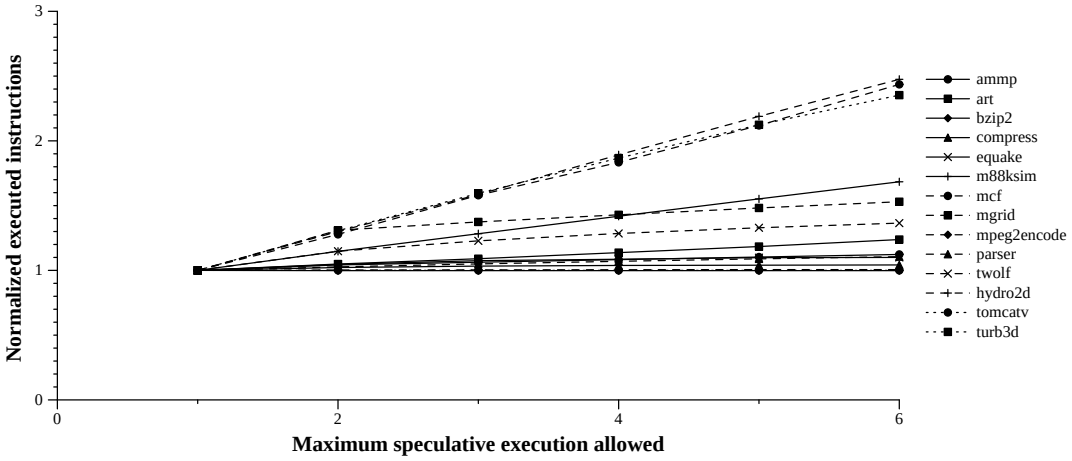

Figure 4.6: Normalized executed arithmetic instructions for various maximum speculative execution allowed

---

In summary, increasing the maximum version number allowed results in an increase in the number of instructions executed. The higher ALU and network traffic, resulting from the large number of instructions executed, generally lowers performance. A few benchmarks show improved performance with increasing version number. However, this improvement in performance is a result of the earlier resolution of matching stores that results in fewer executed loads. Since most of the benchmarks either show no improvement in performance or reduction in performance when we increase the number of allowed maximum version number, we restricted the number of speculative executions to one in the GPA simulator.

4.1.2.2 Impact of Speculative Execution—TRIPS Prototype Simulator

Maximum speculative firing	1	2	3	4	5	6
a2time01	0.765	0.910	0.927	0.936	0.942	0.942
aifftr01	0.606	0.618	0.630	0.651	0.654	0.652
aifirf01	1.036	1.055	1.072	1.069	1.062	1.062
aiifft01	0.585	0.606	0.623	0.641	0.644	0.645
autcor00	1.210	1.210	1.210	1.210	1.210	1.210
basefp01	0.885	0.890	0.894	0.886	0.876	0.867
bezier01	1.740	1.748	1.698	1.670	1.678	1.680
bitmnp01	0.779	0.783	0.779	0.773	0.770	0.768
cacheb01	0.692	0.699	0.710	0.699	0.692	0.691
canrdr01	1.292	1.342	1.351	1.352	1.353	1.353
conven00	0.538	0.538	0.538	0.538	0.538	0.538
fft00	1.359	1.382	1.408	1.408	1.408	1.408
idctrn01	0.738	0.735	0.737	0.770	0.765	0.760
iirflt01	0.525	0.525	0.555	0.636	0.623	0.612
ospf	0.641	0.705	0.714	0.715	0.717	0.717
pntrch01	0.822	0.827	0.822	0.969	0.981	0.983
pktflow	1.001	1.054	1.046	1.047	1.046	1.044
puwmod01	0.801	0.814	0.790	0.762	0.729	0.698
routelookup	0.573	0.573	0.573	0.573	0.573	0.573
rspeed01	0.804	0.837	0.866	0.875	0.883	0.884
tblook01	0.776	0.817	0.818	0.818	0.818	0.818
ttsprk01	0.667	0.697	0.698	0.700	0.701	0.703
viterb00	0.779	0.805	0.828	0.838	0.847	0.847
Mean	0.777	0.799	0.807	0.822	0.820	0.816

Table 4.2: DSRE IPC variation with increasing maximum speculative firing on the TRIPS simulator

---

To validate the results shown in the last subsection, we ran experiments using the TRIPS prototype simulator, and studied the performance for

different maximum version numbers. Table 4.2 shows the performance of the EEMBC benchmarks for different values of maximum version number on the TRIPS prototype simulator.

From Table 4.2, we see that the variation in mean performance on the TRIPS simulator is different from what is seen on the GPA simulator. On the TRIPS simulator, the mean performance increases as we increase the maximum number of times an instruction is allowed to execute speculatively from 1 to 4, and then decreases. To understand the reason for this behavior, we looked at the benchmarks that show the most difference in performance as we increased the maximum version number. These benchmarks include *a2time01, canrdr01, idctrn01, iirflt01, pntrch01, rspeed01*, and *viterb00*.

All the EEMBC benchmarks consist of a main loop that is repeatedly executed for a set number of iterations that can be specified by the user. A large number of these benchmarks write their output using a pointer to a structure. The benchmarks write their results after each major computation within the loop. Hence, there are multiple loads and stores to this pointer in the instruction window. To illustrate this behavior, Figure 4.7 shows part of the source code from the inner loop of the benchmark *rspeed01*. *rspeed01* repeatedly computes the road speed based on differences between timer counter values. The calculation involves straight-forward arithmetic, but must also deal with the situation when the timer rolls over, or when the measurement results show abrupt changes.

The main loop in *rspeed01* repeatedly computes the value of three vari-

```
/* Time to update ? */
if( toothCount1 >= tonewheelTeeth / 2 ) {
   /* Yes, */
  if( toothTimeAccum1 >
      MAX_TOOTH_TIME *tonewheelTeeth / 2 )  {

    /* ...check for zero road speed */
    roadSpeed1 = 0 ;
  }
  else {
    /* ...or compute road speed */
    roadSpeed1 = (varsize)(SPEEDO_SCALE_FACTOR
      (toothTimeAccum1 / tonewheelTeeth * 2 ));
    /* ...then reset the filter counter */
     toothCount1 = 0 ;
    /* ...and clear the accumulator */
    toothTimeAccum1 = 0 ;
   }
}

WriteOut( roadSpeed1 ) ; /* Store result */
```

Figure 4.7: Code in the main loop of *rspeed01*

ables stored in memory, *roadSpeed1, roadSpeed2*, and *roadSpeed3*. Figure 4.7 shows the code for the computation of *roadSpeed1*. After computing each value, the result is stored using the *WriteOut* function. Figure 4.8 shows the source code for this function. This function uses the address in *RAMfilePtr* pointer to write the computed result. The *WriteOut* function writes the value of the result and increments the pointer.

The compiler breaks up the inner loop into a number of hyperblocks. The computation and storage of *roadSpeed1, roadSpeed2*, and *roadSpeed3* is done using three separate hyperblocks that can be in the instruction window at the same time. Figure 4.9 shows part of the TIL code for one of these hyperblocks. The predicated stores enforce bounds checking. From Figure 4.9,

```
n_void
WriteOut( varsize value ) {
  if (( RAMfilePtr+RAMfile_increment) > RAMfileEOF
     RAMfilePtr = RAMfile;
  *(varsize *)RAMfilePtr = value;
  RAMfilePtr += RAMfile_increment;
} /* End of function 'WriteOut' */
```

Figure 4.8: WriteOut function code from *rspeed01*

---

we see that the program repeatedly loads and stores to the address of the *RAMfilePtr* variable. Any load to this address has to get its value from the most recent store. When this code is executed on the TRIPS prototype simulator, the loads to *RAMfilePtr* address resolve before all the previous stores to *RAMfilePtr* address. Matching stores, which arrive after the load, wake up these loads, and the loads send the updated value to their consumers.

Matching stores, which arrive after a load has already speculatively executed the maximum number of times, are not allowed to wake up the load. These loads send their non-speculative result after all prior stores have resolved. If the load receives the last speculative value from the last matching store before the load, it can send a null commit message, when all prior stores resolve. Otherwise, the load has to send the new store value, along with the commit bit. Thus, there is no benefit to using DSRE if the last speculative execution of the load did not get its value from the latest matching store before the load. Matching stores that arrive at the DT do not wakeup loads, if the load has already received the value from a store that is later in program order to the matching store.

```
.bbegin t_run_test$25

    entera  $t18, RAMfilePtr
    ld      $t19, ($t18) L[6]
    entera  $t20, roadSpeed3$$622(
    ld      $t21, ($t20) L[7]
    sd      ($t19), $t21 S[8]
    entera  $t22, RAMfile_incremer
    lws     $t23, ($t22) L[9]
    entera  $t24, RAMfilePtr
    ld      $t25, ($t24) L[10]
    mul     $t26, $t23, $t0
    add     $t27, $t25, $t26
    entera  $t28, RAMfilePtr
    sd      ($t28), $t27 S[11]
    entera  $t37, RAMfilePtr
    sd_t<$t34>  ($t37), $t36 S[14]
    null_f<$t34>    $t38
    sd      ($t38), $t38 S[14]
    entera  $t39, RAMfilePtr
    ld      $t40, ($t39) L[15]
```

Figure 4.9: Piece of TRIPS intermediate language (TIL) code from a *rspeed01* hyperblock to show loads and stores to the address of *RAMfilePtr*

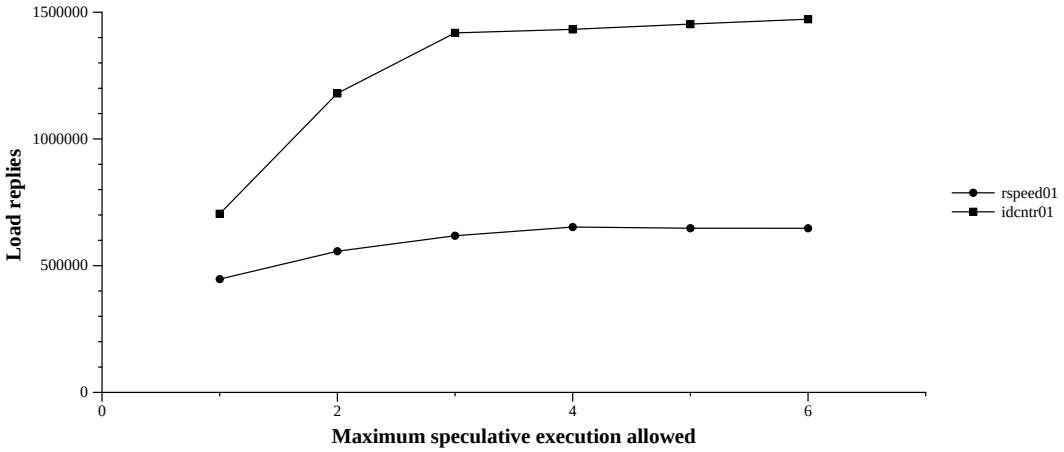

Figure 4.10: Number of load executions for various maximum speculative execution

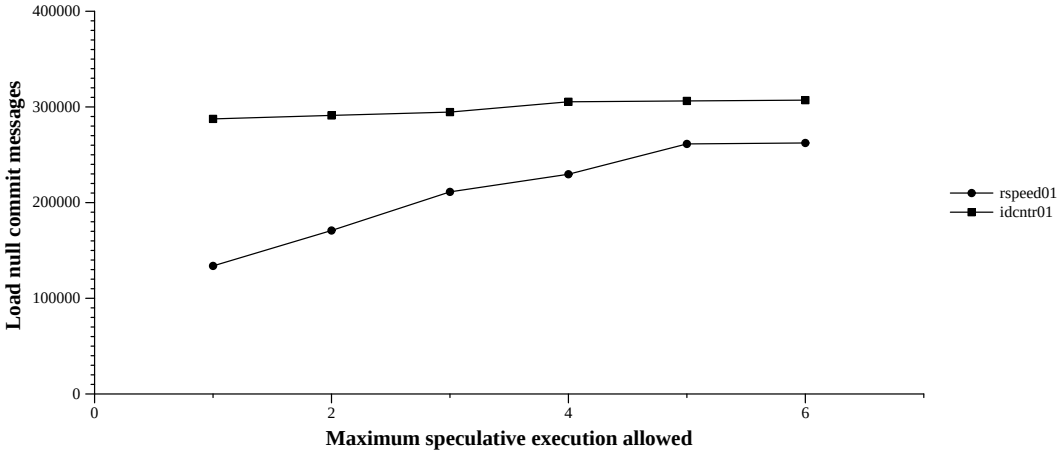

Figure 4.11: Number of load null commit messages for various maximum speculative execution

---

When we increase the number of times a load is allowed to execute speculatively, we increase the number of stores that are allowed to wakeup the load. The increase in the number of stores in turn increases the chances of the load executing speculatively with the right store value. Thus, loads send a larger number of null commit messages when we increase the maximum allowed version number. Figure 4.10 shows the number of times loads execute and Figure 4.11 shows the number of load null commit messages, when we increase the number of times loads are allowed to execute speculatively. We see from Figure 4.10 and Figure 4.11 that for *rspeed01*, the number of null commit messages sent by the DT increases as we increase the maximum version number, while the number of speculative load executions saturates. The larger number of null commit messages result in higher performance for *rspeed01* as we increase the maximum version number.

Increasing the maximum version number does not always result in higher performance. For some benchmarks like *idctrn01, aifirf01, basefp01*, and *matrix01*, performance increases at first and then decreases. To understand this behavior, we looked at the number of null commit messages and speculative load executions in these benchmarks. Figure 4.10 shows the number of times loads execute speculatively, and Figure 4.11 shows the number of null commit messages for *idctrn01* when we increase the number of times loads are allowed to execute speculatively. From Figure 4.10 and Figure 4.11, we see that for this benchmark, the number of load null commit messages saturates after the maximum number of speculative executions reaches 4, while the number of speculative loads executed keeps increasing. Hence, peak performance is obtained when we restrict the number of maximum speculative execution to 4 for *idctrn01*.

In summary, the number of speculative load executions and load null commit messages is benchmark dependent, and is a function of the arrival order of loads and stores at the DT. There are fewer load re-executions if the last program-order matching store before the load arrives at the DT before earlier stores. For the benchmarks that we study in this dissertation, maximum performance is obtained when we restrict the number of maximum speculative firing to 4. Hence, we use this value for the rest of experiments in this dissertation. Future work can involve examining policies that dynamically vary this threshold, depending on the number of speculative executions and null commit messages being sent for a particular phase of a benchmark.

4.2 DSRE Evaluation

In this section, we look at the performance of DSRE and compare it to the other load/store recovery schemes. We first present results obtained using the GPA simulator, and then present results obtained using the detailed TRIPS prototype simulator. We also present an analysis of the results obtained using the TRIPS prototype simulator.

4.2.1 DSRE Performance

Table 4.3 shows the performance of the simulated GRID processor with all of the load/store speculation policies we evaluate in this paper. Performance is displayed in instructions per cycle (counting useful, non-overhead, committed instructions only). We assumed that flushes are rolling, initiated when a misprediction is first detected, which is a higher-performance assumption than initiating flushes when the block containing the faulting instruction is ready to commit.

Column two (the leftmost data column) shows performance using conservative ordering (*cons*), in which every load waits for all prior stores to complete. As we showed in Chapter 2, this conservative model is by far the worst-performing model. The third column shows performance with a pure re-execution mechanism (*DSRE*), in which all loads issue as soon as they are ready, and re-execute if an earlier store resolves to the same address. As discussed in the last section, we restrict the number of times instructions are allowed to fire speculatively to one on the GPA simulator. Pure DSRE pro-

	No flush		Flush on load mis-speculation		
Benchmark	**cons (IPC)**	**DSRE (IPC)**	**all-stores (IPC)**	**one-store (IPC)**	**oracle (IPC)**
ammp	0.94	1.52	2.41	3.11	3.96
art	1.37	1.89	3.72	3.50	3.73
bzip2	1.90	2.14	3.16	3.23	3.24
compress	1.40	1.55	1.56	1.56	1.66
equake	0.79	1.20	1.71	1.71	1.75
m88ksim	0.88	1.10	0.93	1.28	2.31
mcf	0.42	0.79	0.87	0.83	0.88
mgrid	1.27	1.68	1.31	1.56	4.23
mpeg2encode	2.63	3.12	3.43	3.32	3.51
parser	1.27	1.30	1.31	1.31	1.32
twolf	0.88	1.11	1.27	1.36	2.09
hydro2d	0.78	1.34	1.03	1.73	3.35
tomcatv	2.88	3.82	4.96	4.95	4.96
turb3d	0.53	0.72	0.62	0.74	3.85
Mean	0.97	1.40	1.42	1.61	2.30

Table 4.3: IPC of load/store recovery schemes on the GPA simulator

vides a 40% performance boost over conservative load-store ordering, making it a potential alternative to dependence prediction. As shown in Chapter 5, the difference in performance between the DSRE and the oracle policy is primarily due to the commit wave falling behind the execution wave.

Columns 4 and 5 show the performance of traditional dependence prediction, using *all-stores* and *one-store* to selectively stall loads that are predicted to be dependent, and flush the pipeline if a load is speculatively issued before a conflicting store. *all-stores* shows almost exactly the same average performance as DSRE. The more complex, but more aggressive, *one-store* pol-

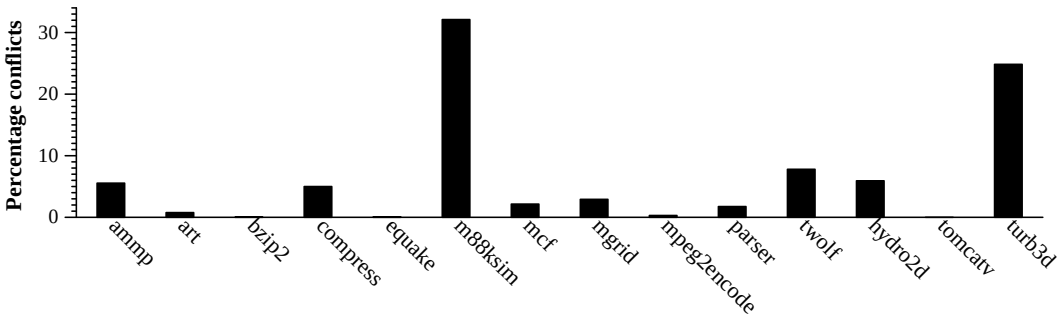

Figure 4.12: Percentage of loads that conflict with earlier stores

icy improves performance over the base case by an additional 13%, since some stalled loads can proceed earlier when their conflicting store arrives, instead of waiting for all stores. Despite these relatively large performance gains, a large gap still exists with the upper-bound performance of an oracle, which shows a mean IPC of 2.30, 43% higher than the *one-store* policy. In the next chapter, we will look at enhancements to the base DSRE technique that bridges this gap in performance.

The *one-store* dependence policy results in performance comparable to the oracle policy for all but six benchmarks. These are *ammp, m88kim, mgrid, twolf, hydro2d*, and *turb3d*. To understand the low performance in these benchmarks, we looked at the dynamic load-store dependences in these benchmarks. Figure 4.12 shows the number of loads that depend on earlier stores during execution, as a fraction of the total number of memory instructions. As seen from Figure 4.12, these benchmarks have a significant number of loads conflicting with earlier stores. The *one-store* predictor is unable to predict these

dependences correctly. *m88kim, mgrid,* and *twolf* suffer too many flushes due to loads being incorrectly predicted conflicting. In *ammp* and *hydro2d*, the predictor is too conservative and predicts a large number of independent loads as being dependent. *turb3d* has a mix of loads incorrectly predicted conflicting and non-conflicting.

Table 4.4 compares the performance of DSRE against the various load/store recovery schemes across the set of EEMBC benchmarks on the TRIPS prototype simulator. From Table 4.4, we see DSRE provides 16% improvement over the conservative scheme. The performance improvement is lower than what is seen with the GPA simulator due to the following reasons:

1. As explained in Chapter 7, the network and ALU contention are more accurately modeled in the TRIPS prototype simulator, and hence influence performance to a larger degree with DSRE.

2. Since the LSQ is physically distributed, the arrival of stores at the data tile is communicated through the data status network (DSN). Hence, stores take longer to resolve in the DT. For example, arrival of a store at DT0 is communicated to DT3 after 3 cycles. Since loads can send their commit bits only after all the stores before them have resolved, propagation of load commit bits is delayed by the distributed nature of the LSQ.

3. The reissue pipeline in the DT adds an extra cycle delay to the propagation of the commit bit.

	No flush		Flush on load mis-speculation		
Benchmark	**cons (IPC)**	**DSRE (IPC)**	**all-stores (IPC)**	**one-store (IPC)**	**oracle (IPC)**
a2time01	0.702	0.936	0.793	0.842	2.418
aifftr01	0.560	0.651	0.714	0.710	2.477
aifirf01	0.884	1.069	1.692	1.677	2.635
aiifft01	0.547	0.641	0.676	0.698	2.592
autcor00	1.208	1.210	1.208	1.208	1.210
basefp01	0.845	0.886	1.068	1.074	1.212
bezier01	1.195	1.670	2.789	2.793	2.789
bitmnp01	0.678	0.773	0.920	0.945	1.714
cacheb01	0.579	0.699	0.861	0.993	1.535
canrdr01	1.197	1.352	1.400	1.431	1.483
conven00	0.535	0.538	0.538	0.538	0.538
fft00	1.052	1.408	2.725	2.726	2.727
idctrn01	0.652	0.770	1.566	1.530	2.719
iirflt01	0.489	0.636	0.849	0.869	1.944
ospf	0.633	0.715	0.906	0.908	0.917
pntrch01	0.820	0.969	0.930	0.900	1.039
pktflow	0.896	1.047	1.187	1.187	1.272
puwmod01	0.703	0.762	0.922	0.913	2.191
routelookup	0.573	0.573	0.573	0.573	0.573
rspeed01	0.697	0.875	0.887	0.889	2.129
tblook01	0.751	0.818	0.821	0.825	0.854
ttsprk01	0.636	0.700	0.743	0.749	0.782
viterb00	0.647	0.838	1.490	1.780	3.053
Mean	0.709	0.822	0.955	0.972	1.361

Table 4.4: IPC of load/store recovery schemes on the TRIPS prototype simulator

4. Only one load is allowed to send a commit bit every cycle. Hence, if there are a number of loads after a store that become non-speculative when the store arrives, we can process null commit messages for only one load per cycle.

5. A large fraction of the variables in the EEMBC suite are either global or static variables. The current compiler did not register allocate these variables, and is forced to allocate them in memory, resulting in a large number of load-store dependences in these programs.

6. The load-store pipeline in the DT stalls during load-store forwarding. Thus, programs with a a large number of load-store dependences incur more stalls in the DT. Also, when a store wakes up a load, the pipeline stalls when the load is being processed to get the forwarded value from the store.

7. For a number of EEMBC benchmarks, within a hyperblock we have loads that compute the data for stores. These loads are interleaved with other loads and stores in the block. This load-to-store dependence serializes the propagation of commit bits between stores. This serialization is illustrated using a piece of TIL code from *a2time* in Figure 4.13. For brevity we have shown only the loads and stores in the block that exhibit load-to-store dependence, and omitted the rest of the instructions. The variables being stored in Figure 4.13 are global variables, and hence were not register allocated by the compiler.

```
.bbegin t_run_test$14 ;
   ld $t7, ($t6) L[2]
   entera $t8, angleCounter
   sd ($t8), $t7 S[3]
   entera $t33, angleCounter
   ld $t34, ($t33) L[10]
   entera $t35, angleCounterLast1$$68
   sd ($t35), $t34 S[11]
   entera $t38, pulseDeltaTime1$$6799
   ld $t39, ($t38) L[13]
   add $t40, $t37, $t39
   entera $t41, rotationTime1$$6826
   sd ($t41), $t40 S[14]
```

Figure 4.13: Piece of TRIPS intermediate language (TIL) code from *a2time01* to show load-to-store dependence

---

From Figure 4.13, we see that the store with LSID 3 depends on the load with LSID 2 for its value. This store can resolve only after the load sends its commit bit. The load can send its commit bit only after all previous stores before the load have resolved, and received their commit bit. The load with LSID 10 can send its commit bit only after all previous stores, including the store with LSID 3, have received their commit bit. The store with LSID 11 depends on the load with LSID 10 for its value. Similarly, the store with LSID 14 depends on the load with LSID 13 for its value. From the TIL code shown in Figure 4.13, we see that the commit bit forwarding for the three stores listed above are serialized, thus delaying the propagation of the commit wave.

The worst performing benchmarks with DSRE include *a2time01, aifftr01,*

aifirf01, aiifft01, bitmnp01, cacheb01, idctrn01, iirflt01, matrix01, pntrch01, puwmod01, and *rspeed01*. All these benchmarks write their output using the *RAMfilePtr* variable which results in multiple stores to the same address in the instruction window. When a load gets woken up multiple times by matching stores, it results in the DT stalling for a cycle every time a value is forwarded. The large number of DT stalls, along with the extra network and ALU traffic generated by the multiple speculative executions, result in the poor performance for these benchmarks.

For some benchmarks like *autocor00, canrdr01, conven00*, and *tblook01*, DSRE performance is similar to that of the oracle policy. These benchmarks do not have a large number of global or static variables. Hence, more variables in these benchmarks are register allocated, resulting in fewer load-store dependences. Also, these benchmarks have small average block size, resulting in fewer useful instructions in the instruction window. Small average block size reduces the number of in-flight load-store dependences, but also results in overall poor performance because of the large overhead associated with fetching and committing the small blocks. Table 4.5 shows the average block size of the EEMBC benchmarks along with the IPC with perfect load-store prediction. We see from Table 4.5 that the benchmarks with small average block sizes have the poorest performance.

Benchmark	Average Block Size (Instructions)	Oracle IPC
a2time01	58.060	2.418
aifftr01	59.252	2.477
aifirf01	49.973	2.635
aiifft01	63.035	2.592
autcor00	18.066	1.210
basefp01	24.231	1.212
bezier01	35.660	2.789
bitmnp01	35.732	1.714
cacheb01	51.703	1.535
canrdr01	22.314	1.483
conven00	7.130	0.538
fft00	31.106	2.727
idctrn01	50.826	2.719
iirflt01	50.658	1.944
ospf	18.739	0.917
pntrch01	34.092	1.039
pktflow	23.889	1.272
puwmod01	57.989	2.191
routelookup	18.871	0.573
rspeed01	60.233	2.129
tblook01	23.821	0.854
ttsprk01	22.602	0.782
viterb00	42.696	3.053
Mean	37.421	1.361

Table 4.5: Average block size and IPC with oracle policy for the EEMBC benchmarks

---

Dependence prediction with the *all-stores* and *first-store* predictor for *fft00* has performance close to oracle. *fft00* has a significant number of loads, but a large majority of these loads are independent loads. Hence, dependence

prediction works very well for this benchmark. With DSRE, these independent loads generate a large number of null commit messages. The extra ALU and network traffic, generated by the null commit messages, result in the poor performance of DSRE when compared to dependence prediction.

DSRE outperforms dependence prediction for *a2time01* and *pntrch01*. As explained earlier, these benchmarks have the serialized loads to the *RAM-filePtr* function that don't benefit from dependence prediction. Dependence prediction also causes load violation flushes in these benchmarks, before the predictor is trained to predict these loads as conflicting. DSRE is able achieve higher performance than dependence prediction because of the lack of pipeline flushes, and the null commit messages that are sent when the correct matching store happens to wakeup these loads.

On the TRIPS prototype simulator, the 1-bit *all-stores* predictor and the 3-bit *first-store* predictor improve the mean performance by 35% and 37% over the conservative policy. There is still 42% and 40% difference in performance between the predictors and the oracle policy. The large difference in performance between the *all-stores* predictor and oracle can again be explained using the load-to-store dependence shown in Figure 4.13.

The *all-stores* predictor uses a PC-indexed 1-bit table to identify loads that cause a dependence violation. The load PC is computed by *xor-ing* the top 5 bits of the blocks address with the reversed LSID of the load. If the bit is set for a load, the load is deferred and sends its reply only after all prior stores have resolved. If the deferred load happens to be part of a load-to-store

```
.bbegin t_run_test$40 ;
   entera $t2, RAMfilePtr
   ld $t3, ($t2) L[0]
   entera $t4, firingTime3$$681
   ld $t5, ($t4) L[1]
   sd ($t3), $t5 S[2]
   entera $t8, RAMfilePtr
   ld $t9, ($t8) L[4]
   mul $t10, $t7, $t0
   add $t11, $t9, $t10
   entera $t12, RAMfilePtr
   sd ($t12), $t11 S[5]
   entera $t23, RAMfilePtr
   ld $t24, ($t23) L[9]
```

Figure 4.14: Piece of TRIPS intermediate language code (TIL) from *a2time01* to show load-to-store and store-to-load dependence

---

dependence chain, the propagation of the load result is delayed, thus resulting in poor performance.

Figure 4.14 shows TIL code, again from the *a2time01* benchmark that illustrates this case. We have shown only the instructions that highlight the load-to-store dependence and the store-to-load dependence in Figure 4.14. From Figure 4.14, we see that that store with LSID 2 depends on the loads withe LSID 0 and LSID 1 for its address and data. The store with LSID 5 depends on the load with LSID 4. Finally, the load with LSID 9 uses the value stored by the store with LSID 5.

During program execution, the load with LSID 9 causes a load-store dependence violation, and the bit corresponding to this load is set in the 1-bit dependence prediction table. When this load is encountered again during

program execution, it is deferred and waits for all prior stores to resolve before sending its reply. Since the stores before the load are serialized due to a load-to-store dependence, it takes longer for the stores to resolve, thus delaying the load reply. Code similar to that shown in Figure 4.14 is found in a number of EEMBC benchmarks that write their result using the *RAMfilePtr*. Like the DSRE policy, the *all-stores* predictor performs similar to the oracle policy on benchmarks with few load-store dependences and small average block size.

The more aggressive 3-bit *first-store* predictor performs better than the *all-stores* predictor for most benchmarks. There is still a large gap in performance between the *first-store* predictor and the oracle policy. This difference can be attributed to the presence of multiple matching stores to the same address in the instruction window. As described in Section 4.1.2.2, the EEMBC benchmarks use a global pointer to store the output after each computation. Loads and stores repeatedly access this pointer during program execution, resulting in a load matching with multiple, earlier in-flight stores. These multiple stores result in the *first-store* predictor becoming more conservative, and behaving like the *all-stores* predictor for these loads, thus reducing performance.

The *first-store* predictor does benefit from an aggressive load wakeup policy for some benchmarks. *viterb00* shows an 19.4% improvement in performance with the *first-store* predictor when compared to the *all-stores* predictor.

The *all-stores* predictor performs better than the *first-store* predictor for *aifirf01* and *matrix01*. The *first-store* predictor results in a larger number of flushes in these benchmarks for loads that match with multiple stores. For

these loads, the *first-store* predictor can incur up to five extra flushes in each data tile before it is trained to predict to defer the loads until all prior stores resolve. These extra flushes result in lower performance with the *first-store* predictor for these benchmarks.

In summary, there is a significant difference in performance between DSRE and the the oracle policy across the set of EEMBC benchmarks. The difference in performance is primarily due to multiple matching stores for a load that results in a large number of stores forwarding their value to loads in the DT. Load-store forwarding results in the DT stalling for a cycle, thus reducing performance. The multiple store forwarding also generates extra ALU and network traffic that reduces performance. The EEMBC benchmarks also have a number of load-to-store dependence that serializes propagation of commit bit among stores. This delay in commit propagation also contributes to the poor performance of DSRE.

Dependence prediction using *all-stores* and *first-store* predictor also performs poorly when compared to the oracle policy across the set of EEMBC benchmarks. In this case also, the difference in performance is primarily due to multiple matching stores to the same address in the main loop of these benchmarks. The multiple matching stores incorrectly wakeup loads, resulting in a larger number of flushes. In the steady state, the predictor becomes conservative for these loads, and defers the load reply until all prior stores resolve. The *first-store* predictor will yield better performance if it can accurately identify the matching store for each load, and allow only that store to wakeup up the

load. Future work can involve looking at predictors that provide this functionality in the distributed TRIPS environment. In the next few sections, we compare the performance of DSRE against the various dependence prediction with perfect branch prediction that results in a larger instruction window and perfect level one and level two caches that results in lower memory latency.

4.2.2 DSRE Performance with Perfect Branch Prediction

	No flush		Flush on load mis-speculation		
Benchmark	**cons (IPC)**	**DSRE (IPC)**	**all-stores (IPC)**	**one-store (IPC)**	**oracle (IPC)**
ammp	0.95	1.55	2.41	3.22	4.23
art	1.38	1.91	3.89	3.15	3.89
bzip2	2.35	2.79	2.51	4.73	5.31
compress	1.98	2.21	2.36	2.39	2.42
equake	0.80	1.26	1.95	1.94	1.99
m88ksim	0.89	1.13	0.95	1.53	2.44
mcf	0.43	0.89	1.07	1.03	1.09
mgrid	1.39	1.65	1.46	1.55	4.31
mpeg2encode	2.85	3.48	3.93	3.93	4.05
parser	1.38	1.41	1.43	1.43	1.44
twolf	0.97	1.21	1.33	1.49	2.79
hydro2d	0.82	1.47	1.08	2.12	3.42
tomcatv	2.93	3.90	5.12	5.10	5.12
turb3d	0.53	0.73	0.66	0.81	4.20
Mean	1.02	1.47	1.55	1.85	2.70

Table 4.6: IPC of load/store recovery schemes on the GPA simulator with perfect branch prediction

In this section, we evaluate the effect of perfect branch prediction on the various data mis-speculation recovery schemes. Perfect branch prediction results in the instruction window of the processor being filled with a larger number of useful instructions. Hence, it has the potential for higher performance by increasing the amount of instruction level parallelism that we can exploit. However, having a larger instruction window also increases the number of conflicting loads and stores that can be present in the window.

Table 4.6 shows the performance of the various load/store schemes with perfect branch prediction. Perfect branch prediction improves the mean performance of the conservative and selective re-execution scheme by 5%, *all-stores* policy by 9%, *one-store* policy by 15%, and the oracle policy by 17%. This trend is in line with what we observed in Chapter 2. Perfect branch prediction increases the number of load and store instructions in the instruction window. The conservative and selective re-execution schemes predict all loads as conflicting, thus delaying commit bit propagation to the consumers of the loads until all prior stores resolve. The *all-stores* and the *one-store* predictor are able to get a performance boost from load-store dependences that are predicted correctly.

The *one-store* policy performs similarly to the oracle policy for benchmarks that do not have a large number of load-store dependences. The conservative *all-stores* policy performs better than the aggressive *one-store* policy for *art* because of the higher number of flushes. *m88ksim* and *mgrid* suffer from a larger number of flushes, *ammp* and *hydro2d* have loads that are incorrectly

predicted dependent, and *twolf* and *turb3d* have a mix of both.

Table 4.7 shows the performance with perfect branch prediction with the TRIPS simulator. Perfect branch prediction improves the performance of the conservative policy by 3%, *all-stores* policy by 4.8%, *first-store* policy by 4.7%, and the oracle policy by 7%. This trend is similar to what is seen on the GPA simulator. The improvements for each policy is lower because of the small size of the EEMBC benchmarks, along with their loop based nature, which makes them incur fewer branch mispredictions than the SPEC benchmarks.

DSRE actually shows a reduction in mean performance with perfect branch prediction on the TRIPS prototype simulator. We analyzed the benchmarks showing the largest reduction in performance. These benchmarks have multiple matching stores to the same address. We found that with DSRE, the larger instruction window from perfect prediction resulted in more contention and traffic in these benchmarks, due to multiple stores waking up the same matching load. This extra contention reduced the mean performance of DSRE with perfect branch prediction.

	No flush		Flush on load mis-speculation		
Benchmark	**cons (IPC)**	**DSRE (IPC)**	**all-stores (IPC)**	**one-store (IPC)**	**oracle (IPC)**
a2time01	0.705	0.768	0.798	0.846	2.447
aifftr01	0.561	0.607	0.700	0.715	2.499
aifirf01	0.930	1.102	1.717	1.698	2.588
aiifft01	0.547	0.586	0.673	0.707	2.602
autcor00	1.444	1.444	1.465	1.465	1.464
basefp01	0.872	0.912	1.110	1.118	1.262
bezier01	1.194	1.741	2.783	2.776	2.789
bitmnp01	0.740	0.861	1.008	1.043	1.975
cacheb01	0.614	0.707	0.951	0.952	2.048
canrdr01	1.270	1.390	1.652	1.732	1.852
conven00	0.483	0.483	0.510	0.481	0.483
fft00	1.035	1.329	2.713	2.658	2.660
idctrn01	0.662	0.748	1.583	1.619	2.862
iirflt01	0.490	0.525	0.841	0.883	1.969
ospf	0.653	0.662	1.053	1.054	1.070
pntrch01	0.823	0.826	0.927	0.969	1.048
pktflow	0.938	1.039	1.269	1.270	1.368
puwmod01	0.711	0.809	0.928	0.919	2.268
routelookup	0.719	0.719	0.719	0.719	0.719
rspeed01	0.699	0.805	0.888	0.890	2.172
tblook01	0.769	0.727	0.864	0.899	0.886
ttsprk01	0.681	0.656	0.804	0.810	0.862
viterb00	0.647	0.780	1.491	1.800	3.046
Mean	0.730	0.793	1.001	1.018	1.457

Table 4.7: IPC of load/store recovery schemes on the TRIPS prototype simulator with perfect branch prediction

4.2.3 DSRE Performance withe Perfect L1 Data Cache

	No flush		Flush on load mis-speculation		
Benchmark	**cons (IPC)**	**DSRE (IPC)**	**all-stores (IPC)**	**one-store (IPC)**	**oracle (IPC)**
ammp	0.97	1.53	2.59	3.22	4.26
art	1.50	1.92	4.09	3.69	4.10
bzip2	2.09	2.35	3.72	3.88	3.90
compress	1.61	1.71	1.81	1.78	1.84
equake	1.04	1.36	2.50	2.51	2.58
m88ksim	0.88	1.11	0.93	1.24	2.32
mcf	0.94	1.11	1.15	1.11	1.17
mgrid	1.50	1.68	1.53	2.07	5.58
mpeg2encode	2.63	3.13	3.43	3.42	3.52
parser	1.34	1.36	1.39	1.39	1.40
twolf	0.93	1.12	1.30	1.37	2.22
hydro2d	0.95	1.35	1.46	2.62	4.11
tomcatv	3.08	4.61	7.80	7.70	7.80
turb3d	0.56	0.73	0.67	0.79	3.96
Mean	1.17	1.46	1.64	1.87	2.69

Table 4.8: IPC of load/store recovery schemes on the GPA simulator with perfect L1 D-cache

In this section, we compare the performance of DSRE against the various load issue schemes with a perfect level one data cache. Having a perfect L1 data cache increases performance by reducing the average memory latency for loads and stores.

Table 4.8 shows the performance of the various load/store schemes with perfect level one data cache. Perfect L1 D-cache improves the mean performance of the conservative scheme by 20%, selective re-execution scheme by

4.3%, *all-stores* policy by 15.5%, *one-store* policy by 16%, and the oracle policy by 17%. Perfect L1 D-cache results in larger performance improvement than perfect branch prediction for the conservative, *all-stores*, and *one-store* policies.

In the conservative policy, all loads are predicted conflicting and send their reply after all prior stores have resolved. With perfect L1 D-cache, the independent loads are able to send data to their consumers with a smaller latency, as the request always hits in the data cache. Loads predicted conflicting by the *all-stores* policy have a similar advantage, and loads that were predicted dependent incorrectly by the *one-store* predictor also benefit from the lower latency to the memory system. Hence, these schemes benefit with perfect L1 D-cache. With selective re-execution, loads access the cache when they first arrive at the memory interface. Latency of cache misses are hidden by the time it takes for previous stores to resolve, and send the commit bit for the load. Hence, selective re-execution does not benefit greatly from perfect L1 D-cache. However, this result does prove that selective re-execution can be used as a mechanism for tolerating memory latencies.

	No flush		Flush on load mis-speculation		
Benchmark	**cons (IPC)**	**DSRE (IPC)**	**all-stores (IPC)**	**one-store (IPC)**	**oracle (IPC)**
a2time01	0.703	0.937	0.791	0.842	2.422
aifftr01	0.567	0.651	0.719	0.715	2.495
aifirf01	0.885	1.067	1.693	1.681	2.614
aiifft01	0.554	0.642	0.681	0.703	2.620
autcor00	1.169	1.175	1.171	1.171	1.172
basefp01	0.854	0.887	1.071	1.076	1.225
bezier01	1.296	1.687	2.792	2.485	2.788
bitmnp01	0.679	0.773	0.922	0.945	1.716
cacheb01	0.589	0.698	0.881	1.014	1.597
canrdr01	1.205	1.356	1.401	1.434	1.486
conven00	0.535	0.538	0.538	0.538	0.538
fft00	1.052	1.408	2.734	2.735	2.736
idctrn01	0.653	0.772	1.549	1.522	2.729
iirflt01	0.490	0.637	0.836	0.867	1.951
ospf	0.669	0.742	0.941	0.942	0.945
pntrch01	0.821	0.970	0.931	0.903	1.041
pktflow	0.955	1.053	1.262	1.262	1.336
puwmod01	0.704	0.763	0.925	0.915	2.194
routelookup	0.573	0.573	0.573	0.573	0.573
rspeed01	0.698	0.875	0.888	0.891	2.142
tblook01	0.754	0.819	0.822	0.827	0.856
ttsprk01	0.638	0.700	0.744	0.749	0.783
viterb00	0.647	0.838	1.490	1.781	3.054
Mean	0.716	0.823	0.959	0.975	1.369

Table 4.9: IPC of load/store recovery schemes on the TRIPS prototype simulator with perfect L1 D-cache

Table 4.9 shows the performance across the set of EEMBC benchmarks for perfect L1 D-cache with the TRIPS simulator. Perfect L1 D-cache improves

the mean performance of the conservative scheme by 1%, selective re-execution scheme by 0.1%, *all-stores* policy by 0.4%, *first-store* policy by 0.3%, and the oracle policy by 0.6%. Again, we see that the perfect L1 D-cache results in larger performance improvements for the conservative, *all-stores*, *first-store*, and the oracle policy. The difference is not as large as what is seen on the GPA simulator because of the smaller memory footprint of the EEMBC benchmarks that results in fewer cache misses.

4.2.4 DSRE Performance withe Perfect L2 Cache

	No flush		Flush on load mis-speculation		
Benchmark	**cons (IPC)**	**DSRE (IPC)**	**all-stores (IPC)**	**one-store (IPC)**	**oracle (IPC)**
ammp	0.95	1.53	2.52	2.67	4.14
art	1.46	1.94	4.09	3.68	4.09
bzip2	2.02	2.28	3.45	3.60	3.62
compress	1.42	1.55	1.58	1.56	1.60
equake	1.03	1.36	2.41	2.46	2.56
m88ksim	0.88	1.11	0.92	1.27	2.32
mcf	0.86	0.98	1.14	1.08	1.16
mgrid	1.50	1.69	1.53	1.90	5.57
mpeg2encode	2.63	3.13	3.43	3.42	3.52
parser	1.27	1.30	1.32	1.31	1.32
twolf	0.88	1.11	1.24	1.32	2.09
hydro2d	0.94	1.35	1.43	2.71	4.11
tomcatv	3.08	4.53	7.81	7.70	7.81
turb3d	0.55	0.72	0.66	0.77	3.95
Mean	1.13	1.42	1.59	1.79	2.59

Table 4.10: IPC of load/store recovery schemes on the GPA simulator with perfect L2 cache

In this section, we examine the effect of a perfect L2 cache on DSRE and the various load issue schemes. Since the L2 cache is unified, simulating a perfect L2 cache reduces both the average instruction and data fetch latencies in the processor.

Table 4.10 shows the performance of the various load/store schemes with perfect level two cache. Perfect L2 cache improves the mean performance of the conservative scheme by 16.4%, selective re-execution scheme by 1.4%, *all-stores* policy by 12%, *one-store* policy by 11%, and the oracle policy by 12.6%. The performance gains with perfect L2 cache are lower than the gains with perfect L1 D-cache. However, the relative performance improvement for the various load-store schemes is similar to perfect L1 D-cache, with the conservative, *all-stores*, *one-store*, and oracle policies benefiting more from the lower memory latency.

Table 4.11 shows the performance across the set of EEMBC benchmarks for perfect L2 cache with the TRIPS simulator. Surprisingly, mean performance with perfect L2 cache is better than performance with perfect L1 D-cache, for all the policies on the TRIPS prototype simulator. The higher performance with perfect L2 cache can be explained by the lower instruction fetch latency for these benchmarks with perfect L2 cache. The EEMBC benchmarks have a larger I-cache miss rate than the D-cache miss rate. With perfect L1 D-cache, the I-cache misses still have to go to the L2 cache or main memory to be serviced. Since the TRIPS processor has a unified L2 cache, with perfect L2 cache, all the I-cache misses are serviced by the L2 cache. Hence,

the performance is higher with perfect L2 cache on the TRIPS simulator.

In summary, this chapter explained implementation of selective re-execution on EDGE architectures. We used load-store dependence prediction as the driving speculation mechanism and evaluated its performance using DSRE and pipeline flushing for mis-speculation recovery on the high-level GPA simulator and low-level TRIPS prototype simulator. We found that DSRE performs similar to the *all-stores* dependence prediction scheme on the GPA simulator. The performance of DSRE is lower on the TRIPS simulator when compared to the dependence prediction schemes. We found that that there is still a 40% gap between the best dependence prediction policy and the oracle policy. In the next chapter, we describe enhancements to DSRE that attempt to close this gap.

	No flush		Flush on load mis-speculation		
Benchmark	**cons (IPC)**	**DSRE (IPC)**	**all-stores (IPC)**	**one-store (IPC)**	**oracle (IPC)**
a2time01	0.704	0.938	0.795	0.844	2.434
aifftr01	0.560	0.651	0.715	0.710	2.479
aifirf01	0.886	1.070	1.703	1.692	2.639
aiifft01	0.547	0.641	0.676	0.698	2.602
autcor00	1.238	1.239	1.238	1.238	1.239
basefp01	0.861	0.899	1.087	1.093	1.243
bezier01	1.196	1.677	2.792	2.423	2.794
bitmnp01	0.680	0.775	0.923	0.947	1.723
cacheb01	0.582	0.702	0.866	1.000	1.555
canrdr01	1.212	1.367	1.414	1.448	1.500
conven00	0.536	0.539	0.540	0.539	0.539
fft00	1.053	1.409	2.732	2.731	2.733
idctrn01	0.655	0.774	1.535	1.544	2.764
iirflt01	0.492	0.640	0.853	0.912	1.987
ospf	0.635	0.715	0.909	0.911	0.919
pntrch01	0.822	0.972	0.932	0.906	1.043
pktflow	0.904	1.050	1.197	1.198	1.284
puwmod01	0.706	0.765	0.925	0.917	2.212
routelookup	0.573	0.573	0.573	0.573	0.573
rspeed01	0.700	0.879	0.892	0.894	2.164
tblook01	0.755	0.822	0.826	0.830	0.859
ttsprk01	0.638	0.701	0.746	0.751	0.785
viterb00	0.648	0.838	1.490	1.784	3.061
Mean	0.712	0.825	0.959	0.977	1.371

Table 4.11: IPC of load/store recovery schemes on the TRIPS prototype simulator with perfect L2 cache

Chapter 5

DSRE Acceleration

This chapter examines two policies, speculative commit slicing and bottom-up commit traversal, to accelerate propagation of commit bits to improve performance with DSRE. We first compare performance on the GPA simulator, and validate it using the TRIPS prototype simulator. We do an analysis of our results obtained using the TRIPS simulator to explain the performance difference still remaining between DSRE with commit slicing and oracle policy. We also study the performance of the enhanced DSRE scheme with perfect branch prediction, perfect L1 D-cache, perfect L2 cache, and for larger instruction window sizes to examine the effects of a larger useful instruction window and lower memory latency.

5.1 Accelerating Commit of Re-executed Blocks

Our results have shown that the commit traversal of the DFG is the single largest impediment to achieving performance close to that of an ideal oracle. Column 2 of Table 5.1 shows the performance of DSRE with ideal commit performance (*p-com*) on the GPA simulator. In the *p-com* policy, every load issues as soon as it reaches the memory interface, resulting in multiple

Benchmark	p-com (IPC)	oracle (IPC)
ammp	3.96	3.96
art	3.73	3.73
bzip2	3.24	3.24
compress	1.66	1.66
equake	1.75	1.75
m88ksim	2.26	2.31
mcf	0.88	0.88
mgrid	4.15	4.23
mpeg2encode	3.51	3.51
parser	1.32	1.32
twolf	2.04	2.09
hydro2d	3.35	3.35
tomcatv	4.96	4.96
turb3d	3.28	3.85
Mean	2.27	2.30

Table 5.1: Perfect commit comparison on the GPA simulator

speculative waves when a store arrives. However, the commit bits in the policy are infinitely fast, so that the commit traversal never inhibits performance. The mean IPC for *p-com* is 2.27, which is within 4% of the upper bound, demonstrating that the commit traversal is the remaining bottleneck. If the commit traversal is sufficiently fast, the performance losses due to load/store conflicts will be negligible. In this chapter, we describe two techniques for accelerating the commit traversal: *speculative commit slicing* and *bottom-up commit traversal*.

5.1.1 Speculative Commit Slicing

Our analyses have shown that a significant portion of the commit traversal's lag behind the execution traversal of the DFG is attributable to late-committing stores. As we saw in Section 4.2 of Chapter 4, the load-to-store dependences in the EEMBC benchmarks, along with the nature of the TRIPS architecture, results in delay in the resolution of stores in the instruction window. Only after all prior stores have received their commit bit can loads forward their commit bits to their consumers (provided, of course that the loads' addresses have also received their commit bits). A single slow store can thus block all subsequent loads from forwarding any commit bits until quite late. Since loads typically reside at the head of dependence chains, a single slow store may thus block any significant advance execution of the commit wave.

To accelerate the commit traversal, we allow some loads to forward their commit bits speculatively—although no modifications are made to architectural state until safe commit is guaranteed. A load that is unlikely to conflict can forward its commit bit, and if no violation eventually occurs, the commit bit speculation improves performance. If a conflict does occur, the pipeline needs to be flushed, since there is no way to recall the commit bit. This strategy is safe because no architectural state is written until all commit bits are received, at which point the processor can detect any violations. Commit slicing thus begins to resemble the two-phase commit approach in databases. In two-phase commit, individual transactions write to a log file

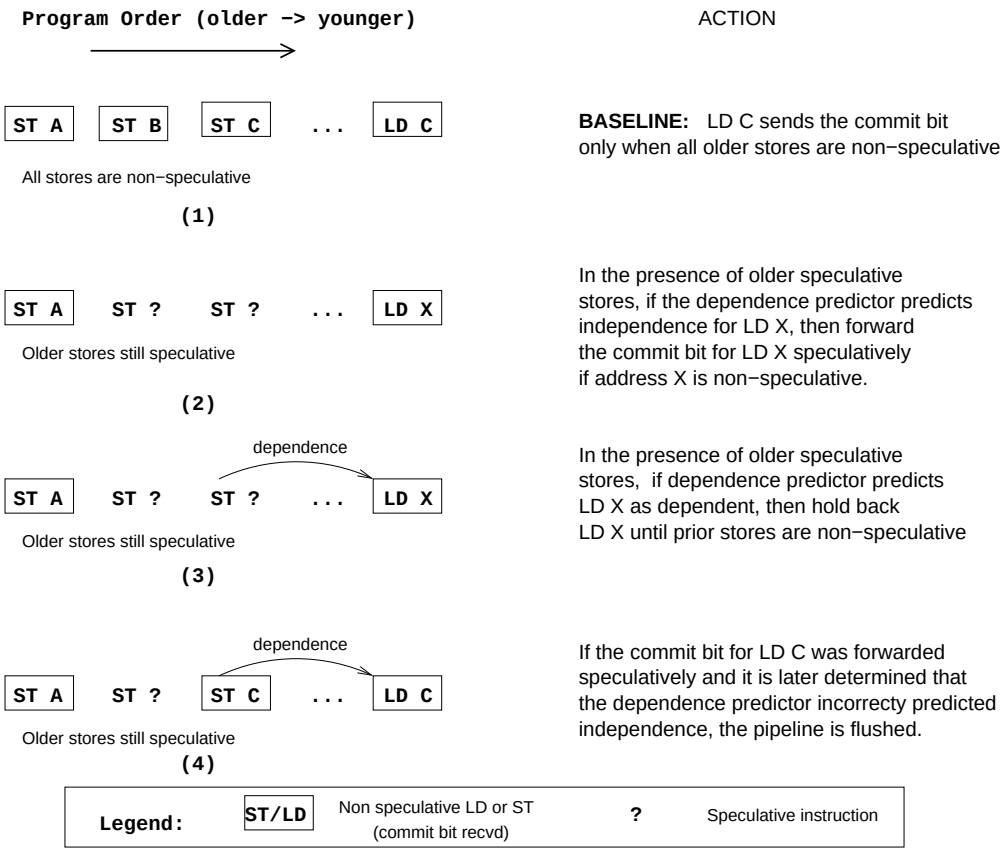

Figure 5.1: Speculative commit slicing

before updating the database on the disk. Writing to the log file improves performance by ensuring that accesses to the database are not serialized. The database application uses the log file to rollback in case of data corruption. In commit slicing, individual loads send commit bits speculatively to break the serial commit dependence. The processor uses pipeline flushing to rollback on a commit mis-speculation.

This policy thus uses a hybrid of *selective re-execution* for the aggressive

execution of loads and speculation with *flushing* for acceleration of commit bits. To issue the speculative commit bits accurately, we re-employed the dependence predictors evaluated earlier (*all-stores*, *one-store*, and *first-store*).

We show an example in Figure 5.1. If the load is predicted independent, the load sends its commit bit as soon as it receives a commit bit from its address, despite the presence of earlier unresolved or uncommitted stores. If a conflict is later detected, the pipeline must be flushed to guarantee correct execution. If the load is predicted to be dependent, then the load will send its commit bit only after all prior stores receive their commit bit.

We measured the performance of speculative commit slicing using both dependence prediction strategies, shown in Columns 4 and 5 of Table 5.2. Using the simpler *all-stores* predictor to perform commit slicing provides a 30% speedup over using it to perform speculative load issue. It also provides a 14% speedup over pure dependence prediction using the more complex *one-store* predictor. Using the *one-store* predictor to do commit slicing, however, provides a smaller 17% speedup over using it for load speculation. Commit slicing with DSRE is faster than using dependence prediction for loads on every benchmark we measured. Commit slicing provides a larger speedup for the *all-stores* predictor, achieving close to the performance of the more complex *one-store* predictor with commit slicing. This larger speedup is because the *all-stores* predictor is more conservative, and hence predicts a larger fraction of the loads as conflicting. This class of loads benefit greatly with DSRE, because only commit bits need to be sent for these loads when they resolve. Thus,

	Flush on load mis-speculation		Flush on commit mis-speculation		
Benchmark	**all-stores (IPC)**	**one-store (IPC)**	**all-stores (IPC)**	**one-store (IPC)**	**oracle (IPC)**
ammp	2.41	3.11	3.27	3.84	3.96
art	3.72	3.50	3.72	3.72	3.73
bzip2	3.16	3.23	3.19	3.19	3.24
compress	1.56	1.56	1.65	1.64	1.66
equake	1.71	1.71	1.74	1.74	1.75
m88ksim	0.93	1.28	1.15	1.40	2.31
mcf	0.87	0.83	0.88	0.85	0.88
mgrid	1.31	1.56	3.52	3.36	4.23
mpeg2encode	3.43	3.32	3.49	3.46	3.51
parser	1.31	1.31	1.31	1.31	1.32
twolf	1.27	1.36	1.72	1.63	2.09
hydro2d	1.03	1.73	2.87	2.94	3.35
tomcatv	4.96	4.95	4.96	4.95	4.96
turb3d	0.62	0.74	0.91	1.00	3.85
Mean	1.42	1.61	1.84	1.88	2.30

Table 5.2: IPC with commit slicing on the GPA simulator

DSRE coupled with a simple predictor can be used to achieve performance comparable to that with a more complex predictor.

Performance for most benchmarks with commit slicing using the *one-store* predictor approaches that using oracle. Four benchmarks, *m88ksim, mgrid, twolf*, and *turb3d* still have considerable room for improvement. These benchmarks incur a large number of flushes due to load-store mispredictions. Commit slicing using the conservative *all-stores* predictor achieves better performance for *mgrid* and *twolf* due to fewer flushes.

Table 5.3 shows the performance with commit slicing on the TRIPS simulator for the set of EEMBC benchmarks. Selective re-execution, with commit slicing using the *all-stores* predictor, outperforms dependence prediction using the *all-stores* predictor by 5.6%. Selective re-execution, with commit slicing using the *first-store* predictor, outperforms dependence prediction using the *first-store* predictor by 4.2%. As explained in Chapter 4, the EEMBC benchmarks, have a large number stores to the same address in the instruction window. The large number of stores to the same address results in the *first-store* predictor becoming conservative and predicting a larger number of loads as *conflicting-all-stores*. These loads benefit greatly from selective re-execution, if they happen to get their speculative value from the most recent matching store, before the load.

conven00, ospf, and *tblook01* show poor performance across all the policies. These benchmarks have small average hyperblocks resulting in a large block fetch and commit overhead. *autocor00, canrdr01, fft00* and *pktflow* show similar performance across the different load-store policies. These benchmarks have few load-store dependences and the dependence predictor is able to predict the load-store dependences correctly.

	Flush on load mis-speculation		Flush on commit mis-speculation		
Benchmark	**all-stores (IPC)**	**one-store (IPC)**	**all-stores (IPC)**	**one-store (IPC)**	**oracle (IPC)**
a2time01	0.793	0.842	1.043	1.797	2.418
aifftr01	0.714	0.715	0.790	0.752	2.477
aifirf01	1.692	1.675	1.681	1.734	2.635
aiifft01	0.676	0.699	0.761	0.714	2.592
autcor00	1.208	1.208	1.208	1.208	1.210
basefp01	1.068	1.072	1.033	1.204	1.212
bezier01	2.789	2.793	2.795	2.784	2.789
bitmnp01	0.920	0.965	0.978	0.896	1.714
cacheb01	0.861	0.992	0.880	1.002	1.535
canrdr01	1.400	1.430	1.433	1.423	1.483
conven00	0.538	0.538	0.538	0.538	0.538
fft00	2.725	2.726	2.726	2.726	2.727
idctrn01	1.566	1.532	1.322	1.466	2.719
iirflt01	0.849	0.877	1.147	1.025	1.944
ospf	0.906	0.908	0.908	0.911	0.917
pntrch01	0.930	0.916	1.064	0.901	1.039
pktflow	1.187	1.188	1.209	1.272	1.272
puwmod01	0.922	0.913	0.879	0.992	2.191
routelookup	0.573	0.573	0.573	0.573	0.573
rspeed01	0.887	0.889	0.997	0.906	2.129
tblook01	0.821	0.825	0.845	0.825	0.854
ttsprk01	0.743	0.748	0.753	0.746	0.782
viterb00	1.490	2.217	1.916	1.786	3.053
Mean	0.955	0.979	1.009	1.020	1.361

Table 5.3: IPC with commit slicing on the TRIPS prototype simulator

pntrch01 and *bezier* have marginally lower performance with the oracle policy when compared to commit slicing with the *all-stores* policy. The oracle

policy in the TRIPS simulator uses a functional emulator to identify load-store dependences in hyperblocks. The emulator does not identify these dependences for blocks that are executed down the wrong control path. Hence, without perfect branch prediction, the oracle policy can incur load violation flushes in blocks that are mispredicted by the branch predictor. *pntrch01* and *bezier* have a number of load-violations in these mispredicted paths, resulting in lower performance than commit slicing.

a2time01 shows a large improvement in performance with commit slicing using the *first-store* predictor. *a2time01* incurs fewer load violation flushes with commit slicing using the *first-store* predictor. The *first-store* predictor we used in these experiments uses a table of 3-bit saturating counters to make predictions. The details of the predictor are explained in Table 7.6 in Chapter 7. The 3-bit predictor has more states that we can use for making a prediction with commit slicing. However, it has a longer training time for loads that conflict with multiple prior stores, and go to a different data tile during each dynamic execution of the load. These loads incur a greater number of flushes with dependence prediction using the *first-store* predictor when compared to commit slicing using the *first-store* predictor.

Dependence prediction outperforms commit slicing for *viterb00*, even though it incurs a larger number of flushes. Flushes have a smaller effect on the performance than commit bit propagation delay in *viterb00*. The blocks that get flushed due to load violations are normally younger, speculative blocks. Because the TRIPS processor uses a rolling flush model, load violations that

happen early in younger blocks do not cause an appreciable drop in performance. The load that caused the violation is handled correctly when the block is re-mapped after the flush.

viterb00 operates on 16-bit words. The benchmark stores branch metrics using a pointer to an array of 16-bit words. These words are then loaded and processed. The conflicting loads in this benchmark are found at the beginning of the block. Also, these loads do not always conflict with earlier stores for every dynamic execution. Dependence prediction, using the *first-store* predictor results in a large number of these loads being predicted independent. When the loads conflict with an earlier store, we have a pipeline flush that gets resolved quickly because the loads are at the beginning of the block. When the loads are predicted independent correctly, we get a speedup by not having to wait for all prior stores to resolve.

Commit slicing results in a large fraction of loads being predicted conflicting. These loads send their replies without the commit bit, and send their commit bits only after all prior stores resolve. Even though commit slicing prevents pipeline flushes when the loads conflict, it also results in unnecessary delay for loads that are independent. This delay outweighs the performance benefits of not flushing the pipeline and results in the poor performance.

There is still a 33% difference in performance between DSRE with commit slicing and the oracle policy. This difference is most pronounced in *aifftr01, aiifft01, bitmnp01, cacheb01, idctrn01, iirflt01, matrix01*, and *puwmod01*. To understand the reason for this performance difference, we looked at one of

```
for( i_1 = j_1 ; i_1 < NUM_POINTS ;
     i_1 += n1_1, passCount_1++ ) {
 realLow_1 = &realData_1[l_1] ;
 /* Scale each stage to prevent overflow *
*realLow_1 >>= STAGE_SCALE_FACTOR ;
 tRealData_1 = *realLow_1 * wReal_1 -
     *imagLow_1 * wImag_1 ;
}
```

Figure 5.2: Piece of source code from the inner loop of *aiifft01* to show store-load-store dependence

---

these benchmarks, *aiifft01*.

aiifft01 is part of the automotive/industrial suite in EEMBC and computes the Inverse Fast Fourier Transform on complex input values stored in real and imaginary arrays. The program is constructed in such a way that in the steady state, there are a number of stores that depend on earlier loads for their data. We saw an example of such behavior in Figure 4.14 in Chapter 4. *aiifft01* also uses the *RAMfilePtr* for storing the output and hence has the same store-load-store dependence seen in Figure 4.14. *aiifft01* also has other variables that result in a store-load-store dependence. Figure 5.2 shows another example code from the main loop of *aiifft01* that results in a store-load-store dependence. The program stores the address of an element of the array, *realData_1*, in the variable *realLow_1*. The array element data is then accessed by using the *realLow_1* variable. The program uses this value to compute the new value of another variable, *tRealData_1*, which is then stored in memory.

Figure 5.3 shows part of the TIL code that corresponds to the C code

```
.bbegin t_run_test$122
   entera $t5, realData_1
   slli $t6, $t3, 3
   add $t7, $t6, $t5
   entera $t8, realLow_1$$7464
   sd ($t8), $t7 S[2]
   entera $t33, realLow_1$$7464
   ld $t34, ($t33) L[17]
   ld $t35, ($t34) L[18]
   entera $t36, imagLow_1$$7465
   ld $t37, ($t36) L[19]
   ld $t38, ($t37) L[20]
   entera $t39, wReal_1$$7452
   ld $t40, ($t39) L[21]
   mul $t41, $t35, $t40
   entera $t42, wImag_1$$7453
   ld $t43, ($t42) L[22]
   mul $t44, $t38, $t43
   sub $t45, $t41, $t44
   entera $t46, tRealData_1$$745
   sd ($t46), $t45 S[23]
```

Figure 5.3: Piece of TIL code from the inner loop of *aiifft01* to show store-load-store dependence

---

in Figure 5.2. From Figure 5.3, we see that the load with LSID 17 needs the value stored by store with LSID 2. This load, in turn, provides the value required for the computation of *tRealData_1*. The value of *tRealData_1* is written to memory by the store with LSID 23. Thus, there is a store-load-store dependence among these three instructions. If the load with LSID 17 is made to wait on all prior stores to resolve before sending the commit bit, it will result in extra delay in the propagation of the commit wave. The address of the store with LSID 2 does not change across different iterations of the loop. Hence, the load with LSID 17 will match with multiple prior stores in the

instruction window. The *first-store* predictor is not able to predict the exact matching store and serializes the load. Similar store-load-store dependence is found in the other benchmarks that show a large difference in performance between commit slicing and oracle.

Thus, the reason for the poor performance of commit slicing on some EEMBC benchmarks is twofold. First, these benchmarks have loads that conflict with multiple prior stores to the same address in the instruction window. The *first-store* predictor is unable to predict the exact store the load conflicts with, and serializes the conflicting load. These loads send their commit bits after all prior stores resolve. Second, these benchmarks have stores that depend on these conflicting loads for their value. Since the conflicting loads can send their commit bits only after all prior stores resolve, commit bit propagation to the stores that depend on the loads is delayed. The delay in the propagation of commit bits to the depending store in turn delays commit bit propagation for serialized loads after the store. Hence, the twin effects of store-to-load and load-to-store dependences results in poor performance in these benchmarks.

There are multiple ways to approach this problem. One approach is to reduce the number of load-to-store dependences in the program. For example, the load-to-store dependence is also exhibited by some static variables, like the iteration counter, in *aiifft01*. This dependence can be eliminated by aggressive compiler optimizations that register allocate static variables. Load-to-store dependences that are a result of the program structure, and hence cannot be eliminated, will result in commit bit propagation delay.

	Flush on load mis-speculation		Flush on commit mis-speculation		
Configuration	**all-stores (cycles)**	**first-store (cycles)**	**all-stores (cycles)**	**first-store (cycles)**	**oracle (cycles)**
Non-optimized	87.5m	84.5m	77.7m	85.3m	22.8m
Optimized	18.7m	18.3m	15.5m	13.2m	10.2m

Table 5.4: Number of cycles (in millions) for program execution for non-optimized and optimized *aiifft01*

	Flush on load mis-speculation		Flush on commit mis-speculation		
Configuration	**all-stores (IPC)**	**first-store (IPC)**	**all-stores (IPC)**	**first-store (IPC)**	**oracle (IPC)**
Non-optimized	0.68	0.70	0.76	0.71	2.60
Optimized	1.45	1.48	1.75	2.06	2.64

Table 5.5: IPC for non-optimized and optimized *aiifft01*

We hand-optimized *aiifft01* and removed some of the redundant loads and stores in the program. Table 5.4 and Table 5.5 compares the performance of the various load-store recovery schemes for both the compiler-generated and the hand-optimized binary. Table 5.4 shows the number of cycles taken by the program for the different load-store dependence prediction schemes while Table 5.5 shows the IPC for the various configurations. We see from Table 5.4 that eliminating redundant loads and stores results in fewer cycles for program execution for all the load-store recovery schemes. Comparing the IPC

of the different load-store dependence prediction schemes for the optimized version of the benchmark, we see that there is still 44% difference in performance between dependence prediction and oracle, due to some stores in the instruction window that are to the same address. DSRE with commit slicing improves performance over dependence prediction by 39%. In the optimized version of the program, DSRE is able to tolerate the extra traffic generated by multiple stores waking up the same load, and yields higher performance. Hence, DSRE has the potential to improve performance significantly with an optimizing compiler.

Bottom-up Commit Traversal, discussed in the next section, is another way to reduce the commit propagation delay. Finally, we can reduce serialization of loads by using more sophisticated commit bit prediction, since only the loads that are serialized by the commit bit predictor encounter this delay. A commit bit predictor, which tries to identify the exact matching store for a load, will result in higher performance. Such a predictor will also result in performance improvement with regular dependence prediction. Store sets [12] and distance-based predictors [73] discussed in Chapter 2 are two such predictors that have been proposed for conventional, superscalar processors. Modifying these predictors to work in the distributed TRIPS environment can be part of future work.

In summary, we saw lower performance gains with DSRE and commit slicing on the TRIPS prototype simulator when compared to the GPA simulator. The lower performance was due to both software (sub-optimal code and

loop-based benchmarks) and hardware (poor dependence prediction and extra ALU and network contention). We showed one example optimized code that showed a large improvement in performance both for the baseline and with DSRE. For programs with similar behavior, we expect performance gains on the TRIPS simulator to increase and match the GPA simulator with better compiler technology that yields optimized code.

5.1.2 Bottom-up Commit Traversal

If all operations—including loads—could execute in a single cycle, selective re-execution would provide no benefit over conservative load/store ordered execution, because the commit DFG traversal would take the same time as the execution traversal. DSRE improves performance because not all operations require a single cycle, especially cache misses, so the commit traversal can catch up to the execution traversal while long-latency operations on the critical path execute. However, since no execution actually occurs on the commit wave, it may be possible for the commit wave to skip nodes in the graph, thus completing more quickly.

Speculative commit slicing essentially removes some arcs from the commit traversal graph speculatively, allowing more of the graph to be traversed in parallel and speeding up the traversal. An alternate approach is to allow commit bits to skip over nodes, going directly from the input to the output of a multi-instruction dependence chain without traversing the intermediate nodes. If the root of a dependence tree has only one speculative input, then

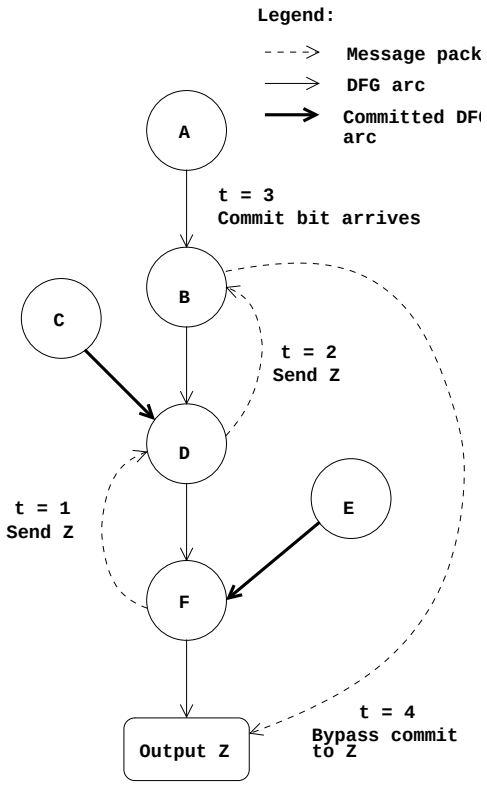

Figure 5.4: Bottom-up commit traversal

the intermediate nodes in the tree can be bypassed when the last committed operand arrives, by sending the commit bit directly to the leaves, provided no execution is still in flight.

Bottom-Up Commit Traversal selectively allows a partial bottom-up traversal to support forwarding of commit bits over multi-hop chains. If a leaf node—in this case an output-producing instruction—of the DFG has only one speculative parent (all other parents, if any, have sent their commit bits), then the output node forwards its target(s) to the one speculative parent. The

output node knows the parent's reservation station address since it has already received an operand from that parent, assuming the address was buffered. When the parent generates a commit bit, it bypasses the intermediate node and sends the commit bit directly to the output, as shown in Figure 5.4. Prior to committing, however, if the parent has only one speculative parent, it too can forward the output address to its parent (the grandparent), which can then either do the same thing (forward up the chain if it has one speculative parent) or send the commit bit to the output, bypassing two nodes. If an instruction holding a bypass target re-fires instead of generating a null commit message, then the bypass chain is discarded and the new operand is forwarded to the children as in the base architecture. When the execution reaches the outputs, they can begin the process of rebuilding the bypassing links anew.

Table 5.6 shows the performance of this bottom-up traversal scheme when combined with speculative slicing. The bottom-up traversal scheme performs marginally better than commit slicing on most benchmarks. The bottom-up scheme will not provide any benefit for instructions that have multiple speculative inputs, since these instructions cannot propagate the bypass information. Also, if the commit bit is sent by the parent before it has received the bypass information from its children, the bypass message does not result in significant speedup. The bottom-up scheme provides the most benefit when the bypass message reaches the root node of a speculative chain, before the root node sends it commit bit.

The bottom-up commit scheme has significant implementation com-

Benchmark	Commit slicing (IPC)	Bottom-up (IPC)	Oracle (IPC)
ammp	3.84	3.86	3.96
bzip2	3.19	3.19	3.24
compress	1.64	1.64	1.66
equake	1.74	1.73	1.75
m88ksim	1.40	1.45	2.31
mcf	0.85	0.85	0.88
mgrid	3.36	3.40	4.23
mpeg2encode	3.46	3.47	3.51
parser	1.31	1.31	1.32
twolf	1.63	1.62	2.09
hydro2d	2.94	2.95	3.35
tomcatv	4.95	4.94	4.96
turb3d	1.00	1.01	3.85
Mean	1.81	1.83	2.23

Table 5.6: IPC with commit bypass on the GPA simulator

plexity. Bypass messages need to carry the version number of the result that the bypass requesting node is expecting. Also, we assumed that a node can send bypass messages to all the nodes that request a bypass message. In a real implementation, a node will be able to send bypass messages to only one or two children, thus limiting the performance gains due to the bottom-up scheme.

m88ksim shows the most improvement in performance with the bottom-up policy. This benchmark incurs fewer flushes with the bottom-up scheme. Thus, the improvement in performance can be attributed to the indirect in-

fluence of interaction between the bottom-up scheme and the dependence predictor training. In summary, the mean performance of bottom-up traversal scheme is marginally better than speculative slicing with the *one-store* policy. However, the bottom-up traversal scheme incurs significant hardware complexity over the base DSRE scheme, and is not worth the marginal performance improvement.

5.2 Optimal Maximum Version Number

To find the impact on performance for various maximum values of version number with commit slicing, we ran experiments on the GPA simulator varying the number of times ALUS are allowed to fire speculatively. Table 5.7 shows the performance for the various benchmarks when we vary the maximum number of speculative firings. We used the *one-store* dependence predictor for commit slicing. Loads that are predicted dependent by the predictor send speculative replies when they arrive at the memory interface. Later arriving stores that match the address of the loads, and are earlier in program order, are allowed to wakeup these loads. The loads send their commit bits only after all previous stores have resolved.

From Table 5.7 we see that for most benchmarks, there is no significant difference in performance when we increase the maximum version number. This list of benchmarks includes *ammp, art, bzip2, compress, equake, mcf, mpeg2encode, parser,* and *tomcatv.* This result is similar to what we observed in Chapter 4. For these benchmarks, the number of speculative firings does

Maximum speculative execution	**1**	**2**	**3**	**4**	**5**	**6**
ammp	3.84	3.82	3.82	3.82	3.82	3.82
art	3.85	3.85	3.85	3.85	3.85	3.85
bzip2	3.19	3.19	3.19	3.19	3.19	3.19
compress	1.64	1.64	1.64	1.64	1.64	1.64
equake	1.74	1.73	1.73	1.73	1.73	1.73
m88ksim	1.40	1.37	1.37	1.38	1.36	1.32
mcf	0.85	0.85	0.85	0.85	0.85	0.85
mgrid	3.36	2.97	3.17	3.17	3.19	3.17
mpeg2encode	3.46	3.47	3.48	3.47	3.47	3.47
parser	1.31	1.31	1.31	1.31	1.31	1.31
twolf	1.63	1.62	1.62	1.61	1.61	1.60
hydro2d	2.94	2.80	2.70	2.61	2.38	2.26
tomcatv	4.95	4.95	4.95	4.95	4.95	4.95
turb3d	1.00	1.03	1.03	1.04	1.04	1.03
Mean	1.88	1.88	1.87	1.88	1.86	1.85

Table 5.7: Commit slicing IPC variation with increasing maximum speculative firing on the GPA simulator

not change significantly when we increase the maximum version number allowed. In these benchmarks, the dependence predictor is able to predict a large percentage of loads correctly. Also, as seen in Chapter 4, these benchmarks do not have a large number of load-store dependences. *ammp* and *compress* have a higher percentage of loads that depend on earlier stores. However, this dependence is satisfied by the first matching store in these benchmarks, thus resulting in fewer refirings.

m88ksim, mgrid, twolf, hydro2d, and *twolf* show a larger variation in performance with maximum allowed version number. Performance of *m88ksim,*

mgrid, twolf, and *hydro2d* decreases as we increase the maximum allowed version number. The number of times instructions that fire speculatively increases in these benchmarks as we increase the maximum version number allowed. These benchmarks have a higher percentage of loads that depend on earlier stores. Also, a large number of loads in these benchmarks are incorrectly predicted as conflicting, and multiple versions of these loads execute in these benchmarks. These loads send multiple speculative values that result in extra ALU and network contention.

The only benchmark that shows increase in performance with larger version numbers is *turb3d*. In this benchmark, increasing the maximum allowed version number decreases the number of load mispredictions, as the predictor becomes more conservative and predicts a larger number of loads as conflicting. These loads do result in more refirings. However, the performance benefit of fewer flushes outweighs the reduction in performance due to the extra contention. In summary, having ALUs fire speculatively no more than once yields the best performance benefits with commit slicing on the GPA simulator.

Table 5.8 shows the performance with commit slicing, using a the 3-bit *first-store* predictor, across the set of EEMBC benchmarks on the TRIPS simulator. From Table 5.8, we see there is less variation in performance on the TRIPS simulator when we vary the maximum version number. Loads that are predicted *non-conflicting* or *conflicting-all-stores* do not send speculative replies. Hence, these loads are not affected by the variation in maximum ver-

sion number. On the TRIPS simulator, performance drops when we increase the number of maximum speculative execution allowed beyond three.

Maximum speculative firing	**1**	**2**	**3**	**4**	**5**	**6**
a2time01	1.786	1.796	1.797	1.797	1.797	1.797
aifftr01	0.756	0.751	0.740	0.752	0.750	0.753
aifirf01	1.740	1.735	1.732	1.734	1.734	1.734
aiifft01	0.698	0.722	0.707	0.716	0.717	0.716
autcor00	1.208	1.208	1.208	1.208	1.208	1.208
basefp01	1.202	1.204	1.204	1.204	1.204	1.204
bezier01	2.784	2.784	2.784	2.784	2.789	2.784
bitmnp01	0.886	0.889	0.889	0.896	0.892	0.890
cacheb01	1.006	1.002	1.002	1.002	1.002	1.002
canrdr01	1.423	1.423	1.423	1.423	1.423	1.423
conven00	0.538	0.538	0.538	0.538	0.538	0.538
fft00	2.726	2.726	2.726	2.726	2.726	2.726
idctrn01	1.455	1.462	1.467	1.466	1.464	1.460
iirflt01	0.995	1.019	1.002	1.025	1.019	1.017
ospf	0.911	0.911	0.911	0.911	0.911	0.911
pntrch01	0.898	0.901	0.904	0.901	0.901	0.901
pktflow	1.273	1.272	1.272	1.272	1.272	1.272
puwmod01	1.013	0.992	0.992	0.992	0.992	0.992
routelookup	0.573	0.573	0.573	0.573	0.573	0.573
rspeed01	0.905	0.906	0.906	0.906	0.906	0.906
tblook01	0.825	0.825	0.825	0.825	0.825	0.825
ttsprk01	0.749	0.746	0.746	0.746	0.746	0.746
viterb00	1.764	1.782	1.786	1.786	1.786	1.786
Mean	1.018	1.020	1.017	1.020	1.020	1.020

Table 5.8: Commit slicing IPC variation with increasing maximum speculative firing on the TRIPS simulator

5.3 Performance Studies with Commit Slicing

In this section, we look at the performance of selective re-execution, with commit slicing, for perfect branch prediction, perfect L1 data cache, and perfect L2 cache. We also examine the scalability of DSRE by increasing the instruction window size from 1K to 4K instructions.

5.3.1 Performance with Perfect Branch Prediction

	Flush on load mis-speculation		Flush on commit mis-speculation		
Benchmark	**all-stores (IPC)**	**one-store (IPC)**	**all-stores (IPC)**	**one-store (IPC)**	**oracle (IPC)**
ammp	2.41	3.22	3.01	4.11	4.23
art	3.89	3.15	3.89	3.87	3.89
bzip2	2.51	4.73	2.97	4.78	5.31
compress	2.36	2.39	2.46	2.45	2.42
equake	1.95	1.94	1.98	1.98	1.99
m88ksim	0.95	1.53	1.20	1.71	2.44
mcf	1.07	1.03	1.08	1.07	1.09
mgrid	1.46	1.55	3.46	3.50	4.31
mpeg2encode	3.93	3.93	4.02	4.02	4.05
parser	1.43	1.43	1.43	1.44	1.44
twolf	1.33	1.49	2.14	2.15	2.79
hydro2d	1.08	2.12	2.98	3.09	3.42
tomcatv	5.12	5.10	5.12	5.11	5.12
turb3d	0.66	0.81	0.96	1.12	4.20
Mean	1.55	1.85	2.04	2.24	2.70

Table 5.9: IPC with commit acceleration on the GPA simulator with perfect branch prediction

Table 5.9 compares performance of commit slicing against dependence prediction with perfect branch prediction on the GPA simulator. Perfect branch prediction improves performance by 9% and 15% with dependence prediction using the *all-stores* and *one-store* predictor. With commit slicing, the performance improvements are 11% and 19% respectively, for *all-stores* and *one-store*. The oracle policy shows a performance improvement of 17.4%. Perfect prediction increases the number of loads and stores in the instruction window, thus reducing the accuracy of the dependence predictor. Selective re-execution with commit slicing provides a larger improvement in performance than dependence prediction by improving performance of loads that are incorrectly predicted conflicting.

Table 5.10 compares the performance of commit slicing against dependence prediction with perfect branch prediction on the TRIPS simulator. Perfect branch prediction improves performance by 4.8% and 4.6% with dependence prediction using the *all-stores* and *first-store* predictor. DSRE with commit slicing shows a reduction in performance with perfect branch prediction similar to what we saw in Chapter 4. Again, this reduction in performance is due to multiple matching stores to the same address, which results in extra network and ALU contention. The oracle policy shows a performance improvement of 7%. We see from Table 5.10 that perfect branch prediction does not result in any appreciable increase in performance due to the small size of the EEMBC benchmarks.

	Flush on load mis-speculation		Flush on commit mis-speculation		
Benchmark	**all-stores (IPC)**	**one-store (IPC)**	**all-stores (IPC)**	**one-store (IPC)**	**oracle (IPC)**
a2time01	0.798	0.846	1.043	1.814	2.447
aifftr01	0.700	0.713	0.790	0.604	2.499
aifirf01	1.717	1.697	1.681	1.732	2.588
aiifft01	0.673	0.735	0.761	0.722	2.602
autcor00	1.465	1.464	1.208	1.464	1.464
basefp01	1.110	1.116	1.033	1.234	1.262
bezier01	2.783	2.776	2.795	2.778	2.789
bitmnp01	1.008	1.061	0.978	0.975	1.975
cacheb01	0.951	0.952	0.880	1.192	2.048
canrdr01	1.652	1.725	1.433	1.680	1.852
conven00	0.510	0.481	0.538	0.481	0.483
fft00	2.713	2.659	2.726	2.659	2.660
idctrn01	1.583	1.573	1.322	1.365	2.862
iirflt01	0.841	0.863	1.147	1.053	1.969
ospf	1.053	1.054	0.908	0.529	1.070
pntrch01	0.927	0.967	1.064	0.904	1.048
pktflow	1.269	1.271	1.209	1.367	1.368
puwmod01	0.928	0.919	0.879	1.073	2.268
routelookup	0.719	0.719	0.573	0.719	0.719
rspeed01	0.888	0.890	0.997	0.908	2.172
tblook01	0.864	0.899	0.845	0.868	0.886
ttsprk01	0.804	0.810	0.753	0.813	0.862
viterb00	1.491	2.226	1.916	1.791	3.046
Mean	1.001	1.024	1.009	1.013	1.457

Table 5.10: IPC with commit slicing on the TRIPS prototype simulator with perfect branch prediction

5.3.2 Performance with Perfect L1 Data Cache

	Flush on load mis-speculation		Flush on commit mis-speculation		
Benchmark	**all-stores (IPC)**	**one-store (IPC)**	**all-stores (IPC)**	**one-store (IPC)**	**oracle (IPC)**
a2time01	0.798	0.846	1.043	1.814	2.447
ammp	2.59	3.22	3.45	4.03	4.26
art	4.09	3.69	4.09	4.08	4.10
bzip2	3.72	3.88	3.81	3.88	3.90
compress	1.81	1.78	1.91	1.83	1.84
equake	2.50	2.51	2.56	2.56	2.58
m88ksim	0.93	1.24	1.15	1.40	2.32
mcf	1.15	1.11	1.17	1.15	1.17
mgrid	1.53	2.07	3.54	3.52	5.58
mpeg2encode	3.43	3.42	3.50	3.48	3.52
parser	1.39	1.39	1.39	1.39	1.40
twolf	1.30	1.37	1.75	1.69	2.22
hydro2d	1.46	2.62	3.08	3.27	4.11
tomcatv	7.80	7.70	7.80	7.79	7.80
turb3d	0.67	0.79	0.92	1.02	3.96
Mean	1.64	1.87	2.05	2.13	2.69

Table 5.11: IPC with commit acceleration on the GPA simulator with perfect L1 D-cache

---

Table 5.11 compares performance of commit slicing against dependence prediction with perfect L1 data cache on the GPA simulator. Perfect L1 data cache improves performance by 15.5% and 16% with dependence prediction using the *all-stores* and *one-store* predictor. With commit slicing, the performance improvements are 11% and 13% respectively for *all-stores* and *one-store*. The oracle policy shows a performance improvement of 17%. The lower im-

provement in performance with selective re-execution and commit slicing is in line with the results seen in Chapter 4.

Table 5.11 compares performance of commit slicing against dependence prediction with perfect L1 data cache on the TRIPS simulator. The performance improvements are not as large with perfect L1 D-cache on the TRIPS simulator because of the small memory footprint of the EEMBC benchmarks.

	Flush on load mis-speculation		Flush on commit mis-speculation		
Benchmark	**all-stores (IPC)**	**one-store (IPC)**	**all-stores (IPC)**	**one-store (IPC)**	**oracle (IPC)**
a2time01	0.798	0.846	1.043	1.814	2.447
a2time01	0.791	0.842	1.043	1.798	2.422
aifftr01	0.719	0.720	0.791	0.756	2.495
aifirf01	1.693	1.676	1.683	1.736	2.614
aiifft01	0.681	0.717	0.763	0.721	2.620
autcor00	1.171	1.171	1.171	1.171	1.172
basefp01	1.071	1.075	1.032	1.213	1.225
bezier01	2.792	2.439	2.797	2.795	2.788
bitmnp01	0.922	0.965	0.978	0.890	1.716
cacheb01	0.881	1.016	0.885	1.005	1.597
canrdr01	1.401	1.434	1.434	1.425	1.486
conven00	0.538	0.538	0.539	0.538	0.538
fft00	2.734	2.734	2.734	2.734	2.736
idctrn01	1.549	1.525	1.314	1.469	2.729
iirflt01	0.836	0.849	1.147	1.022	1.951
ospf	0.941	0.942	0.935	0.940	0.945
pntrch01	0.931	0.913	1.064	0.903	1.041
pktflow	1.262	1.263	1.282	1.336	1.336
puwmod01	0.925	0.915	0.879	0.996	2.194
routelookup	0.573	0.573	0.573	0.573	0.573
rspeed01	0.888	0.891	0.999	0.908	2.142
tblook01	0.822	0.827	0.846	0.835	0.856
ttsprk01	0.744	0.749	0.754	0.748	0.783
viterb00	1.490	2.213	2.204	1.798	3.054
Mean	0.959	0.981	1.016	1.025	1.369

Table 5.12: IPC with commit slicing on the TRIPS prototype simulator with L1 D-cache

	Flush on load mis-speculation		Flush on commit mis-speculation		
Benchmark	**all-stores (IPC)**	**one-store (IPC)**	**all-stores (IPC)**	**one-store (IPC)**	**oracle (IPC)**
a2time01	0.798	0.846	1.043	1.814	2.447
ammp	2.52	2.67	3.31	3.60	4.14
art	4.09	3.68	4.09	4.08	4.09
bzip2	3.45	3.60	3.55	3.61	3.62
compress	1.58	1.56	1.65	1.61	1.60
equake	2.41	2.46	2.53	2.53	2.56
m88ksim	0.92	1.27	1.15	1.43	2.32
mcf	1.14	1.08	1.15	1.13	1.16
mgrid	1.53	1.90	3.54	3.49	5.57
mpeg2encode	3.43	3.42	3.50	3.48	3.52
parser	1.32	1.31	1.32	1.32	1.32
twolf	1.24	1.32	1.66	1.61	2.09
hydro2d	1.43	2.71	3.08	3.20	4.11
tomcatv	7.81	7.70	7.81	7.78	7.81
turb3d	0.66	0.77	0.92	1.03	3.95
Mean	1.59	1.79	1.99	2.07	2.59

Table 5.13: IPC with commit acceleration on the GPA simulator with perfect L2 cache

5.3.3 Performance with Perfect L2 cache

Table 5.13 compares performance of commit slicing against dependence prediction with perfect L2 cache on the GPA simulator. Perfect L2 cache improves performance by 12% and 11% with dependence prediction using the *all-stores* and *one-store* predictor. With commit slicing, the performance improvements are 8.1% and 10.2% respectively for *all-stores* and *one-store*. The oracle policy shows a performance improvement of 12.6%. The lower improve-

ment in performance with selective re-execution and commit slicing is again in line with the results seen in Chapter 4.

Table 5.14 compares performance of commit slicing against dependence prediction with perfect L2 cache on the TRIPS simulator. The performance trend is similar to what we saw with DSRE in Chapter 4. Performance improvements with perfect L2 cache are higher than perfect L1 D-cache, due to the lower instruction cache miss latency.

	Flush on load mis-speculation		Flush on commit mis-speculation		
Benchmark	**all-stores (IPC)**	**one-store (IPC)**	**all-stores (IPC)**	**one-store (IPC)**	**oracle (IPC)**
a2time01	0.798	0.846	1.043	1.814	2.447
a2time01	0.795	0.844	1.045	1.805	2.434
aifftr01	0.715	0.715	0.790	0.745	2.479
aifirf01	1.703	1.687	1.693	1.744	2.639
aiifft01	0.676	0.712	0.761	0.716	2.602
autcor00	1.238	1.238	1.238	1.238	1.239
basefp01	1.087	1.091	1.049	1.231	1.243
bezier01	2.792	2.353	2.798	2.794	2.794
bitmnp01	0.923	0.969	0.980	0.896	1.723
cacheb01	0.866	1.000	0.886	1.010	1.555
canrdr01	1.414	1.447	1.447	1.438	1.500
conven00	0.540	0.540	0.540	0.539	0.539
fft00	2.732	2.732	2.732	2.732	2.733
idctrn01	1.535	1.544	1.338	1.478	2.764
iirflt01	0.853	0.869	1.154	1.058	1.987
ospf	0.909	0.910	0.911	0.913	0.919
pntrch01	0.932	0.919	1.066	0.907	1.043
pktflow	1.197	1.197	1.220	1.285	1.284
puwmod01	0.925	0.917	0.881	1.019	2.212
routelookup	0.573	0.573	0.573	0.573	0.573
rspeed01	0.892	0.894	1.002	0.911	2.164
tblook01	0.826	0.830	0.849	0.831	0.859
ttsprk01	0.746	0.751	0.755	0.750	0.785
viterb00	1.490	2.222	1.948	1.762	3.061
Mean	0.959	0.981	1.015	1.027	1.371

Table 5.14: IPC with commit slicing on the TRIPS prototype simulator with perfect L2 cache

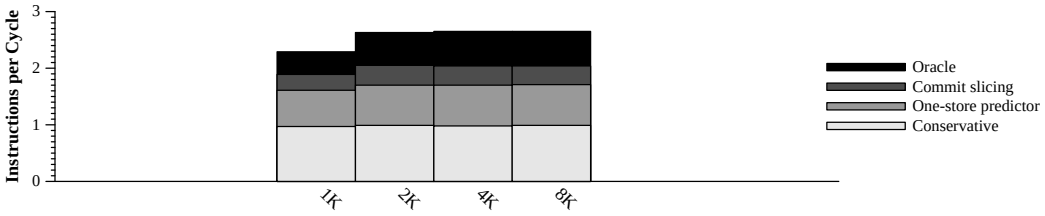

Figure 5.5: DSRE performance with larger instruction window

5.3.4 Performance with a Larger Instruction Window

The selective re-execution mechanism presented in this dissertation uses local state present in the ALUs for recovery. Since the mechanism does not use any centralized structures for recovery, it can be easily extended to future machines with a larger instruction window.

Figure 5.5 shows the IPC for various load issue policies when we increase the size of the instruction window in the processor from 1K to 4K instructions. When the window size is doubled, the performance improves by a mere 2% with a conservative load-issue policy, demonstrating the well-known result that load speculation is necessary to exploit large-window ILP. The oracle policy improves by 26%, showing the potential performance advantages of scaling the window size. Conventional dependence prediction (with flushing) improves by just over half of the ideal, increasing by 14% using the *one-store* policy. The SRE implementations scale better with increasing window size; *one-store* policy with DSRE results in a 25.5% improvement in performance, and *all-stores* improves by 27%. DSRE with commit slicing thus scales similarly in performance to the oracle as the window size is doubled. Beyond an instruction

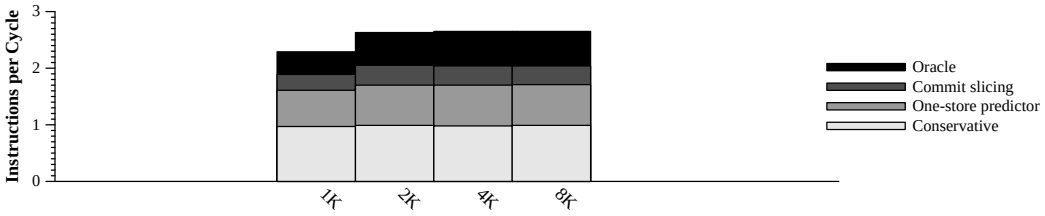

Figure 5.6: DSRE performance with larger instruction window and perfect prediction

window of 2K instructions, performance saturates for all the load issue policies.

Figure 5.6 shows the mean performance of various load issue schemes for various instruction window sizes with perfect branch prediction. Surprisingly, *oracle* is the only policy that shows an increase in performance as the window size is increased beyond 2K. For the oracle policy, the performance increase is the highest when we go from 1K to 2K window and reduces after that. The conservative policy shows no difference in performance when we increase the window size. Commit slicing and dependence prediction show an increase in performance when we go from a 1K instruction window to a 2K instruction window, and show a decrease in performance beyond 2K. We analyzed the benchmarks to find the reason for this behavior and found that the dependence predictor becomes increasingly conservative for larger instruction windows with more conflicting loads and stores. Hence, loads are held back unnecessarily, reducing performance with dependence prediction. Selective re-execution, with commit slicing, results in these these loads generating a large number of speculative executions, thus reducing performance. With

instruction windows larger than 2K instructions, we will need to use better dependence predictors with selective re-execution to improve performance.

In this chapter, we looked at two techniques—*speculative commit slicing* and *bottom-up commit traversal*—to accelerate commit bit propagation with load-store dependence prediction. Although we focused on load-store dependence prediction in this chapter, recovery using DSRE is not limited to load-store dependence prediction. DSRE is designed as a recovery mechanism that can be used by any data speculation engine for low-cost recovery. In the next chapter, we present a brief evaluation of another data speculation mechanism, last-value prediction, and show how multiple speculation engines can work concurrently with DSRE. We also discuss other potential uses for DSRE in the next chapter.

Chapter 6

DSRE Applications

In this chapter, we present a brief evaluation of another data speculation mechanism, last-value prediction, to show how DSRE can work concurrently with multiple data speculation engines. We also discuss how DSRE can be extended to to save energy and provide better reliability in future processors.

6.1 DSRE and Last-Value Prediction

The last few sections have focused on using DSRE to improve the performance for load-store dependence prediction. The selective re-execution mechanism presented in this dissertation is intended to be independent of the underlying data speculation mechanism, and can be theoretically used by multiple disparate speculation engines for recovery. In this section, we evaluate the another data speculation mechanism, last value speculation, and show how DSRE can be used for recovery with multiple speculation engines.

6.1.1 Potential for Last-Value Prediction

Data value locality and reuse is a phenomenon which has recently generated considerable interest in the computer architecture community [4, 5, 19, 37, 39, 42, 55, 69]. Data value reuse results when an instruction produces the same result during different dynamic invocations. A high data value reuse will result in greater performance improvement with data value speculation.

Value locality was first defined by Lipasti et al. and exploited to perform load value prediction [40]. Using simple predictors, the authors achieve 3% and 6% average improvement in performance on processors modeling the PowerPC 620 and Alpha 21164. Value locality and reuse was subsequently extended in a number of directions. Yang and Gupta investigated value locality of load instructions to eliminate redundancy [70]. The value locality of store instructions has been studied in an effort to reduce multiprocessor data and address bus traffic [37]. Also, researchers have proposed a number of predictors in literature for predicting values of instructions [61, 68]. Researchers have also examined compiler optimizations for increasing value reuse [6]. Other work in value prediction has shown that considerable instruction fetch bandwidth is needed to speculate on values effectively [19], which is not an issue in this context because of the high instruction fetch bandwidth provided by the TRIPS architecture.

To investigate the potential for data value reuse in SPEC CPU2000 programs, we modified the sim-alpha simulator to count reuse for each dynamic instruction executed [14]. We used SimPoint simulations and simulated 100

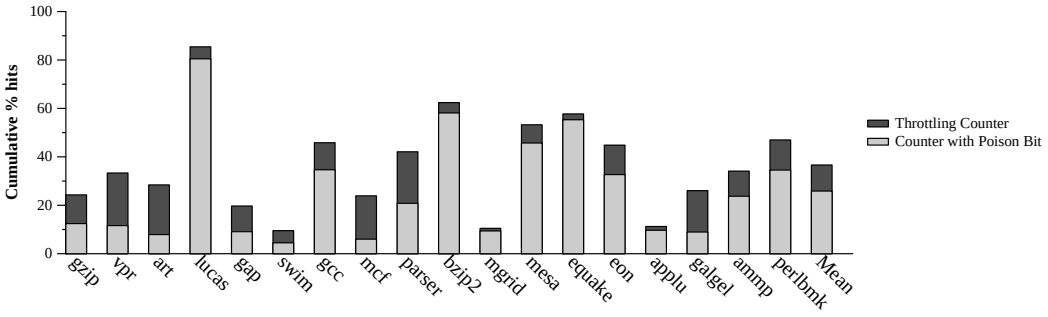

Figure 6.1: Correct value speculations with throttling counter and poison bit

million instructions for each benchmark [62]. We associated 1, 2, and 3-bit saturating counters with each static instruction. Other researchers have used similar confidence estimators in earlier work to increase the accuracy of their predictors [4, 40, 55, 68].

We incremented the counter when an instruction's result matched its previous result and decremented the counter when it did not. Figure 6.1 shows the percentage of retired instructions that produced the same result during successive dynamic invocations, for the highest value of the counter associated with the instruction. For brevity, we show the results for only the 2-bit counter in this section.

From Figure 6.1 we see that on an average more than 36% of the instructions committed produced the same result in at least four successive dynamic invocations. Thus, there is tremendous reuse in the SPEC CPU2000 suite, which suggests that aggressive data value speculation techniques, along with low cost recovery, have potential for performance improvement.

To reduce data value mis-speculation, we associated a poison bit with each static instruction, which is set for an instruction if we mis-speculate, for the duration of the simulation. Other related work in value speculation has examined throttling value speculation of instructions that have low confidence, which has a goal similar to the saturating counter and poison bits that we employ [8]. We throttle data value speculation for instructions whose poison bit is set. We found that even with a poison bit, 26% of the instructions on an average reuse their results.

6.1.2 Recovery with DSRE for Last-Value Prediction

To investigate the effectiveness of the decentralized last-value prediction, we implemented a simple last-value predictor in the GPA simulator. The last-value predictor is indexed using the instruction address and stores the last value produced by the instruction. We associated a 2-bit counter with each entry. We increment the counter every time an instruction produced the same result and decrement it otherwise. We replace the value associated with an instruction when the high bit of the counter is zero and the counter is reset on a replacement. An ALU speculates on an instruction's result when the high bit of the counter associated with the instruction is 1. We also associated a poison bit with each instruction that is set whenever an instruction mis-speculates. Speculation is throttled for instructions whose poison bit is set. We simulated a set of benchmarks from the SPEC CPU95 suite, the SPEC CPU2000 suite, and the MediaBench suite. Decentralized last-value prediction was applied to

Benchmark	Base IPC	Speedup - 2-bit Counter	Speedup - 2-bit Counter and Poison Bit
adpcm	1.3	7.7%	7.7%
art	4.3	-7.0%	4.6%
bzip2	3.6	5.6%	2.8%
dct	7.6	-1.3%	0.0%
m88ksim	1.7	-11.8%	0.0%
mcf	0.9	25.0%	0.0%
mpeg2encode	3.9	-10.3%	0.0%
parser	1.7	-6.2%	0.0%
twolf	1.7	5.9%	5.9%

Table 6.1: Last-value prediction performance on the GPA simulator

only integer instructions in the benchmarks.

Table 6.1 lists the performance of the last-value predictor across the set of benchmarks. The first column shows the IPC of the benchmark on the base case without value prediction. The second column shows the speedup obtained when using only the 2-bit counter. We see from Table 6.1 that using the only the 2-bit counter actually hurts the performance on some benchmarks. However, some benchmarks like *adpcm* and *mcf*, show appreciable speedup with the last-value predictor. We found that the low accuracy of the 2-bit counter generates a large number of mis-speculated values in the GPA resulting in ALUs firing multiple times to generate the right value.

The third column in Table 6.1 lists the speedup obtained with the 2-bit counter and poison bit. We see from the table that using a poison bit never

hurts performance. Also, some benchmarks like *adpcm* and *twolf* show significant improvement in performance. We found that using the poison bit reduces both the correct and the incorrect value predictions. However, the reduction in the number of mispredictions is far greater than the reduction in the number of correct predictions, thus resulting in either increased performance or no change in performance.

In this section, we evaluated a simple last-value predictor in this section. DSRE was used to recover when the last-value predictor mis-speculated. We also enabled commit slicing using the *one-store* predictor in these experiments. Because DSRE has been designed to be independent of the data speculation engine, both load-store dependence speculation and last-value prediction used the same DSRE mechanism for recovery. Even though we did not see a significant increase in performance using last-value prediction, this section does demonstrate how various data speculation engines can concurrently use DSRE for recovery from mis-speculations.

6.2 DSRE and Energy

Selective re-execution has the potential to save energy consumption in microprocessors by re-executing only instructions that are part of the data flow graph of a mis-speculating instruction. However, selective re-execution does result in extra null commit messages in the processor. Also, the extra logic required to support selective re-execution will consume static power in smaller technologies, where leakage is an issue. In our studies, we used version numbers

to throttle speculation to improve performance. Using mechanisms to monitor energy usage, we can use version numbers to throttle speculation to conserve energy. Any implementation of selective re-execution, in future technologies with smaller feature sizes, should also consider energy to establish feasibility.

6.3 DSRE for Reliability

Reliability is emerging as an important issue in microprocessor design at smaller feature sizes. A number of recent papers have examined the growing important of dealing with soft errors that tend to increase with decreasing feature size [22, 23, 38, 63]. Soft errors are caused in processors by electrical noise or external radiation. Transistors in smaller feature sizes are increasingly susceptible to errors from cosmic rays.

Architects have responded to the soft error challenge by designing microarchitectures that are fault tolerant. Solutions primarily involve providing temporal or spatial redundancy with low overhead. For example, DIVA is a microarchitecture that uses spatial redundancy to provide reliability [2, 10]. DIVA uses a slow, reliable substrate to validate the computation of a faster unreliable substrate. Architects have also looked at multithreading techniques to provide redundancy in processors [54, 56].

In an effort to achieve maximum performance at the minimum power budget, researchers have also looked at mechanisms for operating with extremely low safety margins. For example, the Razor microarchitecture continuously varies the voltage to achieve low power consumption during exe-

cution [17]. Razor uses a fault detection and recovery system to adjust the optimal operating point. With power becoming a first-order design constraint, we can expect such mechanisms in future processors.

Selective re-execution can be extended to provide a low-cost recovery mechanism for errors in microprocessors. Logic that computes the probability of an error can determine if a commit bit can be sent with the result. If the probability of a fault is higher than a pre-determined threshold, the result can be sent without the commit. The value can then be re-computed to ensure that the operation executed without a fault. This, and other such mechanisms, are promising topics for future research.

Chapter 7

DSRE on the TRIPS Prototype Simulator

DSRE was initially implemented in a high-level research simulator that modeled one particular instantiation of an EDGE architecture. The GPA simulator used in the initial evaluation did not model some of the low-level details found in an implementation. To validate the performance of DSRE on a hardware implementation, we added support in the the TRIPS prototype simulator for selective re-execution. In Chapter 4 and Chapter 5, we presented results from both simulators. This chapter explains the changes that we made to the various tiles in the TRIPS prototype simulator to support DSRE.

We first start by explaining the extra state required in the various tiles to ensure functional correctness with DSRE. The various pipelines present in the different tiles require extra state bits to ensure correct execution in the presence of multiple versions of an operand. Our initial implementation of DSRE demonstrated poor performance due to the ALU and network contention of the extra messages, as well as the lag in commit bit propagation. We then looked at hardware techniques to alleviate the bandwidth bottleneck and speedup the commit wave. We found that some of the physical constraints encountered in an actual implementation have a significant effect on DSRE,

but using suitable techniques, we can overcome these limitations.

7.1 Supporting DSRE on the TRIPS Processor

This section describes the changes required to the basic DSRE mechanism for implementation on the TRIPS prototype processor. We also describe the changes in the TRIPS processor required to support DSRE.

7.1.1 DSRE with Multiple Producers

In the TRIPS prototype implementation of an EDGE ISA, multiple instructions can target an instruction's input operand. During runtime, predication guarantees that only one producer will fire and send its value to the consuming instruction. When we introduce selective re-execution in this context, it is possible to have a consumer receive inputs for an operand from multiple producers. Figure 7.1 shows a code snippet that illustrates this behavior along with the data flow graph for the set of instructions in the code. In this example, the *tge* instruction compares the values of R1 and R2 and generates a true or false predicate that it sends to the predicated-move instructions. Only one of the *mov* instruction will fire at runtime and send its value to instruction #8.

In the basic selective re-execution mechanism, there was one producer for each consumer. Hence, the producer had complete control over the value of the version numbers reaching the consumer. However, with multiple producers for a single destination, this one-to-one correspondence between the

```
#1 R[0] read G[1] N[1]
#2 R[1] read G[2] N[1]
#3 N[1] tge N[4,p] N[5,p]
#4 N[2] genu 0x1 N[4]
#5 N[3] genu 0x2 N[5]
#6 N[4] mov_t W[0]
#7 N[5] mov_f W[0]
#8 W[0] write G[4]
```

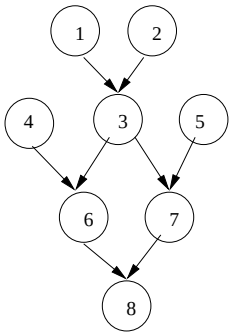

Figure 7.1: EDGE code with multiple sources

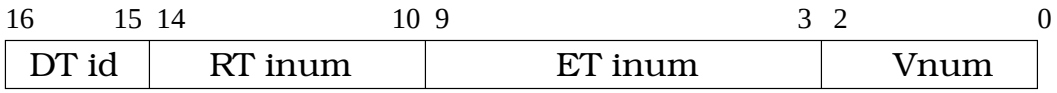

Figure 7.2: Version number with instruction identifier

producer and the consumer is no longer valid. For example in Figure 7.1, with selective re-execution, the *tge* instruction may fire speculatively and generate a false predicate. This predicate can in turn cause instruction #6 to fire and generate an incorrect speculative value that is sent to instruction #8. When the speculation resolves, the *tge* instruction might generate a true predicate and cause instruction #5 to fire and send its result to #8. Thus instruction #8 can get its input from both sources.

To identify the source of an operand, we extend the version number to carry instruction identifiers as shown in Figure 7.2. The instruction identifier is a 14-bit quantity that is used to differentiate replies from the ETs, RTs,

and the DTs. We identify the ETs by using the instruction number of the producer (0-127). We identify replies from the RTs by appending the register read instruction numbers to the ALU instruction numbers. Since there are 32 read instructions in a block, RT identification requires 5 bits. Finally, the top 2-bits are used to identify one of the 4 DTs. DT identifiers are also useful to identify speculative loads that go to different data tiles because of the address interleaving.

Having multiple producers target a single consumer also means that there is no total order among different versions of an operand. To identify null commit messages, we compare the version number received with the commit bit against the last version number received, and if they are identical, the incoming message is treated as a null commit message. Having no total order among different versions of an operand also means that instructions can fire whenever they get a new version number, even if it turns out to be an invalid, older message.

Another aspect of having multiple producers for one operand is that the instruction result needs to be saved and sent with each message, even for null commit messages. The instruction result is required because a consumer might have received another (incorrect) value from a different producer in between, and hence if it needs to fire again, it will not have the right operand. The consumer needs to ensure that the last result generated was using the speculative value that was sent by the producer of the commit bit. Thus, null commit messages in the TRIPS simulator carry the last computed result,

along with the version number of the last computed result.

7.1.2 Changes to the Operand Network

As mentioned in Chapter 3, messages sent on the operand network (OPN) in the TRIPS processor consist of a control packet and a data packet. We send the commit bit and the version number, along with the instruction identifier, in the control packet. Thus, null commit messages are identified by comparing the last version number and instruction identifier received for an operand with the version number and instruction identifier in the control packet. Sending the commit bit and version number with the control packet helps us retire null commit messages, without having to wait for the data packet.

To support DSRE, the control packet in the OPN was extended to carry the commit bit and the version number of the operand. The commit bit is a single bit and the version number is 17 bits. The data packet also carries the commit bit and version number for each operand.

Handling predicate or-ing Handling predicates correctly also requires changes to the basic DSRE mechanism. The TRIPS architecture handles predicates just like other operands. There are essentially two types of predicates—enabling and non-enabling. A predicated instruction executes only if it receives an enabling predicate. With predicate or-ing, an instruction can receive multiple non-speculative, non-enabling predicates. However, it is guaranteed to

```
; Is G[4] <= G[3] <= G[5] ?
#1  R[2] read G[2] N[28]
#2  R[3] read G[3] N[0,1] N[4,0]
#3  R[4] read G[4] N[0,0]
#4  R[5] read G[5] N[4,1]
#5  N[0] tle N[4,P] N[16,P]
#6  N[4] tle_t N[12,P] N[16,P]
#7  N[12] movi_t 1 W[3]
#8  N[16] movi_f -1 W[3]
#9  N[28] ret
#10  W[3] write G[3]
```

Figure 7.3: Predicate or-ing example

receive only one non-speculative, enabling predicate.

With DSRE, predicate or-ing can result in a predicate receiving multiple commit bits. Hence predicates require two commit bit fields—one for the true predicate and one for the false predicate. An instruction can receive the enabling predicate commit bit only once, while it can receive the non-enabling predicate commit bit many times. Figure 7.3 shows an example code that can result in a predicate receiving multiple commit bits.

The code shown in Figure 7.3 does a 3-way comparison of the values stored in registers G[3], G[4], and G[5]. Instruction N[0] compares values in registers G[3] and G[4], and generates a true predicate only if G[4] <= G[3]. Instruction N[0] generates a false predicate if G[4] > G[3]. The predicate generated by N[0] are sent to instructions N[4] and N[16]. Instruction N[4] is predicated on the result of instruction N[0], and executes only if N[0] produced

a true predicate. Instruction N[4] produces a true predicate if G[3] <= G[5]. If N[4] executes, it will send its result to instructions N[12] and N[16]. From this example, we see that instruction N[16] can receive predicate values from both instructions N[0] and N[4]. N[0] and N[4] can both send true predicates to N[16], which is a non-enabling predicate since N[16] is predicated on false. Thus, with predicate or-ring, an instruction can receive multiple commit bits for the non-enabling predicate.

Since an instruction can receive only one non-speculative enabling predicate, we need a single version number field for predicates that stores the version number of the enabling predicate. This version number helps us identify null messages for predicates by looking only at the control packet. Hence, when an instruction receives an enabling predicate, it stores the enabling predicate version number in the predicate version number field.

7.1.3 Changes to the Global Tile

The Global Tile (GT) receives branch updates on the OPN. Branch updates specify the address of the next block to fetch. The GT compares this address with the predicted address to validate branch prediction.

With selective re-execution, the GT can receive multiple branch updates for a block. The GT uses only the update that has the commit bit set to validate the branch prediction. Using speculative branch updates to validate branch prediction can result in higher performance due to earlier branch updates. It can also result in poor performance if the update is incorrect and

results in an unnecessary pipeline flush. In this dissertation, we only use the branch update that has the commit bit set.

The GT also responds to the *mfpc* (move from PC) instruction by sending the value of the PC to the destination specified in the instruction. With selective re-execution, the GT can get multiple *mfpc* requests. The GT uses the commit bit of the *mfpc* request to determine the commit bit for the reply. The GT sets the commit bit for the reply when it receives a *mfpc* request with the commit bit set.

7.1.4 Changes to the Execution Tile

This section describes the changes we made to the execution tile to support DSRE. As shown in Section 4.1.1, with DSRE an instruction has an extra committed state associated with it. We added a number of state bits for each instruction mapped on an ET to track its state through the various pipeline stages. The rest of this section describes these changes in more detail.

The ETs required the most change to support re-execution. First, we added commit bit and version number fields to all the reservation station entries in the ET for each operand. This expansion requires a 1-bit commit bit and a 17-bit version number for operand A, a 1-bit commit bit and a 17-bit version number for operand B, and 2 bits for the true and false commit bits and a 17-bit version number for the predicate. As mentioned in Section 7.1.1, the 17-bit version number also contains a 14-bit instruction identifier. Hence, the number of bits required for the version number is 15. A larger version

number allows an instruction to fire more times speculatively. Second, we used a separate register file to store the result for each instruction. Third, we added a number of status bits to track the extra state for each instruction. These include:

- Null commit bit: Each reservation station entry has a 1-bit null commit message bit for operand A and a 1-bit null commit message bit for operand B. When this bit is set, it means that the instruction has already fired once with the value of the operand in the reservation station. An instruction sends a null commit message if all its operands have their null commit message bit set. The null commit message bit for an operand can be set in two ways:

    1. When a message arrives that has the commit bit set and has the same version number as the last version of this operand. If the instruction has executed with the previous value of this operand, the null commit message bit is set for the operand.

    2. When an instruction executes using the value of an operand that has received its commit bit. Since the output null commit bit is generated by a performing a logical *AND* of the commit bits of all input operands, setting the null commit message bit for an operand when the instruction executes ensures that the ET can send a null commit message for this instruction, if the other operands for this instruction receive null commit message messages.

Null commit messages are identified by doing a logical *AND* of the null commit message bits of all the operands of an instruction. When all the operands of an instruction have their null commit message bit set, the instruction can send a null commit message to its consumers. When the ET receives a null commit message bit for an instruction, it checks to see if the other operand(s) have received their commit bit. It marks the instruction as not issued only if all the operands of the instruction have received their commit bit to prevent redundant execution of the instruction.

- Executed bit: Each ET reservation station has an *executed* bit. The *executed* bit is set when the instruction executes for the first time. An instruction can send a null commit bit only if this bit is set for the instruction.

- Executing bit: This bit is set by the read stage of the pipeline when the instruction is issued to an ALU. The bit is reset after the instruction has finished executing. If the *executing* bit is set, the same instruction cannot be issued again until the bit is reset. The ET also uses this bit to determine if it can select an instruction to send a null commit bit. It might happen that a long latency instruction, like a multiply instruction, gets a null commit message while it is still executing. We want the multiply instruction to send a null commit message to its consumers only after it has finished executing. Resetting this bit prevents null

commit messages from racing ahead of the instruction's result to the instruction's consumers. One drawback of this approach is that it also prevents concurrent execution of different versions of an instruction.

- Issued-executed bit: Each ET reservation station entry has an *issued-executed* bit. The ET resets this bit when an instruction is issued by the select or by the read stage of the pipeline, and sets the bit when the instruction finishes execution. This bit is required for correct identification of null commit messages. In the ET pipeline, an instruction can stay selected for multiple cycles before it executes, if the ET pipeline is stalled. When an instruction is selected by the select stage, the ET sets the *issued* bit for the instruction. A definitely selected instruction that is not issued by the read stage stays in the pipeline until it is issued. If such an instruction receives a null commit message, the instruction can incorrectly send a null commit message before it has executed. The *issued-executed* bit is used to identify this case. The null message bit for an operand is set only if the *issued-executed* bit is set or the *executing* bit is set.

- Num-fired counter: This counter is used for tracking the number of times a particular instruction executed speculatively. This counter is useful for throttling speculative execution of instructions. As we showed in Chapter 4, best performance is achieved when an instruction is allowed to execute no more than 4 times speculatively for the EEMBC benchmarks.

The basic operation of the ET with re-execution is shown in Figure 7.4 and Figure 7.5. Figure 7.4 shows the various operand processing steps in the ET. When an operand is received at the ET, it checks to see if a previous version of the operand has been received. If there are no previous versions of the operand, the ET marks the operand as ready. If a previous version of the operand has been received by the ET, the ET compares the version numbers of the operand. If the current version number of the operand is identical to the previous version number, the operand is guaranteed to have its commit bit set. The ET sets the null commit message bit for this operand, and resets the *issued* bit for the instruction only if the other operands of the instruction have their commit bit set. Resetting the *issued* bit only if the other operands of the instruction have their commit bit set ensures that the instruction does not execute unnecessarily and send the same result as the last execution.

If the version number of the received operand is different from the last version number, the ET checks the *num-fired* counter to determine the number of times this instruction has fired speculatively. If the instruction has fired fewer than the maximum number of times it is allowed to fire speculatively, the *issued* bit for the instruction is reset to re-execute the instruction. If the instruction has already executed the maximum number of times it is allowed to execute speculatively, the *issued* bit for the instruction is reset only if the instruction has received the commit bit for all its operands. Otherwise, the message is ignored.

Figure 7.5 shows the various steps in the execution of an instruction

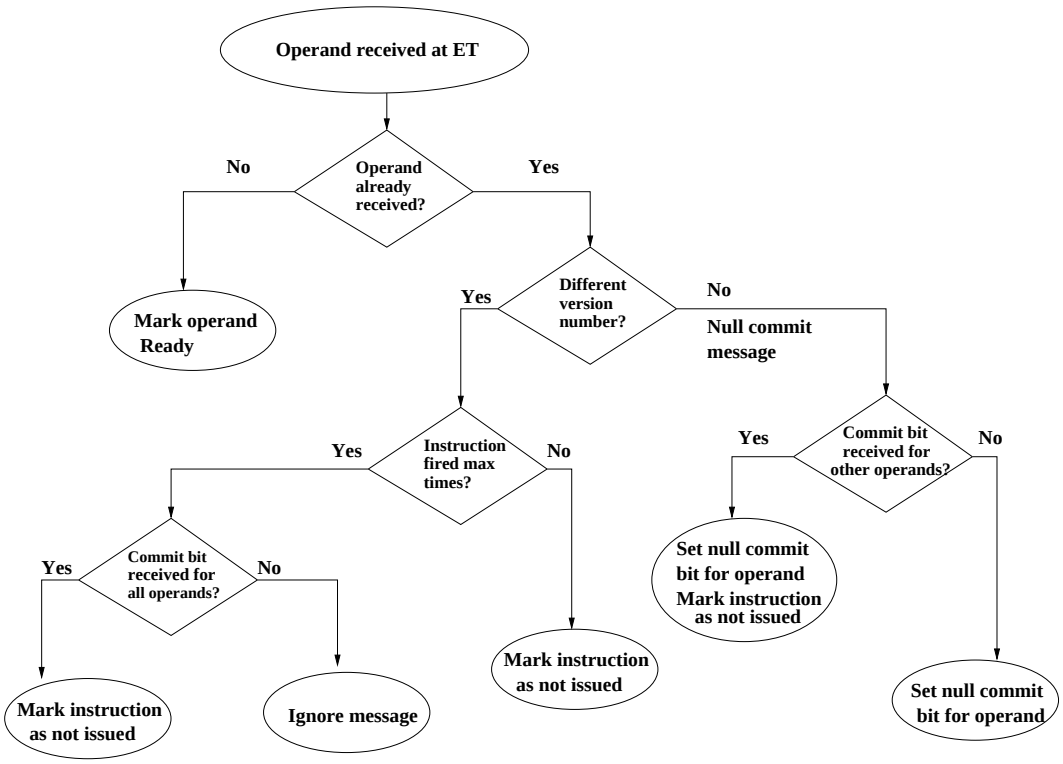

Figure 7.4: Operand processing with re-execution in the execution tile (ET)

with selective re-execution. Instructions that have all their operands ready are selected for issue by the select stage. When the select stage selects a ready instruction, it sets the *issued* bit for the instruction. The select stage also resets the *issued-executed* bit. The instruction then proceeds to the read stage of the pipeline. The read stage checks for functional unit availability, and issues the instruction to the appropriate functional unit. The read stage also sets the *issued-executed* and the *executing* bit for the instruction. The instruction then proceeds to the execute stage of the pipeline where it computes its result. Once the instruction has finished executing, the execute stage resets the *executing* bit

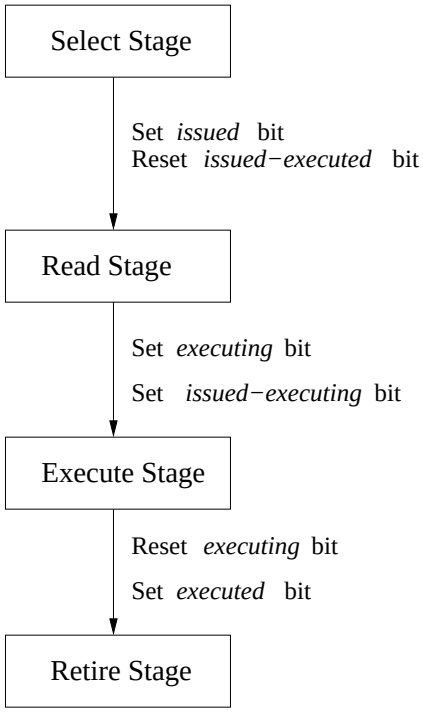

Figure 7.5: Instruction execution with re-execution in the execution tile (ET)

and sets the *executed* bit for the instruction. The execute stage also increments the *num-fired* counter for the instruction.

7.1.4.1 Handing Multiple Versions

With selective re-execution, the ET can receive multiple versions of an operand. When the ET receives a new version of the operand, it resets the *issued* bit for this instruction, so that it can be selected again for execution. The select stage selects an instruction and it is issued by the read stage only if the instruction's *executing* bit is not set. Using the *executing* bit prevents

null commit messages from racing ahead of the actual result to the consumers of the instruction.

7.1.4.2 Identifying Null Commit Messages

The ET identifies null commit messages for operands by comparing the version numbers received for the operand. When the ET receives a new version of an operand, it compares the incoming version number with the last version number for the operand. If the version numbers are the same and the commit bit is set for the new version, then it is identified as a potential null commit message. The ET has to ensure that the instruction executed using the last value of the operand, before it can set the null commit message bit for the operand. To determine if the instruction executed with the last value of the operand, the ET checks the *issued-executed* and *executing* bit of the instruction. If either one of the bits is set, the ET sets the operand's null commit message bit.

The ET identifies output null commit message for an instruction by examining the null message bits for the operands of the instruction. If all the operands of an instruction have their null commit message bit set, then a null commit message is sent for the instruction. The null commit message essentially involves sending the last output of the instruction with the last version number and the commit set. When a null commit message for an operand is received, the ET needs to identify if it can send a null commit message for the instruction, before resetting the instruction's *issued* bit. To

determine if a null commit message can be sent, the ET checks the null commit message bits of the other operands of the instruction, when it receives the null commit message for an operand. If this operand is the last arriving operand, then the *issued* bit is reset so that the select or the read stage can process this instruction and send a null commit message. Otherwise, the ET sets the null commit message bit for the operand, but does not mark the instruction as not issued. Not resetting the *issued* bit ensures that an instruction does not send redundant speculative results to consumers.

7.1.4.3 Handling Predicates

Predicates require special handling in the ET. As shown in Figure 7.3, with predicate or-ing an instruction can get both true and false predicates for an instruction. Once an instruction has executed, we do not need to reset its commit bit when it receives a new version of the predicate. Also, the ET can send a null commit message for the instruction without looking at the predicate version number, when it receives the commit bit for the enabling predicate and the other operands for the instruction have their null commit message bit set. However, this functionality requires looking at the data packet of the predicate.

To identify null commit message bit for enabling predicates, the ET stores the version number of the last enabling predicate received. If the ET receives a commit bit with the same version number, it can determine that only a null commit message needs to be sent for this instruction. With this

optimization, the ET does not need to wait for the data packet to identify null commit message bits for predicates.

7.1.5 Changes to the Register Tile

The register tile handles reads and writes to the architectural register file. The RT processes the block input instructions in each block by sending values from the register file or the write queues. Write instructions wake up pending read instructions when they reach the RT.

Since a producer can send multiple versions of an operand with selective re-execution, the RT assigns a version number to each read reply that it sends on the OPN. To handle the case where multiple producers target the same consumer, the version numbers also contain a register instruction number, to uniquely identify the reply from the RT.

With re-execution, multiple versions for a write instruction can arrive at the RT. Every version of a write instruction that arrives at the RT wakes up any read instructions that are waiting on the write. Reads reply with monotonically increasing version numbers. The RT sets the commit bit for replies that are sent by reading the architectural register file. If the read reply is satisfied by a data-speculative write instruction, the read reply is also speculative and its commit bit is not set. When the write becomes non-speculative, the read sends the commit bit to its consumers.

The RT identifies null commit messages by examining the version number of the write instructions. To support re-execution, the RT includes a ver-

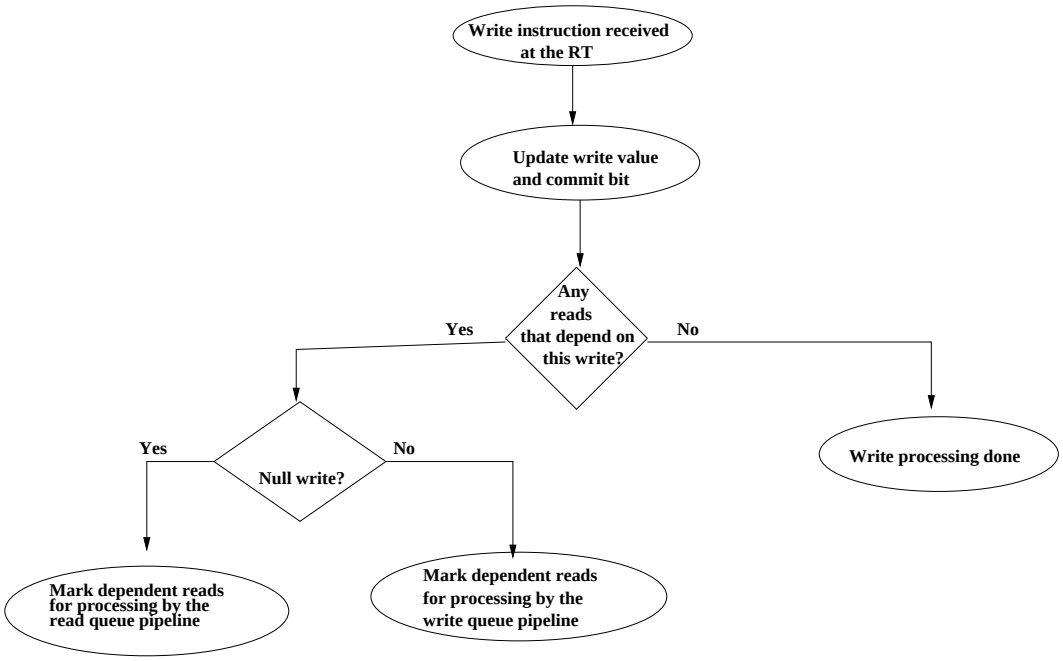

Figure 7.6: Write processing with re-execution in the register tile (RT)

sion number and a commit bit field with each write queue entry. The read queue entries in the RT have two version number fields, a commit bit field, and a null commit message bit field. The two version number fields are labeled in-version-number and out-version-number. The in-version-number field stores the version number of the last write that woke up the read instruction. The out-version-number field stores the version number of the last reply corresponding to this read.

When a new version of a write instruction is received at the RT, it searches the read queue and wakes up all read instructions that depend on this write. The write queue pipeline also compares the version number of the

write instruction with the in-version-number of the read instruction. If the two version numbers match, the write is a null commit message. The null commit message bit for the read instruction is set on a match. When read instructions have the corresponding null commit message bit set in the read queue, they send a null commit message by sending the commit bit with the same version number as the last reply.

With selective re-execution, a speculative write instruction may wake up a corresponding read instruction that is subsequently nullified by a null write instruction. When the RT receives a null write instruction, it resets the issued field of read instructions that were satisfied by the previous version of this write instruction. These reads are then processed by the read queue pipeline, and get new values either from another write instruction or from the architectural register file. Figure 7.6 shows the steps involved when a new write arrives at the RT.

7.1.6 Changes to the Data Tile

The DT is responsible for preserving sequential memory semantics in the TRIPS processor. The load-store queue in the DT tracks the dynamic dependences among in-flight loads and stores. The DT forwards store value from earlier stores to loads.

The dependence predictor in the DT predicts if a load is independent of prior stores. When a load is predicted dependent on a store, it is prevented from sending a reply until all prior stores have resolved. Loads that

are predicted independent incorrectly result in pipeline flushes. Loads that are predicted dependent incorrectly lose an opportunity to send their replies earlier, thus resulting in lower performance.

Withe DSRE, the DT can send speculative replies for loads that are predicted dependent. The commit bit for the load is sent only when all prior stores resolve. Stores that match with a later load can initiate re-firing of the load. Thus, using selective re-execution, a load can send multiple replies speculatively. The load-reissue pipeline in the DT is used to send commit bits for loads when all prior stores have resolved.

Using the dependence predictor in the DT, we can implement commit slicing by sending commit bits for loads that are predicted independent. We evaluate commit slicing using the simple 1-bit predictor implemented in the TRIPS prototype and the more complex *first-store* predictor in the next section.

Load Wake Up Policy With the *first-store* predictor, matching stores are allowed to wake up deferred loads. With selective re-execution, stores that arrive at the DT can be data-speculative. Hence we can have two load wake up policies. In the first policy, data-speculative stores are not allowed to wake up loads by preventing the ETs from sending speculative stores to the DT. Preventing speculative stores from reaching the DT reduces contention in the network and the DT pipeline. However, it prevents the speculative wave from running ahead, and can result in poor performance when the speculative store

value is correct. The second policy allows speculative stores to reach the DT and wake up matching loads. Allowing speculative stores to wake up loads can result in better performance when the store address and the store value of speculative and non-speculative versions are the same. However, if the store value changes, the DT needs to re-fire loads that match with the store. Worse, if the store address changes, we need to identify loads that were woken up incorrectly by the last version of this store and re-fire them. The DT also needs to wake up the loads that match the new address. Identifying both these cases involves doing two content addressable memory (CAM) matches when stores arrive at the DT, and is expensive in terms of timing and power. Also, with speculative stores, the LSQ needs to be augmented to store the address of the previous version of the store. The old address is used to identify loads that might have received incorrect values. Hence, speculative stores also have an area overhead associated with them.

Another case arises with speculative stores when the speculative versions of the same store go to different data tiles. When the non-speculative version of the store eventually arrives at a data tile, its arrival is broadcast on the Data Status Network (DSN) to all the other DTs. Any DT that received a speculative version of this store has to perform a CAM match to determine if any loads received incorrectly forwarded value from the store. All such loads need to be reissued. Since every cycle, each DT gets a message from all the other three DTs on the DSN, in the worst case this will involve 3 CAM matches for stores received by other tiles. Hence, the DT will have to do CAM match

using 5 addresses per cycle in the worst case. Due to the large area and power requirements, we did not pursue this approach and speculative stores were not allowed to reach the DT.

For every load in the LSQ, we added an *executing* bit to support re-execution. The DT uses this bit to ensure that the null commit message are not sent to the load's consumer before the load data. Null commit messages are sent by the reissue pipeline. It might happen that the reissue logic determines that it is safe to send the commit bit for a load, because all prior stores have resolved, while the main load-store pipeline is still processing the load. In this case, the reissue logic can potentially send the commit bit before the load value to the load's consumer. This bit is set when the load enters the DT pipeline, and is reset when the load reply is sent. A null commit message cannot be sent for a load when its *executing* bit is set.

7.2 DSRE Performance

To evaluate the performance of DSRE on an actual EDGE implementation, the TRIPS prototype simulator was modified to support re-execution. As described in the last section, we modified the different tiles in the simulator to implement basic selective re-execution. The details of the simulated processor are shown in Table 7.1.

Feature	Details
ALUs	16 ALUs connected by a routed operand network. The ALUS have both integer and floating point (FP) units.
Instruction latency	1-cycle for basic integer ops like add and shift. 3-cycle, pipelined integer multiply. 24-cycle, non-pipelined integer divide. 4-cycle, pipelined FP add. 4-cycle, pipelined FP multiply. 2-cycle, pipelined FP convert and compare. FP divide not supported in hardware.
Branch Predictor	Next block predictor similar to the Alpha 21264 tournament predictor with local, global, and choice predictors.
Instruction cache	64 KB, 16 KB per bank, 2-way set associative, 64 byte line size.
Data Cache	32 KB, 8 KB per bank, 2-way set associative, 64 byte line size.
L2 cache	2 MB, 2-way set associative, 64 byte line size.

Table 7.1: Simulated TRIPS processor configuration

The initial implementation of DSRE on TRIPS involved adding support for sending multiple speculative versions for the same operand, and sending a commit bit when the operand became non-speculative. Null commit messages in this implementation were treated like regular messages in the ET and acted like single-cycle ALU operations. Thus, null commit messages went through the various pipeline stages before being processed by the ET router. The LSQ in the DT was augmented to store the reply for a load, and this stored load value was sent with the null commit message. We compared the performance of this scheme against an oracle policy that does perfect load-store prediction,

an aggressive policy that treats all loads as independent, a conservative policy that treats all loads as dependent, and a policy with the 1-bit dependence predictor that is implemented in the TRIPS prototype processor. Table 7.2 shows the performance of the EEMBC suite for these configurations.

Benchmark	cons (IPC)	aggr (IPC)	all-stores (IPC)	dsre (IPC)	oracle (IPC)
a2time01	0.687	0.755	0.776	0.705	2.362
aifftr01	0.554	0.592	0.711	0.544	2.432
aifirf01	0.870	0.871	1.657	0.894	2.590
aiifft01	0.542	0.567	0.672	0.535	2.539
autcor00	1.185	1.166	1.132	1.409	1.166
basefp01	0.817	0.849	1.028	0.810	1.164
bezier01	1.202	2.089	2.500	1.262	2.499
bitmnp01	0.644	0.887	0.875	0.635	1.665
cacheb01	0.574	0.668	0.867	0.614	1.532
canrdr01	1.143	1.123	1.325	1.221	1.393
conven00	0.534	0.523	0.537	0.485	0.537
fft00	1.057	2.601	2.633	1.052	2.630
idctrn01	0.644	1.217	1.431	0.653	2.632
iirflt01	0.489	0.477	0.826	0.499	1.910
ospf	0.597	0.810	0.881	0.590	0.857
pntrch01	0.802	0.720	0.868	0.717	1.008
pktflow	0.864	0.980	1.209	0.865	1.221
puwmod01	0.686	0.583	0.901	0.656	2.178
routelookup	0.554	0.554	0.554	0.702	0.554
rspeed01	0.679	0.599	0.881	0.669	2.111
tblook01	0.678	0.673	0.736	0.609	0.763
ttsprk01	0.617	0.675	0.720	0.587	0.758
viterb00	0.627	2.370	1.394	0.589	2.749
Mean	0.692	0.782	0.925	0.691	1.310

Table 7.2: Comparison of initial DSRE implementation on the TRIPS prototype simulator

From Table 7.2, we see that our initial implementation DSRE performs similarly to the conservative policy. The aggressive policy performs 13% better than the conservative policy. Rolling flushes in the TRIPS processor reduce

the cost of flushes contributing to the higher performance for the aggressive policy.

The 1-bit dependence predictor yields an average 18.2% improvement over the aggressive approach. The predictor has 1024 1-bit entries, and an entry is set for a load that mis-speculates and causes a dependence violation. All bits in the predictor are cleared unconditionally after committing 10,000 blocks. Because of the simple nature of the predictor, it is not good for catching complex dependence patterns in benchmarks.

The 4th column in Figure 7.2 shows the performance of our basic selective re-execution scheme. The simple re-execution scheme performs similarly to the conservative policy because only multi-cycle operations benefit from DSRE. The aggressive and 1-bit predictor schemes outperform the basic DSRE scheme.

To study the performance of DSRE with commit slicing, we evaluated the performance of DSRE with the 1-bit predictor, and the more complex 2-bit *first-store* predictor. Table 7.3 shows the performance, with commit slicing, for the two predictor configurations. Commit slicing with the *all-stores* predictor performs worse than the dependence prediction using the *all-stores* predictor, for the initial implementation of selective re-execution.

		Commit slicing	
Benchmark	**DSRE (IPC)**	**all-stores (IPC)**	**first-store (IPC)**
a2time01	0.705	0.779	1.416
aifftr01	0.544	0.673	0.552
aifirf01	0.894	1.572	1.637
aiifft01	0.535	0.639	0.627
autcor00	1.409	1.422	1.422
basefp01	0.810	0.991	1.184
bezier01	1.262	2.487	2.494
bitmnp01	0.635	0.871	0.890
cacheb01	0.614	0.883	1.165
canrdr01	1.221	1.600	1.573
conven00	0.485	0.510	0.483
fft00	1.052	2.621	2.577
idctrn01	0.653	1.243	1.041
iirflt01	0.499	1.016	0.973
ospf	0.590	0.961	0.847
pntrch01	0.717	0.742	0.867
pktflow	0.865	1.115	1.313
puwmod01	0.656	0.761	0.980
routelookup	0.702	0.702	0.702
rspeed01	0.669	0.818	0.883
tblook01	0.609	0.737	0.770
ttsprk01	0.587	0.755	0.776
viterb00	0.589	1.330	1.433
Mean	0.691	0.927	0.971

Table 7.3: Comparison of DSRE with and without commit slicing

We found two reasons for the lower performance of selective re-execution. First, the null commit messages were being treated like regular messages, and hence ended up taking resources within the tiles, like the ALU bandwidth.

Second, the the extra traffic generated by DSRE resulted in more contention on the OPN. We looked at a number of enhancements to the various tile to increase the performance with selective re-execution. The following section explains the cause for poor DSRE performance, along with the techniques we evaluated, to improve performance.

7.3 Performance Enhancements to DSRE on TRIPS

In the simple implementation of selective re-execution, the pipelines in the ET and DT treat null commit messages just like regular messages. Hence, in the ET for example, the messages go through the select, read, execute, and writeback stages of the pipeline. Thus, using null commit messages does not yield any significant performance advantage. Since the messages are treated like single-cycle ALU operations, only multi-cycle operations like multiply tend to benefit from this naïve implementation. In this section, we discuss various modifications to the DSRE implementation to increase its performance.

7.3.1 Accelerating Commit Messages

An important property of null commit messages is that they do not need to use the ALU for computing the result. Since null commit messages are sent on a correct speculation, as long as instructions store the result of their last computation, they can send a null commit message using the stored result.

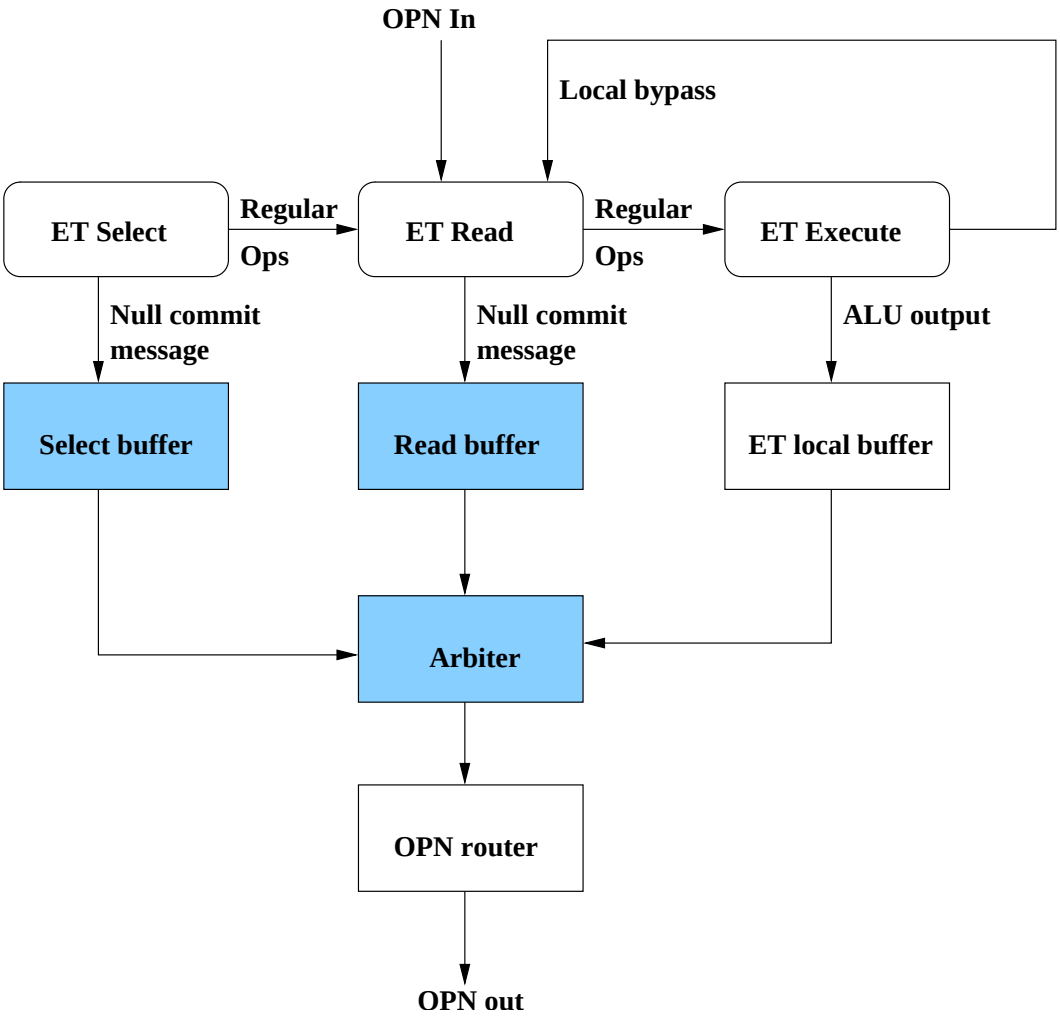

Figure 7.7: Modified execution tile pipeline

When instructions arrive at the ET, we can identify null commit messages by examining the version number in the control packet. Instructions that need to send null commit messages can be retired without having to go through the regular ET pipeline, with little extra logic in the select and read pipeline

stages. With the extra logic, the read stage can send null commit messages for instructions whose last operand is bypassed, and the select stage can handle null commit messages for definitely selected instructions. We added buffers in the select and read stages of the pipeline to store the null commit messages from these stages. These instructions arbitrate for the OPN router with the execute stage of the pipeline to send their replies. The modified ET pipeline is shown in Figure 7.7. The shaded blocks in Figure 7.7 represent structures that we added to the ET to improve DSRE performance. The ability to send null commit messages from the read and select stages reduces ALU contention, and decreases the latency to process null commit messages.

In our simple selective re-execution implementation, the DT sent null commit messages using the main load-store pipeline. The reissue logic in the DT identifies null commit messages after all prior stores have resolved for a load, and these null commit messages pass through the reissue pipeline before reaching the load-store pipeline. Since the LSQ stores the last reply for every load, some extra logic in the reissue pipeline can send null commit messages to the load's consumer from the reissue pipeline. Specifically, we need to allow the reissue stage of the pipeline to write null commit messages to the output buffer in the DT. The ability to send null commit messages from the reissue pipeline reduces contention in the main load-store pipeline and decreases the latency to process null commit messages in the DT.

Table 7.4 shows the performance of DSRE, with commit slicing using the 2-bit *first-store* predictor, for the augmented ET and DT pipelines. The

first column in Table 7.4 shows the performance of DSRE without these enhancements. The second column in Table 7.4 shows the performance of DSRE with accelerated commit messages. From Table 7.4, we see that modifying the ET and DT pipelines to accelerate commit messages results in a 2.4% improvement in performance over our initial implementation. We next analyzed the OPN bandwidth to understand its impact on DSRE performance.

7.3.2 OPN Bandwidth

Selective re-execution always results in extra messages on the OPN, both when the speculation is correct and when it is incorrect. When the speculation is correct, we need to send null commit messages to validate the speculation. When the speculation is incorrect, we need to send the correct value of the operand to the consumers. Thus, extra OPN bandwidth is required to efficiently support selective re-execution.

To evaluate the impact of bandwidth on performance, we ran experiments with infinite OPN bandwidth. We simulated infinite OPN bandwidth by ensuring that packets faced no contention in the network. The tiles can still process only one local packet every cycle. Hence, packets get queued at the input and output buffers in each tile. These buffers were made infinitely large to avoid contention. Infinite OPN bandwidth does not increase the local bypass path in the ET, which is still limited to one.

Column 3 in Table 7.4 lists the performance of DSRE with infinite OPN bandwidth. Comparing column 3 and column 4 in Table 7.4, we see that

there is a 10% improvement in performance with infinite bandwidth, clearly indicating that limited bandwidth on the OPN does limit performance with selective re-execution.

Benchmark	**DSRE (IPC)**	**ET-DT NULL (IPC)**	**Perfect OPN (IPC)**	**Multiple OPN (IPC)**
a2time01	1.416	1.470	1.929	1.807
aifftr01	0.552	0.574	0.636	0.587
aifirf01	1.637	1.648	1.766	1.678
aiifft01	0.627	0.664	0.754	0.677
autcor00	1.422	1.422	1.466	1.465
basefp01	1.184	1.188	1.248	1.234
bezier01	2.494	2.494	2.787	2.790
bitmnp01	0.890	0.908	1.020	0.958
cacheb01	1.165	1.166	1.252	1.195
canrdr01	1.573	1.626	1.724	1.680
conven00	0.483	0.483	0.481	0.481
fft00	2.577	2.577	2.695	2.658
idctrn01	1.041	1.105	1.439	1.135
iirflt01	0.973	1.003	1.057	1.005
pntrch01	0.867	0.867	0.954	0.900
pktflow	1.313	1.316	1.404	1.367
puwmod01	0.980	0.984	1.068	1.010
routelookup	0.702	0.702	0.759	0.719
rspeed01	0.883	0.894	0.957	0.907
tblook01	0.770	0.773	0.919	0.867
ttsprk01	0.776	0.778	0.861	0.811
viterb00	1.433	1.588	1.878	1.716
Mean	0.977	0.996	1.094	1.034

Table 7.4: Performance (IPC) of DSRE with enhancements

Since unlimited bandwidth is not feasible in a real implementation, we implemented a second operand network to double the bandwidth. The tiles were allowed to send operands on either one of the two OPNs, labeled OPN1 and OPN2. Doubling the OPN bandwidth requires twice as many wires between tiles for carrying the operands. It also requires two routers in each tile for routing packets on the two OPNs. We also need logic at the output of each tile to decide which OPN to use for a particular packet. We used a simple arbitration policy that checked OPN1 first to see if we could send a packet. If OPN1 was busy, we checked OPN2. The local bypass in the ET was also doubled to accommodate two instructions that targeted their parent node. Finally, the bypass paths in the ETs were doubled to accommodate two OPN bypasses and two local bypasses. Note that even with two local bypasses, ALUs are allowed to retire only one instruction every cycle. However, the ET retires null commit messages from the select and read stages, and these messages utilize the extra bandwidth.

Column 5 in Table 7.4 lists the performance of DSRE when we double the OPN bandwidth. Comparing column 3 and column 5 in Table 7.4, we see that doubling the OPN bandwidth does not yield the same performance improvement as infinite bandwidth, and results in a smaller 4% improvement in performance.

7.3.3 Dependence Predictor Policy

The final performance improvement technique we evaluated involved varying the *first-store* dependence predictor. The 2-bit dependence predictor we used in our evaluation uses three states for dependence prediction. With selective re-execution, we can use all four states of the predictor to implement different load reply policies. Table 7.5 shows two different load reply policies with a 2-bit dependence predictor. We used a simple algorithm for training the predictor that incremented the counter when a load is incorrectly predicted conflicting and decremented the counter when the load is incorrectly predicted conflicting. The results shown in the last section use Policy 2 in Table 7.5.

Since we can have five different load reply policies with selective re-execution, we implemented a 3-bit *first-store* predictor to choose the appropriate load reply for a load, depending on the state of the predictor counter corresponding to the load. We also evaluated the 3-bit *first-store* store with dependence prediction. Table 7.6 shows the load reply policy for dependence prediction and commit slicing with the 3-bit predictor. We used the same training algorithm as the 2-bit predictor, where the counter for a load is incremented when the load is incorrectly predicted conflicting, and decremented when the load is incorrectly predicted conflicting. The number of entries in the predictor table was 1024.

	Prediction	
Counter Value	**Policy 1**	**Policy 2**
00	**No conflict**: Commit bit is sent along with load reply. Pipeline flush on a mis-speculation.	**No conflict**: Same as policy 1.
01	**Might conflict**: Load reply sent without commit bit when load arrives at the DT. Commit bit sent when all prior store resolve.	**Might conflict**: Same as policy 1.
10	**First store without commit**: Load reply sent without commit bit on first matching store. Commit bit sent when all prior store resolve.	**First store with commit**: Load reply sent with commit bit on first matching store. Pipeline flush on a mis-speculation.
11	**All stores**: Load reply and commit bit sent after all prior stores resolve.	**All stores**: Same as policy 1.

Table 7.5: Load reply policies with a 2-bit predictor

	Prediction	
Counter Value	**Dependence Prediction**	**Commit slicing**
000, 001	**No Conflict**: Load reply sent before prior stores resolve.	**No conflict**: Commit bit is sent along with load reply. Pipeline flush on a misspeculation.
010, 011	**First store**: Load reply sent on first matching store.	**Might conflict**: Load reply sent without commit bit when load arrives at the DT. Commit bit sent when all prior store resolve.
100	**First store**: Load reply sent on first matching store.	**First store with commit**: Load reply sent with commit bit on first matching store. Pipeline flush on a misspeculation.
101	**First store**: Load reply sent on first matching store.	**First store without commit**: Load reply sent without commit bit on first matching store. Commit bit sent when all prior stores resolve.
110	**All stores**: Load reply sent after all prior stores resolve.	**First store without commit**: Load reply sent without commit bit on first matching store. Commit bit sent when all prior stores resolve.
111	**All stores**: Load reply sent after all prior stores resolve.	**All stores**: Load reply and commit bit sent after all prior stores resolve.

Table 7.6: Load reply policies with a 3-bit predictor

Benchmark	cons (IPC)	aggr (IPC)	first-store (IPC)	DSRE with commit slicing (IPC)	oracle (IPC)
a2time01	0.702	0.767	0.842	1.797	2.418
aifftr01	0.560	0.600	0.715	0.752	2.477
aifirf01	0.884	0.890	1.675	1.734	2.635
aiifft01	0.547	0.576	0.699	0.714	2.592
autcor00	1.208	1.210	1.208	1.208	1.210
basefp01	0.845	0.886	1.072	1.204	1.212
bezier01	1.195	2.137	2.793	2.784	2.789
bitmnp01	0.678	0.922	0.965	0.896	1.714
cacheb01	0.579	0.689	0.992	1.002	1.535
canrdr01	1.197	1.189	1.430	1.423	1.483
conven00	0.535	0.526	0.538	0.538	0.538
fft00	1.052	2.696	2.726	2.726	2.727
idctrn01	0.652	1.249	1.532	1.466	2.719
iirflt01	0.489	0.480	0.877	1.025	1.944
ospf	0.633	0.864	0.908	0.911	0.917
pntrch01	0.820	0.740	0.916	0.901	1.039
pktflow	0.896	1.026	1.188	1.272	1.272
puwmod01	0.703	0.609	0.913	0.992	2.191
routelookup	0.573	0.573	0.573	0.573	0.573
rspeed01	0.697	0.633	0.889	0.906	2.129
tblook01	0.751	0.744	0.825	0.825	0.854
ttsprk01	0.636	0.696	0.748	0.746	0.782
viterb00	0.647	2.642	2.217	1.786	3.053
Mean	0.709	0.810	0.979	1.020	1.361

Table 7.7: Comparison of load/store recovery schemes with a 3-bit predictor

Table 7.7 compares the performance of our best selective re-execution scheme against the conservative load issue scheme, aggressive load issue scheme,

dependence prediction using a 3-bit *first-store* predictor, and an oracle policy. All the configurations use two OPN channels for extra bandwidth. From Table 7.7, we see that DSRE with commit slicing using the 3-bit *first-store* predictor improves performance over the conservative scheme by 43.9%, aggressive scheme by 25.9%, and the dependence prediction by 4.2%. Oracle still outperforms DSRE by 33.4%. As discussed in Chapter 5, this gap can be bridged using better compiler technology, better predictor training algorithms for, and more sophisticated predictors for commit slicing.

7.3.4 Performance Summary

The performance improvement over dependence prediction of our best selective re-execution implementation on the prototype simulator is lower than the performance improvement we saw with our high-level GPA simulator implementation. The reasons for this are four fold. First, the TRIPS prototype simulator more accurately models the contention in the ALUs and the network than the GPA simulator. Since contention primarily affects the performance of selective re-execution, the performance improvements are lower in the TRIPS prototype simulator. Second, the TRIPS compiler is still being optimized for performance, and hence the code generated is sub-optimal, as is reflected in the poor IPC numbers for the benchmarks even with an oracle policy. For example, in the binaries we used, the compiler is unable to register allocate static variables or optimize structures, resulting in benchmarks having a larger number of load-store dependences. We saw an example of the

performance improvement with selective re-execution with optimized code in Section 5.1.1. Selective re-execution will produce larger performance benefits with better code that exposes more instruction level parallelism. Third, the distributed nature of the LSQ, coupled with the various pipeline stages, results in a larger propagation delay for null commit messages. Finally, our compiler and simulator infrastructure are not completely mature and allowed us to evaluate only the loop-based EEMBC benchmarks. A number of these benchmarks have similar behavior, and behavior does not vary within the inner loop of each benchmark. The performance of the benchmarks is influenced in large part by their loop-based nature. For example, the loop-based nature of these benchmarks results in a large number of stores in-flight to the same pointer address. In the steady state, loads that have multiple matching prior stores are serialized because the dependence predictor is unable to identify the last matching store. The performance will likely be different with larger benchmarks that have different load-store patterns.

7.4 Logic, Timing, and Area Overhead with DSRE

In this section, we discuss the hardware complexity of selective re-execution in a real implementation. The basic selective re-execution mechanism described in Chapter 4 requires only a commit bit and version number for each operand. However, as we discussed in this chapter, in a prototype implementation, we need more state to deal with the various pipelines, and bypass paths present in the processor. The bandwidth constraints in a real

implementation also increase the resources required for selective re-execution.

The logic for handling commit bits and version numbers can be easily implemented in each tile. We need a 17-bit comparator to compare version numbers to identify null commit messages. We need an adder in the ET and DT to increment version numbers for successive replies. We need counters in the ET and the DT to count the number of times an instruction has fired speculatively. This functionality can be simplified if we allow instructions to fire speculatively only once, and track it by using a single bit that is set when the instruction is executed.

The GT requires very little extra logic to implement selective re-execution. The GT ignores speculative branch update messages and reuses the version number and commit bit of the *mfpc* (move from PC) instructions.

The read and write queues in the RT require extra state for tracking version number of incoming writes and outgoing reads. Waking up dependent reads when writes arrive at the RT does not result in extra complexity, as this functionality is already part of the RT implementation. When a null write arrives at the RT, it needs to reset the read instructions that incorrectly received forwarded values from the previous version of the write. This resetting can be accomplished by simply resetting the *issued* bit of the read, and hence does not involve any significant increase in complexity.

The load-store queue in the DT requires extra state bits to support re-execution. Each entry in the LSQ needs fields for storing the incoming version

number and commit bit of loads and stores. It also needs a field to track the version number and commit bit of load replies. The LSQ also needs a register file for holding the results of loads, and an executing bit and null commit message bit for identifying when it is safe to send null commit messages. The only other significant piece of logic required is in the reissue stage of the pipeline for retiring null commit messages. As explained in Section 7.1.6, to reduce complexity of the implementation, we do not allow speculative stores to reach the DT and wake up loads.

The control and data paths of the OPN have to be expanded to accommodate the version number and the commit bit for each operand. We believe that extra OPN bandwidth can be provided without a significant increase in area. The OPN occupies 4% of the processor area in the prototype RTL. Providing a second OPN will increase the area to slightly less than 8% of the processor area.

The main complexity from selective re-execution arises in the ET. The ET needs to store version number and commit bit information for each operand. It also needs to store the last computed result of an instruction. The ET requires some state bits to identify null commit messages, and to track an instruction through various stages of the ET pipeline. We also need buffers in the read and select stages of the pipeline for storing retired null commit messages.

The main logic complexity in the ET arises from processing multiple inputs at the ET. Doubling the OPN bandwidth and the local bypass requires

that the ET process two inputs on the OPN and two inputs on the local bypass every cycle. Once we add the instruction chosen by the select stage, the arbiter in the read stage of the pipeline has to choose from five different inputs for processing the next cycle by the execute stage. The multiplexor in the execute stage has to be changed from 3:1 multiplexor to a 5:1 multiplexor. These changes are shown in Figure 7.8.

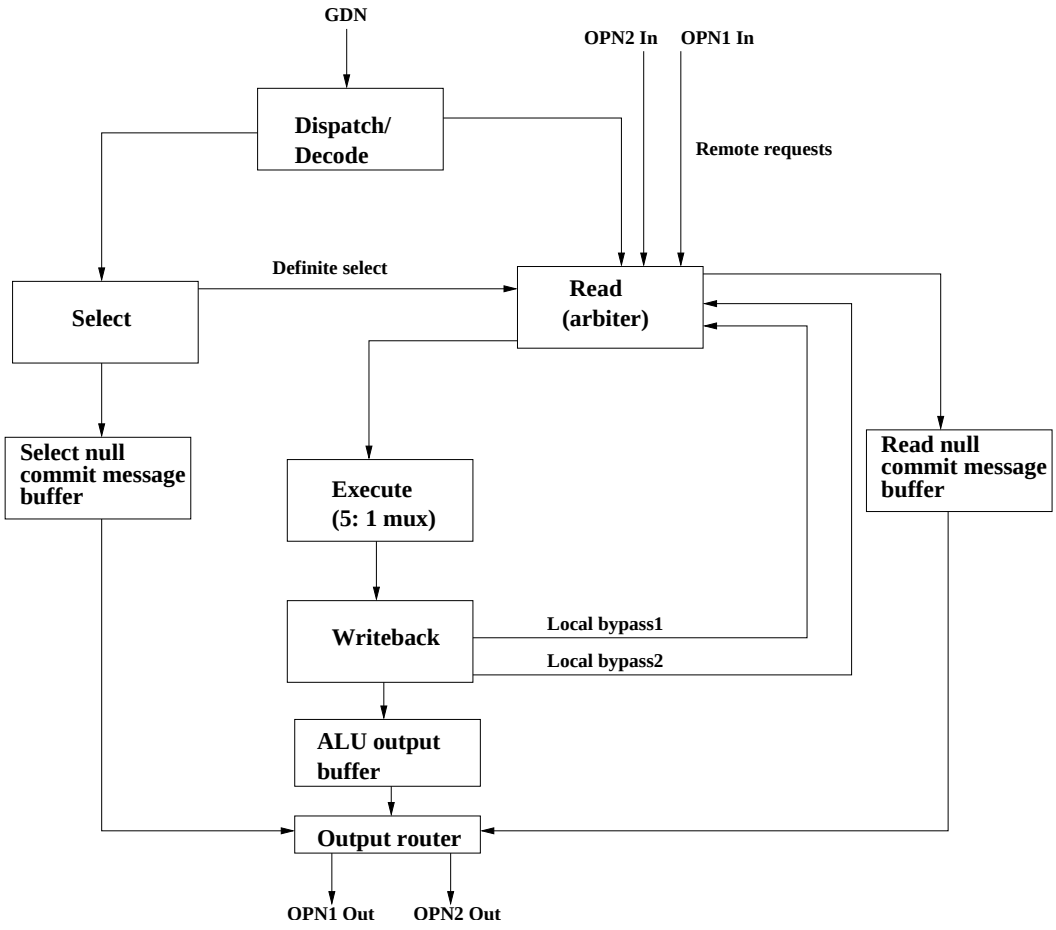

Figure 7.8: Changes to the execution tile required to support the extra bandwidth for re-execution

In summary, implementing selective re-execution requires extra logic in all the major tiles in the TRIPS processor. Even though the mechanism itself uses simple state bits, the various pipelines and bypass paths in the TRIPS processor required extra state for tracking the various states of an instruction. The area overhead associated with selective re-execution is mainly due to the higher bandwidth required to support the extra messages with DSRE. The extra bandwidth was provided by doubling the number of physical channels on the OPN, and augmenting the input and output routers in the various tiles to support the two OPN channels. We found that the complexity of the extra logic required is minimal in most tiles, and the execution tiles required the most hardware support for selective re-execution.

This chapter showed how DSRE can be implemented on a prototype TRIPS processor. We used the TRIPS prototype simulator for implementing and evaluating DSRE. The TRIPS prototype simulator has been validated against the TRIPS prototype RTL, and models all the low-level details of the implementation. We identified the changes required to the basic DSRE mechanism described in Chapter 4 to accommodate the TRIPS prototype processor implementation of its EDGE ISA. We also identified the extra state that is required in each tile of the processor for functional correctness with DSRE. The initial implementation DSRE resulted in poor performance, and we suggested and evaluated techniques to improve the performance. We also did an analysis of the logic, timing, and area overhead with DSRE. We found that the performance improvements with DSRE are lower on the TRIPS prototype

simulator when compared to the GPA simulator. We identified the reasons for the lower performance and suggested solutions that can bridge this performance gap. The next chapter summarizes this work and discusses its broader implications.

Chapter 8

Conclusions

With pipeline flushes becoming expensive in wide-issue, deeply pipelined machines, mechanisms for low-cost recovery from mis-speculations will become increasingly important in future microprocessors. Selective re-execution is one such mechanism for low-cost recovery from data mis-speculations. Although modern superscalar processors implement selective re-execution in a limited fashion, its complexity in a conventional implementation makes it unsuitable as a general mechanism for recovery from data mis-speculations.

In this dissertation, we have designed and evaluated a selective re-execution mechanism for a new class of instruction set architectures, Explicit Data Graph Execution (EDGE) architectures. EDGE architectures are a dataflow-like architecture, in which the instructions specify their outputs explicitly, and do not specify their inputs. Instructions in this architecture execute when they receive all their inputs. The explicit specification of consumers in the instruction set obviates the need for dynamic reconstruction of the data dependences in the processor, thus significantly reducing the complexity of designing a selective re-execution mechanism for EDGE architectures. The mechanism we propose is distributed, and uses simple state bits for recovery.

We used an EDGE-based TRIPS processor as the hardware substrate for evaluating the proposed mechanism. Since EDGE architectures have a block atomic execution model, blocks of instructions are fetched and committed atomically. Instructions in a block stay mapped on the reservation station until the block is ready for commit. Hence, the processor can initiate re-execution of instructions by re-injecting the mis-speculating input.

We used load-store dependence speculation as the driving speculation mechanism for evaluating the performance of the proposed DSRE mechanism. Load-store dependence speculation involves predicting the dynamic dependences between in-flight loads and stores. As shown in Chapter 2, aggressive issue of loads is important for high performance in future, large instruction window machines to exploit high instruction-level parallelism.

We used two software implementations of the TRIPS processor to understand and evaluate the proposed DSRE mechanism. Our initial evaluation involved a high-level, Trimaran-based, GPA simulator that loosely modeled the TRIPS architecture without some of the resource constraints encountered in a real implementation. We formulated basic mechanisms for ensuring correctness of the mechanism using the high-level simulator. These mechanisms involved associating a commit bit with each operand to indicate when the operand became non-speculative with respect to data speculation, and a version number to identify the correct non-speculative version of the operand.

We compared the performance of our basic mechanism against different load issue policies. Our results showed that dependence prediction, using

pipeline flushing as a recovery mechanism, outperformed the configuration that used re-execution for recovery. We identified the lag in the propagation of the commit messages as the reason for the poor performance of the selective re-execution mechanism, and evaluated two mechanism for accelerating the commit wave. These mechanisms resulted in selective re-execution outperforming the best dependence predictor.

After initial evaluation on the high-level GPA simulator, we validated the mechanism on the more accurate TRIPS simulator that faithfully models all the details of the TRIPS prototype processor. We found bandwidth and ALU contention to be significant bottlenecks to performance with selective re-execution. We proposed and evaluated mechanisms to alleviate these constraints. The commit bit propagation delay was exacerbated in the TRIPS prototype processor due to the distributed nature of the LSQ. Our results showed that selective re-execution does provide performance benefits in future, large instruction window machines, but it needs to be carefully tuned to account for the limitations imposed by an actual implementation.

Should designers consider DSRE?

DSRE on the TRIPS prototype simulator resulted in a mere 4% mean performance improvement across the set of EEMBC benchmarks. In the light of this, we are forced to ask whether the extra complexity with DSRE is worth the performance improvement. We identified four reasons for the lower performance of DSRE on the TRIPS prototype simulator:

1. More accurate network and ALU contention modeling.
2. Extra logic required to handle the complexity associated with a prototype implementation.
3. Sub-optimal code that exposed low instruction level parallelism.
4. Sub-optimal code that generated a large number of redundant loads and stores.

We hand-optimized one benchmark, *aiifft01*, by removing some redundant loads and stores, which resulted in a 39% improvement in performance over dependence prediction with selective re-execution. The low performance of the non-optimized code with DSRE was primarily due to the fourth reason. The performance improvement with the hand-optimized code demonstrated that the DSRE mechanism is able to tolerate the extra ALU and network contention, and provide speedup despite the added overhead. Although we hand-optimized only one benchmark in this dissertation, we expect DSRE will provide substantial improvement in performance with load-store dependence speculation on optimized benchmarks with similar characteristics. Whether future TRIPS workloads will have these characteristics remains to be seen and is an open question.

8.1 Dissertation Summary

This research has demonstrated one way to implement distributed, selective re-execution for Explicit Data Graph Execution architectures. Using

one particular implementation of an EDGE architecture, the TRIPS processor, we evaluated the performance benefits of the mechanism, on a high-level simulator, and a low-level simulator that models a prototype implementation in great detail. The basic mechanism requires simple, distributed, local state machines, and hence is scalable to future, communication-dominated technologies. DSRE mechanisms will become increasingly important for high performance in large instruction window machines of the future. Mechanisms such as the one presented in this dissertation will provide future, distributed microarchitectures with low-overhead recovery from value mispredictions.

The evaluation on a high-level GPA simulator showed that the processing of the commit tokens, not ALU or network contention, caused the most performance losses in the DSRE mechanism. We evaluated one technique (speculative commit slicing) that achieved 82% of the performance of an oracle predictor, and proposed and evaluated a bottom-up commit graph pre-traversal for hiding parts of the commit graph traversal. The bottom-up commit graph pre-traversal approach did not result in performance improvements large enough to justify its additional hardware complexity.

When evaluated on the TRIPS prototype simulator that accurately models the bandwidth constraints of a prototype processor implementation, ALU and network contention significantly constrained the performance of the DSRE mechanism. We proposed and evaluated mechanisms for overcoming these constraints. These involved adding a second network for carrying operands, and adding extra logic in the execution and data tile pipelines to

expedite commit messages.

We proposed a dependence predictor that works in the distributed, TRIPS environment. We used the dependence predictor for load-store dependence speculation without re-execution, and also used it to drive the commit slicing mechanism with re-execution. Our results showed that the 17% performance improvement we see with re-execution on the high-level simulator reduces to 4.2% on TRIPS prototype simulator, because of the implementation constraints.

The selective re-execution mechanism that we propose has a cost associated with it, both when the speculation is correct and when it is incorrect. When the speculation is correct, we need to send null commit messages that result in extra network and ALU contention. The commit bit propagation delay is another cost associated with correct speculation. The cost associated with incorrect speculation is the extra ALU and network contention generated by the speculative values. Traditional mis-speculation recovery with pipeline flushing has no cost associated with correct speculation, but has a higher cost associated with incorrect speculation due to pipeline flushes. Our results have shown that to achieve high performance, we need to use both recovery mechanisms. Confidence estimators can be used to choose between the recovery mechanism to use for each speculation. High confidence predictions can use traditional mis-speculation recovery, while low confidence predictions can use a DSRE based recovery. We used two types of dependence predictors as confidence estimators for load-store dependence prediction. Such mechanisms will

become increasingly important in future, large instruction window processors with multiple predictors.

Although we focused mainly on load/store dependence speculation, DSRE mechanisms can easily handle other types of value speculation, including value prediction, predicate prediction, and even "physical speculation," executing instructions on ultra-fast or ultra-low-energy ALU that may occasionally produce a wrong answer but has physical benefits in the common case [17]. We did a brief evaluation of last-value prediction to show that DSRE can work with multiple speculation engines concurrently. The following sections discuss some future directions for this research.

8.2 Looking Ahead

The selective re-execution mechanism proposed in this work was evaluated in the context of load-store dependence prediction. However, the nature of the mechanism makes it amenable for use with with other types of data speculation like last-value prediction, stride prediction, and coherence speculation among others.

8.2.1 Closing the Performance Gap

Even with all the proposed enhancements, the oracle policy outperforms selective re-execution with commit slicing by 33.4% on the TRIPS prototype simulator. We found delay in commit propagation to be the reason for this difference in performance. A number of factors contributed to this delay.

The benchmarks that have the largest performance gap have store-load-store dependences. Compiler optimizations in the future will likely remove some load-to-store dependences by register allocating these variables. However, to achieve performance close to oracle, we will need better commit slicing predictors that can identify the exact matching store for a load. The *first-store* predictor that we use for commit slicing is unable to identify the exact matching store, when there are multiple stores in flight to the same address. Loads that conflict with multiple prior stores are forced to send their commit after all prior stores resolve, resulting in unnecessary delay in the commit bit propagation. Dependence prediction will become increasingly difficult in future, large instruction window machines with a large number of loads and stores in flight. The difficulty in predicting load-store dependences, coupled with higher pipeline flush costs, will result in a growing difference in performance between an oracle policy and pure dependence prediction. Using more sophisticated predictors that work in a distributed environment for dependence prediction, will only be able to bridge part of this performance gap. Techniques like selective re-execution, coupled with more sophisticated dependence prediction, will be required to achieve performance close to oracle.

The compiler and simulation infrastructure for the TRIPS prototype is still under development, and is not mature enough to run large programs like the SPEC CPU2000 suite. We used a set of EEMBC benchmarks for our evaluation in this dissertation, which consist of small kernels that are repeatedly executed within a main loop. Hence, particular characteristics of

these programs are amplified due to the repeated execution. For example, the *RAMfilePtr* variable used in a number of EEMBC programs results in a large number of load-to-store and store-to-load dependences. This dependence slows down the propagation of the commit bit. Another example is the presence of a large number of stores to the same address that are part of different loop iterations. We hand-optimized one benchmark, *aiifft01*, and showed how DSRE can yield large performance improvement on optimized binaries even on the detailed TRIPS implementation. The low variation in the benchmark behavior over the simulated region results in poor performance in these benchmarks. Benchmarks that are more varied in their behavior will show better performance with DSRE.

Finally, the TRIPS compiler that we used to compile the EEMBC benchmarks is still being optimized to produce higher performing binaries. For example, the compiler currently does not have support to register allocate static variables that results in a large number of load-store dependences. The hyperblock generator in the compiler is also being optimized to produce larger hyperblocks. Performance with DSRE will improve with an optimized TRIPS compiler.

8.2.2 Speculative Dataflow Machines?

Branch mispredictions still cause enormous performance losses in high-end processors, and branch predictors are improving with only diminishing returns. While some other architectural proposals advocate moving to a "more

pure" dataflow model [67] that has little control, they merely shift the control dependences to data dependences that must be executed conservatively, both in registers and memory, placing a tight asymptote on achievable parallelism.

DSRE mechanisms can enable a different solution in emerging EDGE architectures—the compiler grows enormous hyperblocks to control-flow graph merge points, which encompass any control flow splits and merges. Within these large, predicated blocks, a predicated producer of a value may choose to execute speculatively and inject its operands to the rest of the graph. If the actual needed operand should have been generated on a different path, the correct operand can be re-injected and handled gracefully by the DSRE mechanism. The execution of predicates can thus be removed from the critical path by speculating the values of certain predicates, with a low-overhead, DSRE-supported recovery guaranteed if the predicate was mispredicted.

The EDGE architecture model with huge hyperblocks, little explicit control flow, and a fine-grained dataflow ISA, starts to resemble in many aspects past dataflow machines like Monsoon [51], but with one important distinction: the dynamic changing of dataflow arcs can be supported by forwarding values into the DFG speculatively and aggressively, with the DSRE mechanism providing a clean recovery if wrong. Monsoon also had multiple functional units connected by a dynamic network, with each functional unit having a token-store for receiving tokens. This token-store was made explicit in the data flow model to simplify resource management. However, dataflow arcs in Monsoon were fixed, as it did not have support for speculation. EDGE

architectures enable conventional, imperative languages, and data speculation, coupled with distributed selective re-execution, may eventually make dataflow architectures truly competitive by also allowing them to achieve high performance on irregular codes while supporting traditional programming models and languages.

Bibliography

[1] Haitham Akkary and Srikanth Srinivasan. Using perceptron-based branch confidence estimation for speculation control. In *Proceedings of The Tenth International Symposium on High-Performance Computer Architecture*, pages 265–274, December 2004.

[2] Todd Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *International Symposium on Microarchitecture*, pages 196–207, November 1999.

[3] Doug Burger et al. Scaling to the end of silicon with EDGE architectures. *IEEE Computer*, 37(7):44–55, July 2004.

[4] Martin Burtscher and B. G. Zorn. Exploring last n value prediction. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 66–76, Oct 1999.

[5] Martin Burtscher and Benjamin G. Zorn. Hybrid load value predictors. *IEEE Transactions on Computers*, 51(7):759–774, July 2002.

[6] B. Calder, P. Feller, and A. Eustace. Value profiling and optimization. *Journal of Instruction Level Parallelism*, 1:1–6, 1999.

[7] B. Calder and G. Reinman. A comparative survey of load speculation architectures. *Journal of Instruction-Level Parallelism*, 2, May 2000.

[8] Brad Calder, Glenn Reinman, and Dean M. Tullsen. Selective value prediction. In *Proceedings of the 26th International Symposium on Computer Architecture, ISCA-99*, pages 64–74, 1999.

[9] Jichuan Chang, Jaehyuk Huh, Rajagopalan Desikan, Doug Burger, and Guri Sohi. Coherence decoupling: Using sharing speculation and value prediction to improve multiprocessor performance. In *Proceedings of the First Value Prediction Workshop VPW03*, June 2003.

[10] Saugata Chatterjee, Chris Weaver, and Todd Austin. Efficient checker processor design. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 87–97, December 2000.

[11] Ben-Chung Cheng, Daniel A. Connors, and Wen mei W. Hwu. Compiler-directed early load-address generation. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 138–147, December 1998.

[12] George Z. Chrysos and Joel S. Emer. Memory dependence prediction using store sets. In *Proc. of the 25th Annual Int'l Symp. on Computer Architecture (ISCA'98)*, pages 142–153, June 1998.

[13] Weihaw Chuang and Brad Calder. Predicate prediction for efficient out-of-order execution. In *Proceedings of the 17th annual International Conference on Supercomputing*, pages 183–192, November 2003.

[14] Rajagopalan Desikan, Doug Burger, Stephen W. Keckler, and Todd M. Austin. Sim-alpha: a validated execution driven alpha 21264 simulator. Technical Report TR-01-23, Department of Computer Sciences, University of Texas at Austin, 2001.

[15] Dan Ernst and Todd Austin. Practical Selective Replay for Reduced-Tag Schedulers. In *Proceedings of the 2nd Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD-2)*, pages 58–63, June 2003.

[16] Dan Ernst, Andrew Hamel, and Todd Austin. Cyclone: A broadcast-free dynamic instruction scheduler with selective replay. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA'03)*, pages 253–262, June 2003.

[17] Dan Ernst, Nam Sung Kim, Sanjay Pant, Shidhartha Das, Rajeev Rao, Toan Pham, Conrad Ziesler, David Blaauw, Todd Austin, Krisztian Flautner, and Trevor Mudge. Razor: A low-power pipeline based on circuit-level timing speculation. In *Proceedings of the 36th Annual International Symposium on Microarchitecture*, pages 7–18, December 2003.

[18] I. Flores. Lookahead control in the IBM system 370 model 165. *IEEE Computer*, (7):24–38, November 1974.

[19] Freddy Gabbay and Avi Mendelson. The effect of instruction fetch bandwidth on value prediction. In *Proceedings of the 25th International Symposium on Computer Architecture, ISCA-98*, pages 272–281, 1998.

[20] David M. Gallagher, Willliam Y. Chen, Scott A. Mahlke, John C. Gyllenhaal, and Wen mei W. Hwu. Dynamic memory disambiguation using the memory conflict buffer. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages (ASPLOS-VI)*, pages 183–193, October 1994.

[21] Amit Gandhi, Haitham Akkary, and Srikanth T. Srinivasan. Reducing branch misprediction penalty via selective branch recovery. In *Proceedings of The Tenth International Symposium on High-Performance Computer Architecture*, pages 254–264, December 2004.

[22] Scott Hareland, Jose Maiz, Mohsen Alavi, Kaizad Mistry, Steve Walsta, and Changhong Dai. Impact of CMOS process scaling and SOI on the soft error rates of logic processes. *Symposium on VLSI Technology Digest of Technical Papers*, pages 73–74, 2001.

[23] Peter Hazucha and Christer Svensson. Impact of CMOS Technology Scaling on the Atmospheric Neutron Soft Error Rate. *IEEE Transactions on Nuclear Science, Vol. 47, No. 6*, pages 2586–2594, Dec. 2000.

[24] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, Doug Carmean, Alan Kyker, and Patrice Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal Q1*, 2001.

[25] Jaehyuk Huh, Jichuan Chang, Doug Burger, and Gurindar S. Sohi. Coherence decoupling: Making use of incoherence. In *Proceedings of the*

Eleventh International Conference on Architectural Support for Programming Languages (ASPLOS-XI), pages 97–106, October 2004.

[26] Daniel Jiminez and Calvin Lin. Dynamic branch prediction with perceptrons. In *Proceedings of The Seventh International Symposium on High-Performance Computer Architecture*, pages 197–206, January 2001.

[27] Daniel A. Jimnez. Piecewise linear branch prediction. In *Proceedings of the 32nd annual International symposium on Computer Architecture*, pages 382–393, June 2005.

[28] Stefanos Kaxiras and James R. Goodman. Improving CC-NUMA performance using instruction-based prediction. In *Proc. of the 5th International Symposium on High Performance Computer Architecture*, pages 161–170, Jan 1999.

[29] Stefanos Kaxiras and Cliff Young. Coherence communication prediction in shared-memory multiprocessors. In *Proc. of the 6th International Symposium High Performance Computer Architecture*, pages 156–167, 2000.

[30] Stephen W. Keckler, Doug Burger, Charles R. Moore, Ramadass Nagarajan, Karthikeyan Sankaralingam, Vikas Agarwal, M.S. Hrishikesh, Nitya Ranganathan, and Premkishore Shivakumar. A wire-delay scalable microprocessor architecture for high-performance systems. In *Proceedings of the 2003 International Solid-State Circuits Conference*, February 2003.

[31] James B. Keller, Ramsey W. Haddad, and Stephan G. Meier. Scheduler which discovers non-speculative nature of an instruction after issuing and reissues the instruction. United States Patent 6,564,315, May 2003.

[32] R. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, March 1999.

[33] Ilhyun Kim and Mikko Lipasti. Understanding scheduling replay schemes. In *Proceedings of The Tenth International Symposium on High-Performance Computer Architecture (HPCA'04)*, pages 138–147, December 2004.

[34] A-C. Lai and B. Falsafi. Memory sharing predictor: The key to a speculative coherent DSM. In *Proceedings of the 26th Annual Int'l Symp. on Computer Architecture (ISCA'99)*, pages 172–183, May 1999.

[35] Chih-Chieh Lee, I-Cheng K. Chen, and Trevor N. Mudge. The bi-mode branch predictor. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 4–13, December 1997.

[36] J. K. L. Lee and A. J. Smith. Branch prediction strategies and branch target buffer design. *IEEE Computer*, 17(1), Jan 1984.

[37] Kevin M. Lepak and Mikko H. Lipasti. On the value locality of store instructions. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 182–191, 2000.

[38] Peter Liden, Peter Dahlgren, Rolf Johansson, and Johan Karlsson. On Latching Probability of Particle Induced Transients in Combinational

Networks. In *Proceedings of the 24th Symposium on Fault-Tolerant Computing (FTCS-24)*, pages 340–349, 1994.

[39] Mikko H. Lipasti and John Paul Shen. Exploiting value locality to exceed the dataflow limit. *International Journal of Parallel Programming*, 26(4):505–538, 1998.

[40] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. Value locality and load value prediction. In *Architectural Support for Programming Languages and Operating Systems*, pages 138–147, 1996.

[41] Gabriel H. Loh and Dana S. Henry. Predicting conditional branches with fusion-based hybrid predictors. In *Proceedings of the 11th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 165–176, September 2002.

[42] Pedro Marcuello, Jordi Tubella, and Antonio Gonzalez. Value prediction for speculative multithreaded architectures. In *Proceedings of the 32nd International Symposium on Microarchitecture, MICRO-32*, pages 230–236, Nov 1999.

[43] Scott McFarling and John Hennessy. Reducing the cost of branches. In *Proceedings of the 13th annual International symposium on Computer Architecture*, pages 396–403, June 1986.

[44] Amit A. Merchant, David J. Sager, and Darrell D. Boggs. Computer processor with a replay system. United States Patent 6,163,838, December

2000.

[45] Amit A. Merchant, David J. Sager, Darrell D. Boggs, and Michael D. Upton. Computer processor with a replay system having a plurality of checkers. United States Patent 6,094,717, July 2000.

[46] Andreas Moshovos, Scott E. Breach, T. N. Vijaykumar, and Gurindar S. Sohi. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 181–193, June 1997.

[47] S. S. Mukherjee and M. D. Hill. Using prediction to accelerate coherence protocols. In *Proc. of the 25th Annual Int'l Symp. on Computer Architecture (ISCA'98)*, pages 179–190, June 1998.

[48] Ramadass Nagarajan, Sundeep K. Kushwaha, Doug Burger, Kathryn S. McKinley, Calvin Lin, and Stephen W. Keckler. Static Placement, Dynamic Issue (SPDI) Scheduling for EDGE Architectures. In *13th International Conference on Parallel Architecture and Compilation Techniques*, pages 74–84, October 2004.

[49] Ramadass Nagarajan, Karthikeyan Sankaralingam, Doug Burger, and Stephen W. Keckler. A design space evaluation of grid processor architectures. In *Proceedings of the 34th International Symposium on Microarchitecture*, pages 40–51, December 2001.

[50] Ramesh Panwar and Ricky C. Hetherington. Appartus for executing coded dependent instructions having variable latencies. United States Patent 5,987,594, November 1999.

[51] G.M. Papadopoulos and D.E. Culler. Monsoon: an explicit token-store architecture. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 28–31, May 1990.

[52] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th International Symposium on Microarchitecture, MICRO-34*, pages 294 – 305, Dec 2001.

[53] Nitya Ranganathan, Ramadass Nagarajan, Doug Burger, and Stephen W. Keckler. Combining hyperblocks and exit prediction to increase front-end bandwidth and performance. Technical Report TR-02-41, Department of Computer Sciences, The University of Texas at Austin, Austin, TX, September 2002.

[54] Steven K Reinhardt and Shubhendu Mukherjee. Transient Fault Detection via Simultaneous Multithreading. In *International Symposium on Computer Architecture*, pages 25–36, July 2000.

[55] Glenn Reinman and Brad Calder. Predictive techniques for agggressive load speculation. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture, MICRO-98*, pages 127–137, Dec 1998.

[56] Eric Rotenberg. AR/SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. In *International Symposium on Fault-Tolerant Computing*, pages 84–91, 1998.

[57] Eric Rotenberg, Quinn Jacobson, Yiannakis Sazeides, and Jim Smith. Trace processors . In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 138–148, December 1997.

[58] Eric Rotenberg, Quinn Jacobson, and James E. Smith. A study of control independence in superscalar processors. In *Proceedings of The Fifth International Symposium on High-Performance Computer Architecture (HPCA'99)*, pages 115–124, January 1999.

[59] Amir Roth and Gurindar S. Sohi. Register integration: a simple and efficient implementation of squash reuse. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 223–234, December 2000.

[60] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Stephen W. Keckler, Doug Burger, and Charles R. Moore. Exploiting ilp, tlp and dlp with the polymorphous trips architecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 422–433, June 2003.

[61] Yiannakis Sazeides and James E. Smith. The predictability of data

values. In *Proceedings of the 30th International Symposium on Microarchitecture, MICRO-30*, pages 248–258, Dec 1997.

[62] Timothy Sherwood, Erez Perelman, and Brad Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Technique*, pages 3–14, September 2001.

[63] Premkishore Shivakumar, Michael Kistler, Stephen W. Keckler, Doug Burger, and Lorenzo Alvisi. Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 389–398, June 2002.

[64] James E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th annual International symposium on Computer Architecture*, pages 135–148, May 1981.

[65] Avinash Sodani and Gurindar S. Sohi. Dynamic instruction reuse. In *Proceedings of the 24th annual International symposium on Computer Architecture*, pages 194–205, June 1997.

[66] Jared Stark, Paul Racunas, and Yale N. Patt. Reducing the performance impact of instruction cache misses by writing instructions into the reservation stations out-of-order. In *International Symposium on Microarchitecture*, pages 34–43, December 1997.

[67] Steven Swanson, Ken Michelson, Andrew Schwerin, and Mark Oskin. Wavescalar. In *Proceedings of the 36th Annual International Symposium on Microarchitecture*, pages 291–302, December 2003.

[68] Kai Wang and Manoj Franklin. Highly accurate data value prediction using hybrid predictors. In *International Symposium on Microarchitecture*, pages 281–290, Dec 1997.

[69] J. Yang and R. Gupta. Energy-efficient load and store reuse. In *IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 72–75, 2001.

[70] Jun Yang and Rajiv Gupta. Load redundancy removal through instruction reuse. In *International Conference on Parallel Processing*, pages 61–68, 2000.

[71] T. Y. Yeh and Yale N. Patt. Alternative implementations of two-level adaptive branch prediction. In *Proceedings of the 20th annual International symposium on Computer Architecture*, pages 124–134, May 1992.

[72] T. Y. Yeh and Yale N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *Proceedings of the 21st annual International symposium on Computer Architecture*, pages 257–266, May 1993.

[73] Adi Yoaz, Mattan Erez, Ronny Ronen, and Stephan Jourdan. Speculation techniques for improving load related instruction scheduling. In

Proc. of the 26th Annual Int'l Symp. on Computer Architecture (ISCA'99), pages 42–53, May 1999.

[74] Huiyang Zhou, Chao ying Fu, Eric Rotenberg, and Tom Conte. A study of value speculative execution and misspeculation recovery in superscalar microprocessors. Technical report, ECE Department, N. C. State University, January 2000.

Index

Vita

Rajagopalan Desikan was born in New Delhi, India on May 23^{rd} 1977, the son of Desikan Rajagopalan and Radha Desikan. He received a Bachelor of Engineering degree in Electrical and Electronics Engineering from the Regional Engineering College, Trichy, India in May 1999. He entered the graduate program in Computer Engineering at the University of Texas at Austin in Fall 1999. In December 2001, he received a Master of Science degree in Computer Engineering, and subsequently joined the PhD. program in Computer Engineering at the University of Texas at Austin.

Permanent address: 3605 Steck Avenue
Apt. 1050
Austin, Texas 78759

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.