IMPROVING PROCESSOR EFFICIENCY BY EXPLOITING COMMON-CASE BEHAVIORS OF MEMORY INSTRUCTIONS

A Thesis
Presented to
The Academic Faculty

by

Samantika Subramaniam

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
May 2009

IMPROVING PROCESSOR EFFICIENCY BY EXPLOITING COMMON-CASE BEHAVIORS OF MEMORY INSTRUCTIONS

Approved by:

Professor Gabriel H. Loh, Advisor
School of Computer Science
Georgia Institute of Technology

Professor Nathan Clark
School of Computer Science
Georgia Institute of Technology

Professor Milos Prvulovic
School of Computer Science
Georgia Institute of Technology

Professor Hsien-Hsin S. Lee
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Hyesoon Kim
School of Computer Science
Georgia Institute of Technology

Aamer Jaleel, PhD
Intel Corporation

Date Approved: 10 December 2008

To my family, for their unconditional love and support

ACKNOWLEDGEMENTS

I take this opportunity to express my deepest gratitude to Professor Gabriel H. Loh, my thesis adviser for his unwavering support, immense patience and excellent guidance. Collaborating with him has been a tremendous learning experience and has been instrumental in shaping my future research career. I will especially remember him for his endless energy and superb guidance and thank him for continually bringing out the best in me. I thank Professor Milos Prvulovic for helping me throughout this journey, by being an excellent teacher and a helpful co-author as well as for providing me very useful feedback on my research ideas, presentations, thesis and general career decisions. I thank Professor Hsien-Hsien S. Lee for his excellent insights and tips throughout our research collaboration. He has always been a wealth of information to me and has helped me become a better researcher. I thank Professor Hyesoon Kim for her invaluable suggestions during my thesis proposal, dissertation writing and for always responding promptly to my requests. I thank Professor Nathan Clark for agreeing to serve on my thesis committee and for his feedback to help improve my work. I thank Aamer Jaleel from Intel Corporation for taking the time to be the external member on my thesis committee.

I would like to thank my friends, Kiran, Guru, Ioannis D, Rahul, Jon, Yuejien, Richard, Mrinmoy, Dong Hyuk, Dean, Gopi, Ioannis S, Carina and countless other friends and colleagues that I have met at Georgia Tech and who have supported me throughout. I thank Susie McClain, Deborah Mitchell, Becky Wilson and Alan Glass for their assistance in administrative matters and for providing me guidance throughout my time at Georgia Tech. I thank the various program committees and reviewers for their constructive feedback on my research papers. I thank Intel Corporation for research equipment, awarding me a fellowship to pursue my graduate studies and for hiring me as an intern with the MTL group in Santa Clara. This experience contributed greatly to my research career. I would also like to thank Georgia Tech and the College of Computing for giving me the opportunity and

the infrastructure to pursue my graduate studies at this prestigious university.

I thank my parents, Mohana and Seetharam Subramaniam for being my pillars of strength throughout this journey. I thank my sister Menaka, my brother-in-law Vikram, my niece Kavya, my grandmother Visalakshi, my grandaunt Jayalakshmi, my mother-in-law Mangala and other family members for being tremendous sources of support and encouragement. I thank my late grandfather, Ramachandran, for his support and blessings all my life. Finally I would like to thank Sankar, my husband, for his endless love and support throughout my graduate career.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Thesis statement: Processor efficiency can be improved by exploiting common-case and predictable behaviors of memory instructions.

The main contribution of this dissertation is exploring the impact of, and improving memory instruction processing to improve the efficiency of the processor. In particular, we propose and evaluate five techniques that tackle certain inefficiencies that exist in memory instruction processing units of conventional processors. Our proposals leverage certain commonly-occurring behavioral patterns in memory instructions which allow for relatively simple and scalable designs to improve processor efficiency.

One of the main aspects of this thesis is the study of memory dependence prediction. We explore the fundamental concepts in memory dependence prediction and propose, to the best of our knowledge, one of the most high-performing and power-efficient memory dependence predictors. Using the fundamental principles and benefits of memory dependence prediction, we then propose an application of it in simultaneous multi-threaded processors that reduces the resource contention in, and improves the performance of these processors.

In our third proposal, we focus on a useful corollary of the predictability in memory dependences, which is the predictability in memory data forwarding, to simplify, reduce and even eliminate some of the power-hungry and area-inefficient resources required by memory instructions in conventional processors. The designs proposed have low implementation complexity and high scalability.

We then focus our attention on another memory instruction behavioral pattern, that is predictability in instruction criticality and leverage this behavior to improve the performance of the processor as well as reduce its power consumption and area requirements.

Finally we explore out-of-order allocation and deallocation of certain microarchitectural resources used by memory instructions in the processor which allows us to reduce the energy consumption and the area overhead of the processor.

Memory instructions constitute a significant portion of the instructions in traditional programs and as such their processing is very important to the performance, power, area, design complexity and scalability of modern processors. This research presented in this dissertation improves the processing of these instructions so that the above-stated parameters can be optimized and improved, which can lead to the design of highly efficient processors.

CHAPTER I

INTRODUCTION

1.1 Motivation

For some years now, processor designers have known that an increase in the number of available transistors due to technology scaling does not automatically translate into an increase in performance. Similarly, increasing the size or number of critical resources in the processor does not always translate into an equivalent increase in performance. Unfortunately, the power consumption of processors, however, increases exponentially with an increase in device count or structure size. These observations have led both designers and researchers to realize that it is not sufficient to design processors simply with a view to increasing performance, or reducing energy consumption or increasing scalability and so on. Instead, a more balanced or "holistic approach" to designing processors, which delves into and improves the core algorithms that utilize the processor structures, is necessary.

Keeping these observations in mind, we believe that improving processor efficiency works as an excellent goal in processor design. To improve efficiency, one needs to study the processor's critical structures and their associated control logic in order to decide how they can be improved. This process can further be streamlined by focusing on the frequently-accessed and contentious structures as well as the common instructions that access these structures. Memory instructions constitute nearly 40% of all instructions and are generally very important to performance, hence in our research we explore improving processor efficiency by focusing on memory instructions.

1.2 Efficiency of Current Processors

The efficiency of a system describes how well a system produces certain desired effects for a given set of inputs. For a processor, there are a number of desirable effects or metrics that can be used to describe its efficiency: performance, power, area, design complexity, scalability etcetera. Increasing processor efficiency may involve improving any one of these

metrics or simultaneously optimizing more than one of them. For example, increasing the performance achieved or reducing the power consumed can help increase efficiency. On the other hand, reducing the power consumed while reducing design complexity and minimizing performance degradation may also be a desirable goal that improves the efficiency of a processor. Finally, some innovative approaches to improving the processor efficiency metrics may focus on one or two metrics and get improvement in another metric at a negligible cost. For example, reducing power consumed and design complexity may reduce access latency which could also improve the scalability of the processor. Thus, we can use a number of approaches to improve the efficiency of current processors. Any approach used, however, generally requires innovative microarchitectural proposals and designs that can optimize instruction processing and the associated control logic in the processor.

In this dissertation, we focus on optimizing the processing of memory instructions to improve one or more of the stated efficiency metrics which in turn improve the overall efficiency of the processor.

1.3 Memory Instruction Processing

Memory instructions are of two kinds: load instructions, which retrieve data from the memory into the processor, and store instructions, which write data from the registers in the processor to the memory. In this section we briefly describe the potential impact of memory instruction processing on processor efficiency. In particular, we explain certain inefficiencies, which exist in memory instruction processing, that we address in this dissertation.

1.3.1 Conservativeness in Memory Instruction Processing

The first aspect of conservative memory instruction processing that we address in this dissertation is speculative memory execution.

Instruction level parallelism and speculative instruction execution are powerful tools to improve the performance of single-threaded microarchitectures. While branch prediction is a widely adopted and universally accepted technique to speculatively schedule branches as well as instructions following the unresolved branch, the use of instruction level speculation while processing memory instructions has only recently started appearing in mainstream

processor microarchitectures such as the Intel Core 2 Duo [19]. Store instructions write data to memory, hence allowing a store instruction to speculatively update data is not possible without special and non-trivial modifications to be able to undo updates to the data cache and memory. Since store instructions are not usually on the critical path of execution, however, conservative store processing does not significantly degrade performance. A load instruction, on the other hand, retrieves data from the cache which is generally used by subsequent instructions in the program. Being able to retrieve a load's data early can impact the scheduling of several instructions following the load and can thus provide a significant performance improvement. Unfortunately, executing a load before it is known to have resolved all of its dependencies can be dangerous, since the power cost of flushing and re-executing a wrongly-executed load and all of its subsequent instructions can be very expensive. Additionally the wrongly-executed instructions could have wasted critical resources that might have been used by other non-speculative instructions. Past proposals that have explored aggressive instruction level speculation in load instructions have either been limited in their performance benefit or have proposed sophisticated designs which are very complex and inefficient with respect to power and access latency [35, 17]. Due to these reasons most current processors use a conservative approach and only schedule loads when they have resolved all of their dependencies. This could limit the performance of the processor and ultimately impact its efficiency.

The second aspect of conservative memory instruction processing that we address in this dissertation is of instruction criticality, with specific emphasis on load criticality. Most load processing mechanisms treat all loads equally irrespective of how the load impacts subsequent instructions or overall processor performance. This may seem like a simple and reasonable strategy to handle loads, but consider a scenario where a critical shared resource is simultaneously budded on by many load instructions. Age-based priority can resolve the contention but the solution may be sub-optimal, since data fetched by the oldest load (or older loads) may not be needed by any subsequent instruction; i.e., the load's age does not necessarily imply its criticality. Furthermore, providing no criteria to classify loads, means that any load optimization, like speculative load execution described above, or data

forwarding needs to be applied to all loads equally. Not only is this wasteful in terms of processor hardware but also power, since many load optimization implementations require fairly complex and power-hungry designs. This extra burden on the processor resources does not always translate into extra performance, since as explained earlier not all loads matter to processor performance.

Prioritizing loads based on their importance or criticality can streamline execution greatly and provide various performance and power benefits that can help improve processor efficiency. Unfortunately, previous proposals to determine load criticality have been fairly complex to implement and limited in their application viability. Hence, load prioritization has so far been an academic concept that has never really been implemented in hardware.

1.3.2 Scalability Concerns

Load and store instructions are processed in microarchitectural queues known as the load queue and store queue, respectively. These buffers have a considerable amount of control logic associated with them that ensures the correct execution of memory instructions. Unfortunately, this complex control circuitry increases the power consumed and area occupied by the queues. These power-hungry structures are also at risk of becoming thermal hot-spots since they are accessed very frequently in the processor. Finally, since these structures are on the critical path of the processor, their size impacts their access latencies, which can slow down the entire program execution. As a result, it becomes difficult to increase the sizes of these structures beyond a limit without running into power, temperature and latency problems. With trends in the processor industry indicating a shift from high-frequency designs to microarchitectures that aggressively target ILP and require large buffers, as is evidenced by Intel's "Nehalem", scalability of the load queue and the store queue is likely to become a vital problem [33]. Even for current microarchitectures, the load and store queues are structures that cause considerable power concerns. Unless this scalability challenge is resolved, it could become difficult to improve or even maintain the efficiency of the processor.

As explained above, the LQ/SQ scaling problem is primarily due to the large buffer sizes that are required to expose ILP and improve performance. One of the reasons for needing large queues, however, is the inefficient processing of memory instructions in these queues. A classic example of this inefficiency is that a load instruction enters the LQ as soon as it is allocated and leaves only when it is committed; but the load instruction uses the LQ resources and control circuitry only when it executes and fetches its data. So, for a large portion of the load's lifetime, it is sitting in the large and power-hungry LQ not even utilizing it. Stores face a similar situation. Techniques that can reduce the amount of time that loads and stores spend in the LQ and SQ , respectively, without impacting performance, are useful to explore in this context. These techniques can reduce the hardware and power required by the LQ/SQ and utilize this in some other part of the processor thereby improving overall processor efficiency. There have been a few proposals that are geared toward building smaller load and store queues [75], but they have fairly high design complexity and hence the scalability of these queues still remains a challenge.

1.3.3 Multi-threaded Microarchitectures

The problems described above have been explained in the context of single-threaded machines. With multi-threaded machines, the problems can be compounded. These machines generally have larger-sized resources and more complex circuitry that single-threaded out-of-order machines, and so their power budgets are already constrained. Additionally, since many of the resources need to be fairly shared and managed across all threads, the performance per thread can sometimes be degraded. In such situations, a conservative approach to scheduling memory instructions can be quite undesirable. There is another challenge associated with multi-threaded architectures. While there are considerable resources available to the processor overall, the manner in which they are partitioned between the threads is not always optimal. Fairness is often a dominant parameter in deciding the allocation of resources, which does not always translate into efficient resource utilization. This could limit the throughput provided by the processor , which in turn could adversely impact the efficiency of the processor.

1.4 Memory Instruction Behaviors

The previous section described some of the inefficiencies in the processing of memory instructions that could limit the potential of current and future microarchitectures. While these challenges do pose serious design issues, there is scope and potential for improvement. In this section, we will briefly walk through some specific behavioral properties of memory instructions and their interactions and provide interesting insights into how their processing can be improved. A more detailed study of the following properties is presented in the next chapter.

1.4.1 Predictability in Memory Dependencies

Memory disambiguation is a technique employed by high-performance, out-of-order processors that execute loads and stores out of program order. Speculative memory disambiguation allows the processor to disambiguate memory instructions even when there is the possibility of a dependence between the instructions. A memory dependence between a store and a following load occurs if the store instruction writes a data value to the same address that the load instruction reads from. Since address calculation is performed at execute, the address of an older store instruction may be unresolved at the time that a later load instruction is waiting to be scheduled. Conservative techniques employed in many processors do not use speculative memory disambiguation since the recovery for a misspeculated load involves a pipeline flush, which increases the power consumption and wastes useful processor resources thereby hurting processor performance. Analysis of memory address patterns show, however, that the instances of true dependencies follow a very predictable pattern. Thus **memory dependence prediction** can be employed to speculatively execute a load instruction and its dependent instructions while minimizing the need for recovery.

1.4.2 Predictability in Data Forwarding

Most of the complex and power-hungry control logic associated with the load and store queues is used to support the memory disambiguation and data forwarding functions necessary for memory instruction processing. Data forwarding is a mechanism by which load

instructions retrieve values that have not yet been written to the cache. Store instructions that have executed and have a value ready, need to forward this value to a younger load instruction reading from the same address, so that the load does not retrieve stale data from the memory. The alternative to data forwarding is to stall a load access until the matching older store does write to memory; this obviously increases the execution time which is very undesirable and thus data forwarding is employed in nearly all modern out-of-order processors. Determining when data need to be forwarded requires associative searches of the store queue. This leads to the increased power consumption of the load and store queue, which as explained earlier, poses a scalability challenge. Analysis of data forwarding patterns show, however, that the data forwarding pairs (store-load) are very predictable. This insight can be used to facilitate **data forwarding prediction** and completely eliminate the associative searches, simplifying the hardware considerably while not hurting the performance.

1.4.3 Predictability in Instruction Criticality

The earlier observations dealt mainly with how memory instructions interact with one another. Even when considered independently, however, memory instructions have certain characteristic behaviors that can be used to improve their processing. Certain load instructions have a more pronounced effect on the performance of the processor than others. Being able to accurately identify these critical loads can open up various avenues of optimizations that are applied based on the priority of the load.

The main concern with load criticality is defining the set of heuristics or criteria that are used in classifying critical loads. Past proposals have been varied and diverse in their approach to defining and classifying critical load instructions. In our research we choose a heuristic that is an accurate indicator of load criticality and is also very predictable. This allows us to design and implement a simple **load criticality prediction** algorithm. The second important concern with load criticality is selecting optimizations that can benefit from it. Since loads have the potential to significantly impact many instructions following them, optimizations that prioritize some loads at the cost of others need to be carefully explored and evaluated. Mispredictions in load criticality can have severe effects since they

are hard to track and correct unlike branch or memory dependence mispredictions. The next chapter describes in detail the properties of critical loads as well as the benefits of being able to accurately predict them. The next chapter also briefly outlines how we apply load criticality information to help improve processor efficiency.

1.4.4 Conservative Allocation and Deallocation Policies

Another interesting observation regarding memory instructions (this can apply to other instructions, too) is that the utility of an instruction or the number of cycles when it is required to perform useful work in the processor (for example forwarding a value, helping to maintain in-order semantics), may be far less than the actual time that the instruction spends in the critical structures or the out-of-order core of the processor. As a result, memory instructions often sit idle occupying critical processor resources, which requires larger buffers to be built to avoid stalls. This contributes to wastage of critical resources as well as power since larger structures have longer wires and consume more power. Being able to determine the utility of an instruction can help streamline the utilization of processor resources and reduce the inefficiency in the design of the processor.

Specifically, the allocation and deallocation of LQ and SQ resources for memory instructions is done fairly conservatively in the pipeline in order to ensure correctness of the system. This approach leads to inefficient LQ and SQ designs which also contribute to the scalability challenge explained earlier. Fortunately, there are techniques that can allow **aggressive allocation and deallocation of resources** of memory instructions, while still maintaining correct in-order semantics. Exploiting these techniques can help reduce the amount of time loads and stores spend in the critical queues thereby allowing us to build smaller queues while maintaining processor performance.

1.5 Scope and Overview of the Dissertation

1.5.1 Scope

The preceeding sections have helped to establish two facts. First, inefficiencies exist in current processors with respect to achieved performance and required power and area budgets.

The manner in which memory instructions are processed contribute significantly to these inefficiencies. Second, there are certain properties of memory behavior that can intuitively be used to make the designs of the structures that support memory instruction processing more efficient. In this dissertation, we present designs that exploit these memory behavior patterns to improve the processing of memory instructions. Our techniques improve one or more of the stated efficiency metrics, which can improve the overall efficiency of the processor. We employ a two-pronged approach to achieve our goal.

1. We propose and evaluate optimizations that use predictable or common-case memory behavior to reduce the latency of memory instructions and improve overall resource utilization. These techniques are able to improve the performance of the processor.

2. We propose and evaluate optimizations that simplify, reduce and even eliminate specific hardware required for the processing of memory instructions. We get this hardware and consequent power reduction with negligible loss in performance.

Processor efficiency can also be improved by changing other design parameters such as the number of cores on a chip. A multi-core design can be used to significantly improve processor throughput as is evidenced by Intel's Core2 Duo [19]. For the design methodology used in our research work, we focus on single-core designs and propose optimizations that improve processor efficiency in a single-core environment. The number of cores, however, is orthogonal to the proposed designs that exploit memory instruction behaviors. Hence, our designs can be extended to improve efficiency even in a multi-core scenario. In Chapter 8, we briefly discuss how our techniques can be applied to multi-core processors.

1.5.2 Overview of Dissertation

In this dissertation, we first give a detailed overview on the common-case behaviors that exist in memory instructions in Chapter 2. We focus on the following memory behavior patterns in this thesis: predictability of memory dependences, predictability of load instruction criticality and the conservativeness in allocation and deallocation of critical processor resources. We start with designing an efficient, scalable, and high-performance memory dependence predictor. Chapter 3 provides a detailed description and evaluation of

our proposed memory dependence prediction mechanism. Our second optimization applies accurate memory dependence prediction to improve the efficiency of the fetch engine of a simultaneous multi-threaded processor, which in turn improves the overall throughput. This optimization is described in Chapter 4. We then use predictable memory dependence patterns to help identify predictable data forwarding patterns and use these to eliminate critical, power-hungry hardware in the processor with no loss in performance. Chapter 5 describes and evaluates this optimization. In Chapter 6, we describe load criticality and propose the design of an efficient criticality predictor. We also explore several applications that can exploit this criticality information to improve the overall efficiency of the processor. In Chapter 7, we present our design for aggressive allocation and deallocation for the LQ and the SQ and show how our techniques can be used to build much smaller queues with no loss in performance. Finally, in Chapter 8 we present our conclusions and discuss some extensions to our thesis that can help to further improve processor efficiency.

CHAPTER II

AN OVERVIEW OF MEMORY INSTRUCTION BEHAVIOR

2.1 Speculative Memory Disambiguation and Memory Dependencies

Out-of-order load execution can significantly improve performance by exposing more instruction level parallelism (ILP). Unfortunately, out-of-order load scheduling and execution is a non-trivial problem. Load instructions may have data dependencies through memory where an earlier store instruction writes to an address and the load instruction reads from the same memory location. The effective addresses of all load and store instructions may not be available because the corresponding address computations may not have executed. This leads to a dependence ambiguity: depending on the result of a store's address computation, a later load instruction may or may not actually be data dependent. If the load is actually independent, then it should be scheduled for execution to maximize performance. If the load is data-dependent, then it must wait for the store instruction.

Consider the code example as shown in Figure 1. Here, the store instruction writes to the memory location specified by the value in register R1, and the load instruction reads the memory location specified by the value in R2. A dependence checking scheme based purely on register dependencies would consider these instructions to be independent. Since both of the instructions access memory, however, the microprocessor cannot statically determine, prior to execution, if the memory locations specified by both instructions are different or are in fact the same. The apparent confusion or ambiguity is because the locations that the memory instructions read/write to depend on the values contained in registers R1 and R2. If the locations are different, the instructions are independent and can be successfully executed out of order. However, if the locations are the same (the address stored in R1 is equal to the address stored in R2), then the load instruction is dependent on the store to produce its value. Memory disambiguation is the technique used to resolve the dependencies between memory instructions in a processor.

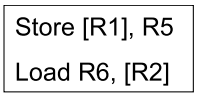


Figure 1: Example showing the ambiguous dependence that can arise between memory instructions.

Like many other properties in microprocessors, memory dependencies exhibit a form of temporal locality. Memory dependence prediction is a technique for speculatively disambiguating the relationships between loads and stores [50]. A load instruction that issues too early due to a dependence misspeculation, however, may retrieve a stale value from the data cache and propagate this incorrect value to its dependent instructions. Many cycles may pass between when the load instruction issues and when the processor finally detects the ordering violation. At this point, a large number of instructions from the load's forward slice [89] may have already executed. Tracking down and rescheduling all of these instructions is a difficult task, and so a memory dependence misspeculation is typically handled by flushing the pipeline. Overly aggressive load scheduling combined with the high cost of pipeline flushes can therefore result in a net performance decrease, power increase and overall reduced efficiency.

To avoid costly pipeline flushes and improve processor performance, load and store execution history can be incorporated in memory dependence prediction. Since memory dependences follow a predictable pattern, this approach improves the accuracy of the predictor. Most memory dependence predictors use history information to predict the *dependence status* of a load: whether a dependence currently exists or not, which determines if the load can issue. The Alpha 21264 employed such a dependence predictor called the *Load Wait Table* that tracks all loads that experience ordering violations [35]. When a store executes and exposes a load ordering violation, the load's PC indexes into the Load Wait Table and sets a bit. On subsequent instances of the load, the Load Wait Table will indicate that the load previously caused a memory ordering violation, and the scheduler will force the load to wait until all earlier store addresses have been resolved. The processor periodically clears the table to avoid permanently preventing loads from speculatively issuing and to allow the predictor to adapt to phasic application behavior in different parts of the program. The

Intel Core 2 has a similar mechanism for tracking misspeculated loads that should be forced to wait for previous stores [19]. The memory dependence predictor used is comprised of a table of saturating counters which provides more accuracy than the 1-bit entry table used in the Alpha 21264.

Memory dependence prediction is useful for several reasons. As described above, it can help issue load instructions early which can potentially schedule instructions dependent on the load early as well. This can reduce the overall execution time thus improving efficiency. In our research work, we design an advanced and novel memory dependence predictor that improves processor performance while providing a simple and efficient hardware implementation. Our proposed design is called **Store Vectors** and is described in Chapter 3.

Memory dependence prediction can also be used to manage the resources of the processor more efficiently. Consider a load that is known to have a memory dependence with a store that has not issued, and will not be ready to issue for 100 cycles. The load has to wait for this store to issue and get its data but the load does not need to hold up processor resources while waiting for this store. Since the load is not going to issue until the store does, the load's dependent instructions cannot issue either and do not need any processor resources as well. These resources can instead be assigned to other independent instructions during the time that the load is waiting. This approach can not only improve the overall throughput of the processor but can also reduce the required hardware resources. In our research work, we apply memory dependence prediction to improve resource utilization in the fetch engine of an SMT processor which improves the processor performance and overall efficiency. Our proposed design is called **PEEP** and is described in Chapter 4.

2.2 Load and Store Queues and Data Forwarding Patterns

Memory operations occur very frequently in most applications (one out of every three or four instructions), and therefore a high-ILP processor must have large structures to buffer all of these loads and stores. The load and store queues buffer these instructions, and they also perform the additional tasks of memory disambiguation and store-to-load value forwarding which is also known as data forwarding. Standard implementations of these

queues require a large amount of fully-associative search logic (i.e. content addressable memories or CAMs) that end up limiting the queue sizes for a given clock frequency or power budget. Furthermore, the CAMs consume a large amount of power which affects the thermal design power (TDP) of the overall processor. More detail on the exact operations of the queues and their implementations will be provided later in this dissertation.

There is an ironic aspect in the difficulty of scaling the load and store queues in that, for a large fraction of memory instructions, the costly CAM logic will not actually result in hits/matches. That is, true instances of memory dependencies or actual occurrences of data forwarding are fairly infrequent. Even for very large instruction windows, many stores end up writing to the data cache before any loads from the same address are ready to receive the data, implying that the data-forwarding mechanism of the store queue is underutilized. As a result, many load instructions search the store queue for an earlier write to the same address, but do not find a match. These loads end up retrieving their data values from the data cache, and do not effectively make use of the store queue's CAMs. The processor includes all of this associative logic to make sure that loads and stores execute correctly, but it slows down the common case of no forwarding.

A useful corollary to the predictability in memory dependencies is that data forwarding patterns are also very stable. That is, a static load that receives a value from a static store, usually receives a value from the same store during all of its dynamic instances. This stable relationship was also quantitatively demonstrated by Tyson et al [86]. This observation coupled with the infrequency in data forwarding occurrences means that a prediction mechanism, similar to memory dependence prediction, can be designed to facilitate data forwarding prediction. Much of the hardware from the memory dependence predictor can be reused along with a mechanism that verifies the prediction. Such an approach allows us to reduce the inherent complexity in the load and store queues while maintaining all their required functionality. In our research work, we design an efficient data forwarding prediction mechanism that removes all of the CAM-logic from the load queue and completely eliminates the store queue while maintaining all required functionality and incurring no loss in performance. Our proposed design is called **Fire-and-Forget** and is described in

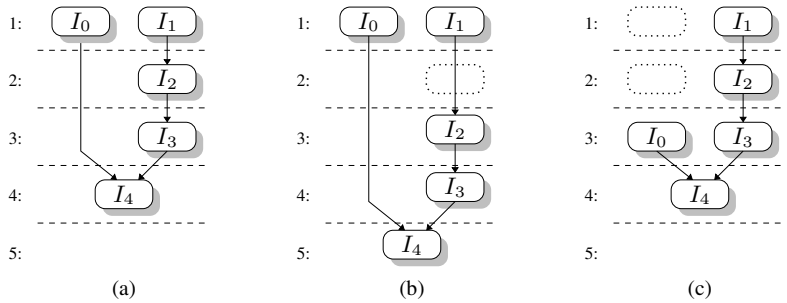


Figure 2: Dataflow graph that depicts how delaying the execution of some instructions increases the critical path.

Chapter 5.

2.3 Instruction Criticality and Criticality-Aware Load Execution

The sensitivity of overall processor performance to the execution latencies of different instructions may vary for many reasons. In a modern out-of-order processor, the hardware dynamically reschedules instructions and can execute multiple instructions per cycle. For some instructions, any increase in the instruction's execution latency, even by a single cycle, will result in a corresponding increase in overall program execution time. Such instructions are said to be on the program's critical path of execution (or the instruction is simply called critical). For other instructions, one may be able to increase the latency of execution (or defer the start of execution) by one or more cycles without impacting performance.

Criticality can be described with a program's data-dependency graph; there exists one or more paths through the graph with a maximum length and such paths are called *critical paths*. Any instruction that appears in this path (or paths) is critical. Consider the dataflow graph shown in Figure 2(a). If each instruction takes a single cycle to execute, then the longest path through this graph (the critical path) has a length of four, starting from I_1 and ending at I_4. Therefore, all four instructions I_1 to I_4 are critical instructions. For example, Figure 2(b) shows that when I_2's execution is delayed by a cycle, then length of the longest path increases. In contrast, instruction I_0 is not critical because we can delay its execution (or increase its execution latency) and the critical path length remains unchanged as shown in Figure 2(c). Note that at some point, if we delay I_0 by enough cycles (in this case more than two cycles), it can form a new longest path and therefore become critical. Non-critical

instructions exhibit some amount of *slack*, which is the amount of delay tolerable before becoming critical [22]. Critical loads are load instructions that lie on the critical path of execution. Since load instructions constitute a large chunk of the program code and they often have long latencies due to cache misses, they could have a significant impact on the performance of the processor.

Being able to identify critical loads can be useful since the processing of these loads can be prioritized. Additionally load optimizations can be stream-lined to focus on critical loads since these loads are known to have the most significant impact on performance. Since load instructions frequently access various important structures in the processor, optimizing their processing can help to improve the efficiency of the processor. In our research work, we design a load criticality predictor that can identify critical loads. We also explore and evaluate some relevant applications where load criticality information can help improve the performance as well as reduce the power consumed and area required by the processor. Our proposed load criticality predictor or **LCP** and its associated applications are described in Chapter 6.

2.4 Allocation and Deallocation Constraints Imposed on Load and Store Queues

As explained in the previous chapter, the number of cycles that an instruction spends in the out-of-order core may be much more than the number of cycles that it is required to. This disparity or inefficiency in the usage of resources for memory instructions is primarily due to rigorous ordering constraints imposed on these instructions during the allocation and deallocation phases. In a conventional processor, allocation is an "all or nothing" process. An instruction that successfully allocates all of its necessary resources (reorder buffer (ROB) entry, reservation station (RS) entry, and load queue (LQ) or store queue (SQ) entry if the instruction is a load or store, respectively) is permitted to proceed onward into the out-of-order execution engine. If even a single resource cannot be obtained, then allocation for this *and all subsequent* instructions stalls. This makes sense in many scenarios because, for example, if an instruction fails to allocate a ROB entry (because the ROB is full), then there is no point in trying to allocate any following instructions because it can

be guaranteed that they will not be able to allocate a ROB entry either (as the ROB is still full). There are some allocation scenarios, however, where such a strict constraint can be relaxed. Not all instructions need LQ and SQ entries, and therefore just because one instruction cannot complete allocation due to, say, an unavailable LQ entry, this does not mean that the next instruction would not be able to acquire all of its resources unless it was also a load.

At the other end of the out-of-order core, the commit stage has several responsibilities. The first is that the commit stage must ensure that any externally visible architected (ISA) state gets updated in an order consistent with an in-order, sequential execution of the same code. This typically involves moving an instruction's result from its temporary location in the physical register file (PRF) or the ROB to its externally visible location in the architecture register file (ARF). For stores, this involves writing their results into the cache hierarchy at which point they are externally visible. The second responsibility of the commit stage is the deallocation of resources. Except for the RS (which is deallocated soon after an instruction issues), the commit stage will reclaim an instruction's ROB entry, as well as its LQ or SQ entry if the instruction is a load or store, respectively. Deallocation is bound to commit, since, for example, one generally cannot deallocate a ROB entry before the contents of that entry have been written back to the ARF, which in turn only occurs when the instruction commits. There are some deallocation scenarios, however, that do not have to happen in strict commit order. If an instruction has executed and forwarded its data to all dependent instructions and there is some way to guarantee that this instruction will complete correctly, this instruction no longer needs the resources assigned to it and can make them available for other instructions.

The scenarios described above, which present opportunities for aggressive allocation and deallocation of LQ and SQ resources, occur often in modern processors and will be described in more detail later on in this dissertation. In our research work, we propose relaxing the constraints on in-order allocation and deallocation of memory instructions to reduce the microarchitectural resources required by the processor while maintaining correctness as well as processor performance. Our proposed design is called **DEAD** and is described in

Chapter 7.

2.5 Applying Characteristics of Memory Behavioral Patterns to Improve Processor Efficiency

The previous chapter gave an overview of the inefficiencies that exist in current processors with regard to memory instruction processing while this chapter discussed some characteristic and common-case behaviors of memory instructions. The rest of the dissertation will describe, in detail, our proposed techniques that exploit these observations and optimize various structures in the processor responsible for the processing of memory instructions. Each of the designs will focus on optimizing one or more of the efficiency metrics such as performance, power and energy consumption, area overhead, access latency and scalability. The first three proposed designs, Store Vectors, PEEP and Fire-and-Forget, are primarily based on the predictability of memory dependences. The fourth proposal, LCP, exploits load criticality to improve processor performance and overall efficiency while our fifth proposal, DEAD, explores out-of-order allocation and deallocation of LQ and SQ resources to avoid the common-case inefficiencies of in-order allocation and deallocation of loads and stores, thereby improving the efficiency. These five memory instruction processing optimization techniques are focused toward a common goal of improving processor efficiency by exploiting common-case and predictable behaviors of memory instructions.

CHAPTER III

STORE VECTORS: DESIGNING A SCALABLE, EFFICIENT, AND HIGH-PERFORMANCE MEMORY DEPENDENCE PREDICTOR

3.1 Introduction

In this chapter, we focus on the issue of speculative memory disambiguation to improve the performance of current and future out-of-order processors [79]. Predictability in memory dependences is used to facilitate the task of speculative memory disambiguation. Unfortunately, current designs of memory dependence predictors, do not incorporate very sophisticated and accurate prediction mechanisms and are limited in the performance they provide. Advanced, high-performing memory dependence predictors, on the other hand, are not very scalable since they are power-hungry and very complex. In our work, we present the design of a high-performing, yet scalable and simple, memory dependence predictor that out-performs other state-of-the art predictors. We also present a detailed evaluation which shows that the Store Vectors technique for memory dependence prediction provides a significant performance benefit at a modest area and power budget which helps to improve the processor efficiency.

As explained in Chapter 2, speculative memory disambiguation, facilitated by memory dependence prediction, is used to speculatively issue load instructions. The most accurate memory dependence predictors rely on identifying relationships between load instructions and one or more earlier stores that are likely data-flow predecessors. Before a load instruction can issue, it must wait until these predicted dependencies have resolved. With current processor trends indicating a steady increase in the size of the instruction window, the processor needs to predict the memory dependencies for many more loads. Additionally the likelihood of the loads actually being dependent on prior unresolved stores also increases. The identification of these memory dependencies and the communication of their resolutions typically require complex logic. The load-store queue (LSQ), however, is already a

very complex circuit with a full set of CAMs for detecting load-store ordering violations as well as supporting store-to-load data forwarding; adding too much more circuitry to support memory dependence prediction may not be practical because of the negative consequences on the critical path latency of the LSQ and the overall processor clock frequency.

Conventional implementations of non-memory instruction schedulers are often CAM-based. The associative logic causes the scheduler to be a timing critical path [55]. Dependency-vector/matrix scheduler organizations have been proposed for faster and more scalable implementations [28, 11, 62]. We propose a new memory dependency prediction and scheduling algorithm based on this idea of dependency vectors [79]. The rest of the chapter is organized as follows: the next section discusses prior work on memory dependence prediction, and in particular it reviews the Store Sets algorithm in greater detail. Section 3.3 explains our proposed Store Vector approach, providing a step-by-step description and example of the technique. Section 3.4 presents the performance results. Section 3.5 proposes and evaluates additional optimizations of the baseline Store Vectors predictor to use significantly less hardware. Section 3.6 provides more detail on why Store Vectors is able to outperform other proposed memory dependence predictors, especially Store Sets. Section 3.7 briefly describes the implementation complexities associated with these memory dependence predictors. Section 3.8 summarizes the main results of the chapter and presents our conclusions.

3.2 Background

3.2.1 Memory Dependence Predictors

The concept of *memory dependence locality* was introduced by Moshovos [50]. In his thesis work, Moshovos characterized two forms of locality: memory dependence status locality, and memory dependence set locality. Memory dependence status locality makes the observation that when a load experiences a store dependency, subsequent instances of the same load will likely experience a dependency as well. Status locality makes no statement about *which particular store(s)* a load may or may not be dependent on. Memory dependence set locality makes the observation that if a load is dependent on a set of store instructions, then

future instances of that load will likely be dependent on the same set of stores. The basic dependence predictors do not attempt to exploit the dependence set locality property. The most basic predictor is a *naive* or *blind* predictor that simply predicts all load instructions to not have any store dependencies. A blind predictor will never cause a load to wait when store dependencies are not present, but it will always cause a costly misprediction if a dependency exists.

Status-based prediction is a technique where the processor uses past behavior to predict future behavior, and, based on this speculation selects some loads to wait for all prior store address to resolve while allowing other loads to speculatively issue. Dependence status predictors are simple to implement and hence all of the memory dependence predictors which have been implemented in industrial designs are status based predictors. The Load Wait Table predictor employed in the Alpha 21264 and the predictor used in the Intel Core 2 which were described in chapter 2 are examples of dependence status based predictors.

These status-based prediction schemes do not take instruction timing into account. A load that is predicted to have a dependency will wait for *all* previous store instructions before issuing. In practice, the load will only be dependent on one or a few of the previous stores. Even if all of these store dependencies have computed their addresses, the load must conservatively wait until the non-dependent stores have resolved as well. A status-based predictor may correctly predict the status of a load and avoid a pipeline flush, but potential ILP may still be lost from preventing loads from issuing in a timely fashion. Memory dependence predictors based on observing the exact sets of stores that a load collides with have shown to provide better prediction accuracy and overall performance. Moshovos proposed a dependence history tracking technique that identifies load-store pairs known to have memory dependencies [51, 50]. When a load speculatively issues and violates a true dependency with an older store, the processor records the program counters (PC) of both the load and the store in a table. Subsequent executions of the load instruction will check the table to see if it had conflicted with a store in the past. If so, the load will also check the store queue (using the store PC recorded in the table) to see if that store is present. If the store is present in the store queue, then the load must wait until the store

has issued. This load-store pair approach has the advantage that misspeculations can be avoided, but at the same time the dependent loads are not delayed longer than necessary.

Different dynamic invocations of a load may have memory dependencies with different static stores. Moshovos also proposed an extension to load-store pair identification that associates a bounded number of stores with each load [50]. Chrysos and Emer generalized this approach with their Store Sets algorithm that allows one or more load instructions to be associated with one or more stores [17]. Previous work showed that a Store Sets memory dependence predictor provides performance close to that of an oracle predictor.

3.2.2 Store Sets

The Store Sets algorithm groups a load's conflicting stores into one logical group or a *Store Set* [17]. Each Store Set has a unique identifier, called the Store Set identifier (SSID). Each load and each store may belong to one and only one Store Set. Figure 3 shows the hardware organization of the Store Sets data structures. The Store Sets Identification Table (SSIT) is a PC-indexed, tagless table that tracks the current Store Set assignment for each load and each store instruction. The example in the figure shows Stores A and C and Load X belonging to Store Set number 2, while Store B belongs to Store Set number 0. This means that in the past, Load X has had memory ordering violations with Stores A and C. The SSIT combined with proper SSID assignment track the active Store Sets in the program.

To prevent memory ordering violations, a load instruction must wait on any unresolved store instructions that belong to the load's Store Set. To determine if any such stores are present in the store queue (STQ), each store updates the last fetched store table (LFST) which indicates the store queue index of the most recent in-flight store. At dispatch, a load instruction consults the LFST and if an active store is present, a dependency is established between the two instructions. In Figure 3, the most recently fetched store (from the load's Store Set) resides in STQ entry 4 as indicated by the LFST and the dependency is illustrated by the bold arrow. To make a load wait on *all* active stores in its Store Set, all stores within the same set are also serialized. This is represented by the dependency arc between Store A and Store C in Figure 3. As described, each store can only belong to a single Store Set

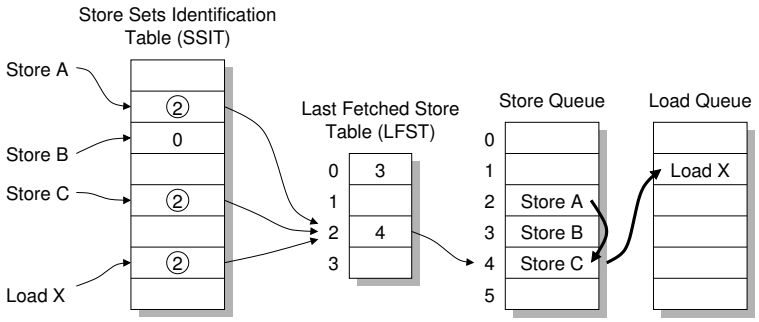


Figure 3: Store sets data structures and their interactions.

determined by the value in the single SSIT entry corresponding to the store. This means that if there are two different loads that are dependent on the same store, then the store will "ping-pong" back and forth between the two loads. To address this problem, Chrysos and Emer proposed a modified SSID assignment rule called Store Sets merging that allows more than one load to share the same Store Set.

The Store Sets algorithm introduces new dependencies from stores to stores, and from stores to loads. These dependencies prevent costly memory ordering violations while allowing independent loads to aggressively issue out-of-order. However, the load and store queues must incorporate new hardware to track and enforce the Store Sets dependencies. The load and store queues are already very complex structures, requiring a large amount of content addressable memory (CAM) logic for the detection of memory ordering violations and to support store-to-load data forwarding [13, 59]. Compared to a non-memory instruction scheduler, the load and store queue CAMs are much larger because they must deal with 64-bit addresses rather than 7-8 bit physical register identifiers. Furthermore, the load and store queues must also employ some form of age or order-tracking information because a load must be able to distinguish between an older (in program order) store and a younger store to the same address. In situations where there exist multiple older stores to the same address, a load needs the age information to make sure that it receives its data from the more recent store.

The original Store Sets work did not clearly describe the exact hardware implementation of the algorithm. Here, we outline two possible approaches. If we implement Store Sets using associative logic, this requires a complete set of CAMs much like the ones already

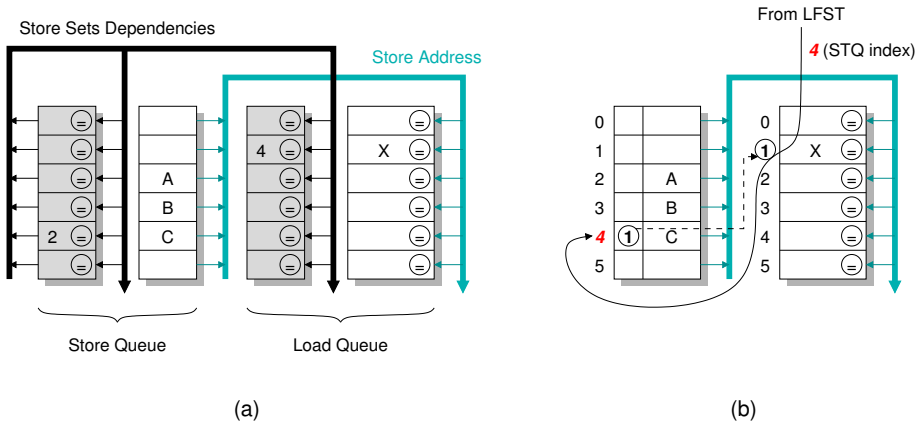


Figure 4: Store Sets implementation using a (a) CAM-based fully-associative organization and (b) a RAM-based direct-mapped organization.

present in the load and store queue and the related broadcast buses to track the Store Sets dependencies, as illustrated by the shaded blocks in Figure 4(a). As described earlier, each store updates its LFST entry with its store queue index. For each subsequent load belonging to this Store Set, the load will read this store queue index from the LFST and store this in the corresponding load queue entry. When the store executes, it broadcasts its store queue index to all of the load queue entries, and any loads waiting on this store will make note of the resolved dependency. Furthermore, the store must also broadcast its store queue index to all of the store queue entries since all stores within the same Store Set will be serialized. The net effect of this approach to implementing Store Sets is the addition of an entire extra set of CAMs to both load and store queues. If multiple loads and/or stores can issue each cycle, then these CAMs will also need to support additional ports. This CAM-based implementation would lead to load and store queues that do not scale well to larger sizes, grossly increase power consumption, and may possibly limit processor clocks speeds as well.

A second potential implementation of Store Sets uses a direct-mapped RAM-based organization, as shown in Figure 4(b). Stores update the LFST with their store queue indexes as usual. When a load (or store) in the same Store Set dispatches/allocates, it first consults the LFST to find the index of the last fetched store (*4* in this example). The load then uses this index to directly access (direct mapped) the store queue entry to insert its own

load queue index (1 in this example). When the store finally issues, it uses this load queue index to directly access the corresponding load queue entry to notify the load that this dependency has now been resolved (shown by the dashed arrow). This approach does not require CAMs, which have high area/power/latency costs, but there are a few additional complications. Store set merging can result in multiple loads all being dependent on the same store. Like the Explicit Data Forwarding (EDF) technique [63], each store queue entry must be able to support waking up multiple dependent loads, which requires storing more than one load queue index. If each store can have up to k dependent loads, then the store's store queue entry must have the capacity to track k load queue indexes, and the load queue itself needs k write ports if the store is to notify all of these loads without incurring any additional delay. If Store Set merging and predictor table aliasing effects cause the store to have greater than k dependents, then either dependence prediction will suffer because some loads cannot be added to the store's output dependency list, or even more complicated schemes (such as allocation of extra store queue entries) will be required. Another complication with this RAM-based implementation of Store Sets is that when a younger load inserts its load queue index into a store queue entry, the load may in fact be a speculative instruction in the shadow of a mispredicted branch. When the processor uncovers the branch misprediction, some recovery mechanism is required to remove the load queue index from the parent store queue entry.

The LFST also creates a new critical loop similar to the register renaming logic. It is possible that multiple stores need to update the LFST in the same cycle. If all of these stores belong to the same Store Set, then some sort of dependency checker is required to correctly set up the intra-group dependencies, and a prioritized write logic is needed to make sure that only the last store in the Store Set updates the corresponding LFST entry. Similar to the wrong-path pointers in the RAM-based Store Sets implementation, the LFST entries may also contain store queue indexes that correspond to wrong-path instructions. The LFST would need support for checkpointing or some other recovery/repair mechanism.

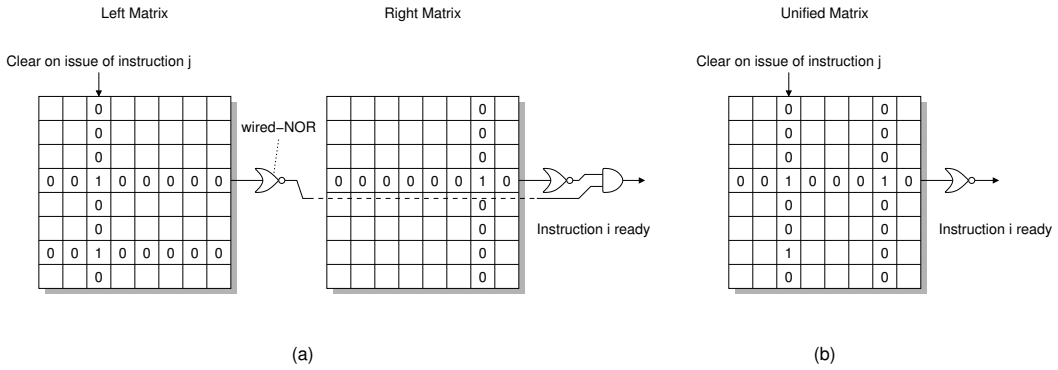


Figure 5: (a) Two-matrix scheduler, (b) single-matrix unified scheduler.

3.2.3 Scheduling Structures

Instruction schedulers typically use content addressable memory (CAM) organizations. Each issuing instruction broadcasts a unique identifier to notify its dataflow children that the dependency has been resolved. Each instruction must monitor all of the broadcast buses, constantly comparing the identifiers of its inputs with the broadcast traffic. Palacharla analyzed the structure of CAM-based schedulers and found that the critical path delay increases quadratically with the issue width and the number of entries [55].

The dependency vector or dependency matrix scheduler organization is an alternative scheduler topology designed to be significantly more scalable. Goshima et al. proposed to replace the left and right CAM banks of a conventional scheduler with left and right dependency matrices as illustrated in Figure 5(a) [28]. For a W-entry scheduler, each matrix has W rows and W columns; one for each instruction. If instruction i is data-dependent on instruction j, then the matrix entry at row i and column j is set to one. So long as instruction i has a bit set in its row, then the corresponding input dependency has not been resolved. When instruction j issues, it clears *all* bits in column j, thus notifying any dependents in the window that the parent instruction has been scheduled. The critical path logic is significantly reduced as compared to a CAM-based scheme. The tag comparison for computing readiness has been replaced by a single wired-NOR and an AND gate to check that both left and right inputs are ready. The multi-bit tag broadcast has been replaced by a single bit latch-clear signal. The matrix structures only contain one bit per entry which makes the total area significantly smaller than a CAM-based scheduler that contains

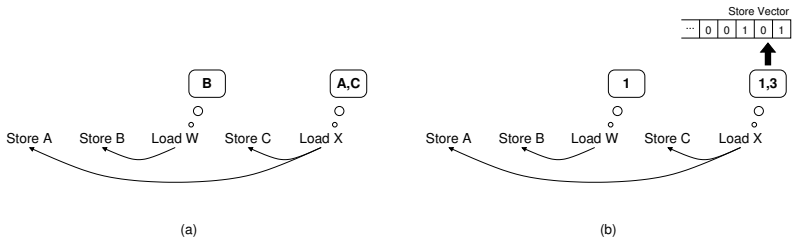


Figure 6: (a) Store-address tracking of dependencies, and (b) store position or age tracking of dependencies.

registers for the dependency tags, comparators, large broadcast buses, and additional logic. Figure 5(b) illustrates a single-matrix implementation, suggested by Brown et al [11]. Each matrix row can contain multiple non-zero entries to denote all of the dependencies, which halves the matrix area and removes the AND gates for detecting both left-and-right input readiness. Our store-vector dependence predictor's hardware implementation is based on this compact, scalable single-matrix scheduler structure.

3.3 Store Vector Dependence Prediction

We propose a new algorithm for memory dependence prediction based on *Store Vectors*. Store vectors are different than the load-store pair and Store Set approaches in that Store Vectors do not explicitly track the program counters (PC) of stores that collide with loads. Instead, we implicitly track load-store dependencies based on the relative *age* of a store. Consider the example in Figure 6. The five load and store instructions are listed in program order, and Load W has had ordering violations with Store B in the past, and Load X has had ordering violations with both Stores A and C. Figure 6(a) illustrates the PC-based dependency information used by previous pair or set-based approaches. In contrast, an age-based approach illustrated in Figure 6(b) stores the relative age or positions of the stores. Load X remembers that it had previous conflicts with the most recent store and the third most recent store. A load's *Store Vector* records the relative positions or ages of all stores that were involved in previous memory ordering violations. The Store Vector for Load X is illustrated in Figure 6(b). In the general case, the length of the Store Vector will be equal to the number of store queue entries although we discuss different ways to optimize Store Vectors to only track a certain number of store dependencies in Section 3.5.

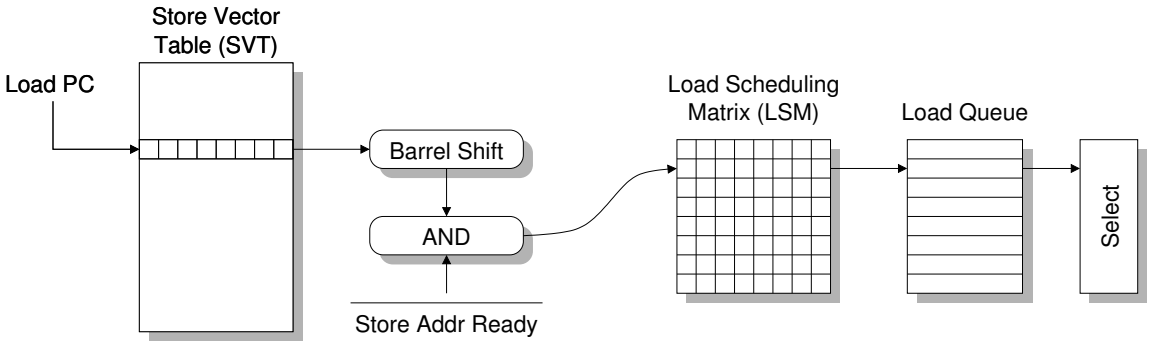


Figure 7: Store vectors data structures and interaction with a conventional load queue.

3.3.1 Step-by-Step Operation

The Store Vector algorithm has three main steps: lookup/prediction, scheduling, and update due to ordering violations. These are described in turn below, followed by an example.

Lookup/Prediction — The primary data structure for recording load-store dependencies is the Store Vector Table (SVT). For each load, the least significant bits of the load's PC provides an index into the SVT, shown in Figure 7. The corresponding Store Vector is then rotated and copied into the load scheduling matrix (LSM or simply the *matrix*). The process for setting a load's Store Vector is described later. The LSM consists of one row for each load queue entry, and one column for each store queue entry (the matrix need not be square).

An SVT entry records a load's Store Vector in a format where the least significant bit corresponds to the most recent store before the load. The rightmost column of the LSM may not correspond to the most recently fetched store. A barrel shifter must rotate the vector such that the least significant bit is aligned with the column of the most recent store. Any bits in the vector that correspond to already resolved stores must be cleared to prevent the load from waiting on an already resolved dependency (which could result in deadlock). This is accomplished by taking a bitwise AND of the Store Vector with the bits from each store queue entry that indicates if the store's address is *not* ready. Finally, the vector is written into the matrix.

Note that for both the Store Sets and Store Vector techniques, the prediction lookup latency can be overlapped with other front-end activities. Both approaches use PC-based

table lookups which could in theory be initiated as early as the fetch stage, although for power reasons one would likely defer the lookup to the decode stage to avoid unnecessary predictions for non-load instructions.

Scheduling — After the prediction phase has written a load's Store Vector into the LSM, there may be some bits in the vector that are set. The position of the bits indicate the stores that this load is predicted to have a dependency with. While any of the bits are still set, the load will not be considered as ready. The hardware implementation of the matrix is identical to the single-matrix scheduler described earlier in Figure 5(b). A wired-NOR determines if any unresolved predicted dependencies remain.

Each store queue entry is uniquely mapped to a column of the matrix. When the store issues, it simply clears all of the bits in its corresponding column. Note that stores in the same vector can issue in any order, whereas with Store Sets all of the stores are serialized. Furthermore, a store can be a predicted input dependence for any number of load instructions (more than one entry per column may be set), whereas with Store Sets the LFST forces each store to belong to only a single Store Set.

Update — Initially, vectors in the SVT are initialized to all zeros. In this fashion, all load instructions will initially execute as if under a naive/blind speculation policy. When a load-store ordering violation occurs, the store's relative position/age is determined. The position of the most recent store with respect to the offending load is easily determined because the processor already keeps this information for tracking the ordering of all load and stores. The difference between the store queue indexes of the most recent store and the store involved in the ordering violation provides the relative age of the offending store. This bit is then set in the load's SVT entry.

Eventually, all bits in the vector may get set, thus making the processor behave as if it was incapable of any memory speculation at all. Similar to the 21264 load wait table and Store Sets, we periodically reset the contents of the SVT to clear out predicted dependencies that may no longer exist due to changes in program phases, dynamic data values, or other reasons.

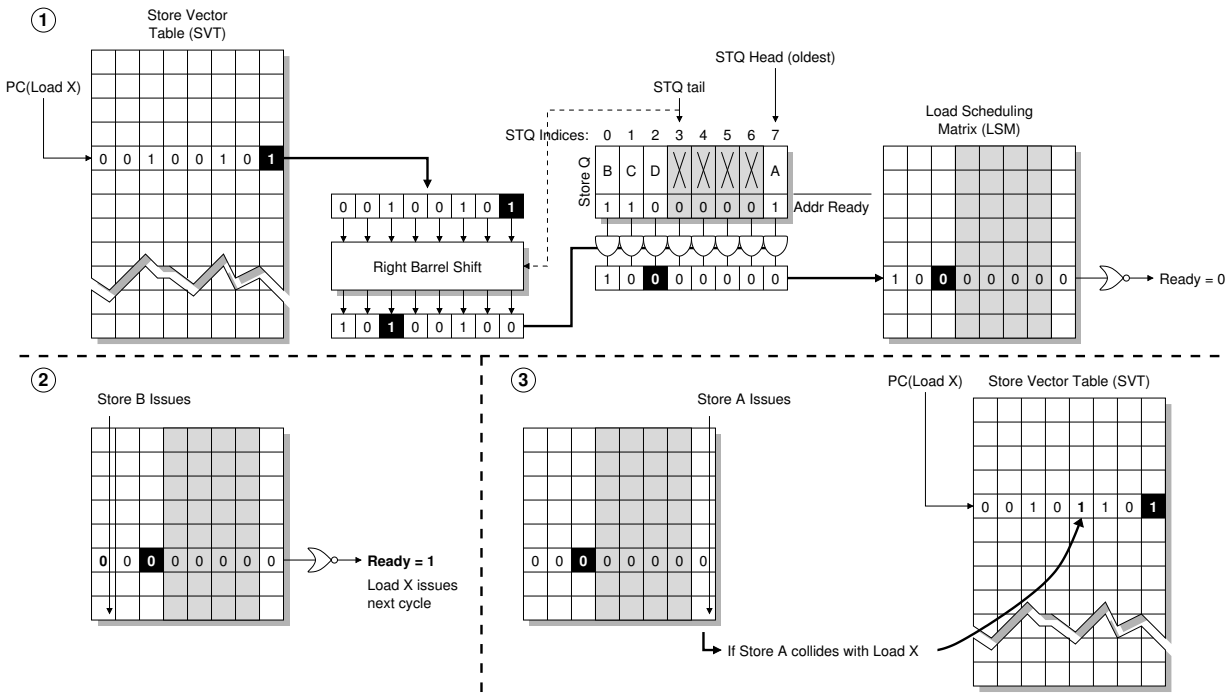


Figure 8: Example Store Vector operation for the ① prediction phase, ② scheduling phase, and ③ update phase.

3.3.2 Example

This sub-section provides a detailed example of predicting a load's dependencies, scheduling the load, and updating the load's Store Vector when an ordering violation occurs. The different steps of the example are illustrated in Figure 8.

① **Prediction:** After decoding the instruction Load X, a hash of the load's PC is used to select a vector from the Store Vector Table. The Store Vector bit that corresponds to the most recent store fetched before this load is indicated with shading. The 1's in the vector indicate that Load X is predicted to be dependent on the most recent store (shaded), the third most recent store, and the sixth most recent store.

In this example, the most recent store is Store D, which has been allocated to store queue (STQ) entry 2. The STQ-head points to the oldest STQ entry, and the STQ-tail points to the next available STQ entry. If we right barrel shift (least significant bits wrap around to the most significant positions) by an amount equal to the STQ-tail, the Store Vector bit for the most recent store will now be properly aligned with the most recent STQ

entry. In this case, the right barrel shift is by three; note that the position of the shaded bit has moved to reflect the location of the most-recent store.

Corresponding to the non-zero bits in load's Store Vector, the most recent store is Store D, the third most recent is Store B, and a sixth most recent does not exist. The Store Vector predicts that Load X is dependent on the most recent store, but at the time of dispatch, Store D had already issued and so Load X should not wait on Store D. Each entry of the store queue has a bit that indicates whether the corresponding store address is unknown. Since the STA (store address) and STD (store data) are two separate uops this bit is cleared when the address of the store is known irrespective of whether the data is ready or not. In the case of invalid entries, the bit is cleared to indicate that the address is known. By taking a bitwise AND, we clear the Store Vector bit that corresponded to the invalid store (the sixth most recent store) as well as a store that had already been resolved (Store D). This final Store Vector is written into the Load Scheduling Matrix in the row specified by the load's Load Queue entry index.

② **Scheduling:** In this example, Load X only has one remaining dependency in its Store Vector. At some point in the future, Store B will issue. At this point, Store B clears its column in the Load Scheduling Matrix. Each STQ entry can simply have a single hardwired connection to the clear signal for the corresponding column. Load X's Store Vector no longer contains any 1's, and so the wired-NOR will raise its output to indicate that Load X's predicted store dependencies have all been satisfied. Assuming that Load X's address has already been computed, it will proceed to bid for a memory port and then issue.

③ **Update:** If Load X issued and the load and store queues do not eventually detect any memory ordering violations, then no other actions are required. If after Load X speculatively issues and Store A ends up writing to the same address as Load X, then there will be a memory ordering violation. In this case, Load X and all instructions afterward are flushed from the pipeline and re-fetched. Store A is the fourth most recent store with respect to Load X, although it may not be the fourth most recent store in the store queue because other store instructions may have been fetched since Load X was dispatched. To prevent

future memory-ordering violations between Store A and Load X, the store updates Load X's Store Vector in the SVT, setting the bit for the fourth most recent store.

3.4 Evaluation

This section presents the performance evaluation of the various memory dependence prediction algorithms.

3.4.1 Methodology

We use cycle-level simulation to evaluate the performance of the memory dependence predictors. In particular, we use the MASE simulator [38] from the SimpleScalar toolset [3] for the Alpha instruction set architecture. We made several modifications related to memory dependence prediction and scheduling, including support for separate load and store queues (as opposed to a unified load-store queue), Store Sets and the Store Vector algorithms. In addition to comparing the performance of Store Sets with Store Vectors, we also simulate the load wait table (LWT) described in Section 3.2.1 since it has low implementation complexity. For LWT, Store Sets and Store Vectors, we reset the predictor tables every one million instructions. The minimum latency from fetch to execution is twenty cycles, and we simulate a full in-order front-end pipeline as opposed to the default SimpleScalar behavior of simply stalling fetch for some number of cycles.

We model three main processor configurations, whose parameters are listed in Table 1, to evaluate the performance of the different memory dependence predictors. The parameters that differ between the configurations are noted in the specific configuration column while all other parameters are common to all the configurations. The 'Medium' configuration represents a moderately aggressive processor similar to current microarchitectures. The 'Large' configuration has larger hardware structures and a wider machine width to represent a more aggressive processor. Finally we also model an 'Extra-Large' configuration, which represents a highly aggressive machine, that exposes more opportunities for load-store dependencies to exist. While the extra-large configuration is definitely more aggressively positioned than current processors and would be difficult to implement using conventional design methodologies, there have been some innovative research proposals that

Table 1: Simulated processor configurations.

Configuration	Medium	Large	Extra-Large
Processor Width	6	8	8
Scheduler Size (RS)	64	96	128
Load Queue Size	32	96	128
Store Queue Size	32	96	128
ROB Size	256	512	2048
Memory Ports	2	4	4
ALU	4 Int, 4 FP		
Mult	1 Int, 1 FP		
IL1 Cache	16KB, 4-way, 3-cycle		
DL1 Cache	16KB, 4-way, 3-cycle		
L2 Cache	512KB, 8-way, 10-cycle		
L3 Cache	4MB, 16-way, 30-cycle		
Memory	250 cycles		
ITLB	64-entry, FA		
DTLB	64-entry, FA		
L2 TLB	1024-entry, 8-way, 7-cycle		

have put forth implementable designs of similar, extra-large configurations [61, 18]. Due to the large buffer structures, the designs presented in these proposals could encounter several load-store dependencies and may benefit by using advanced memory dependence prediction algorithms. Hence we include this configuration design point in the evaluation.

We simulated a variety of applications from the following suites: SPECcpu2000, MediaBench [39], MiBench [29], Graphics applications including 3D games and ray-tracing, Pointer-intensive benchmarks [4] as well as Bioinformatics benchmarks [1]. All SPEC applications use the reference inputs, where applications with multiple reference inputs are listed with different numerical suffixes. To reduce simulation time, we used the SimPoint 2.0 toolset to choose representative samples of 100 million instructions [56]. All average IPC speedups are computed using the geometric mean.

3.4.2 Performance Results

Our baseline processor configuration uses naive/blind speculation. We also use a perfect oracle dependence predictor as an "upper bound." The oracle predictor is perfect in the sense that it correctly predicts which stores a load has true dependences with while ensuring that the load is only made to wait until these specific stores have computed their addresses. The oracle guarantees perfect dependence prediction, but this does not necessarily result in maximum performance because in rare cases misspeculated loads can still have performance

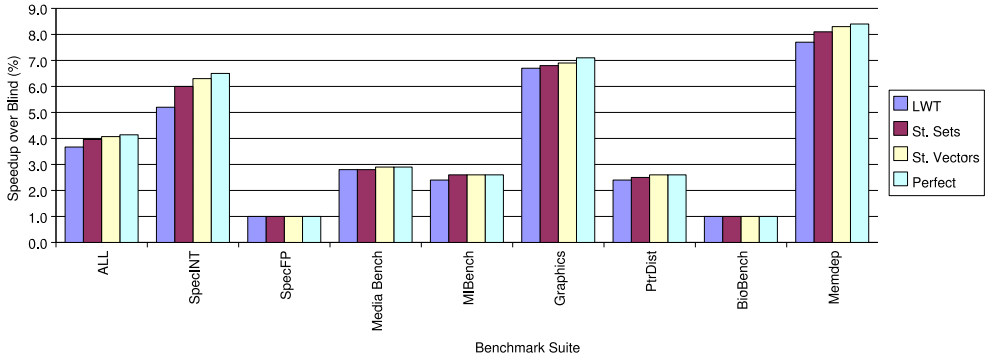


Figure 9: Performance of the 'medium' sized configuration across all benchmark suites. Memdep refers to those applications which were found to exhibit memory dependence-sensitivity.

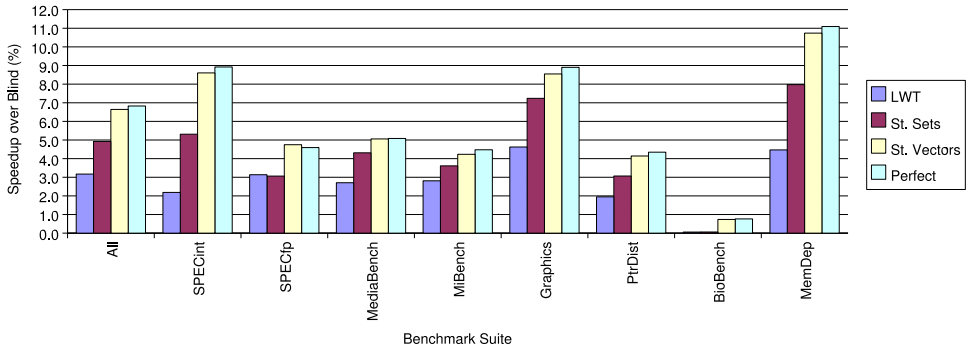


Figure 10: Performance of the 'large' sized configuration across all benchmark suites. Memdep refers to those applications which were found to exhibit memory dependence-sensitivity.

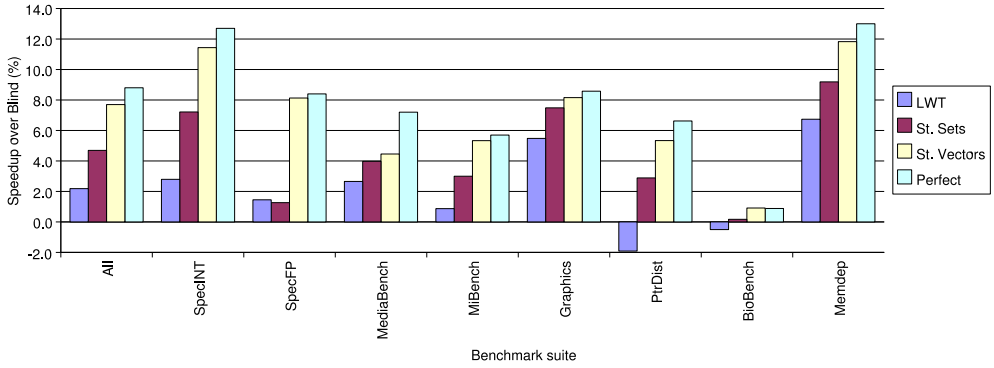


Figure 11: Performance of the 'extra-large' sized configuration across all benchmark suites. Memdep refers to those applications which were found to exhibit memory dependence-sensitivity.

benefits due to prefetching effects or early branch misprediction detection.

Figure 9, Figure 10 and Figure 11 depict the average per-suite performance numbers of the simulated memory dependence predictors for the medium, large and extra-large configurations, respectively. We call an application *dependence-sensitive*, labeled as Memdep in the plots, if perfect dependence prediction results in at least a 1% IPC speedup over blind speculation. Many applications are not sensitive to memory dependences and only experience few (single digits) ordering violations over the course of millions of instructions. We also present the detailed performance results of the various memory dependence predictors for all benchmarks (both dependence-sensitive and non-dependence-sensitive) for the medium configuration in Table 2 and Table 3. The per-application performance numbers for the large and extra-large configurations are presented in the Appendix.

We now discuss the results in detail. For the medium-sized processor configuration, Figure 9 shows that perfect/oracle dependence prediction achieves a 4.1% speedup over blind speculation across all applications, and 8.5% on the dependence-sensitive programs. All of the dependence predictors, including the simple Load Wait Table as well as the more sophisticated Store Sets and Store Vector algorithms, achieve a performance level that is relatively close to that of the perfect predictor. On the dependence-sensitive programs, the LWT's 7.7% performance improvement over blind speculation is quite close to Store Set's 8.1%, Store Vector's 8.4% and the perfect predictor's 8.5% speedups. From these results, we conclude that for moderately-sized microarchitectures, a simple dependence predictor like the Load Wait Table (LWT) is sufficient. Adding a more sophisticated predictor does not provide much additional performance, and this may not justify the added complexity associated with implementing the predictor. Per-application speedup numbers for the medium configuration further show that there is not much variation in the performance of the memory dependence predictors. There are some integer applications like $gcc-2$, however, that suffer quite a significant performance loss over blind prediction when using the LWT predictor due to its conservativeness. In this particular application we observed many missed opportunities for speculative load scheduling.

For more aggressively provisioned processors, the results and conclusions are slightly

Table 2: The performance of the memory dependence predictors simulated on the 'medium' configuration for the entire set of applications. An 'x' signifies that the benchmark is dependency sensitive (> 1% performance change between blind prediction and perfect prediction).

Benchmark Name			Base IPC	LWT	St. Sets	St. Vectors	Perfect
adpcm-dec	M		3.83	0.0%	0.0%	**0.0%**	0.0%
adpcm-enc	M		1.58	0.0%	0.0%	**0.0%**	0.0%
ammp	F		1.32	0.0%	0.0%	**0.0%**	0.0%
anagram	P	x	2.06	11.5%	11.6%	**11.7%**	11.7%
applu	F		0.72	0.0%	0.0%	**0.0%**	0.0%
apsi	F	x	2.14	1.8%	1.8%	**1.9%**	1.9%
art-1	F		1.53	0.0%	0.0%	**0.0%**	0.0%
art-2	F		1.49	0.0%	0.0%	**0.0%**	0.0%
bc-1	P	x	2.17	5.6%	6.0%	**6.8%**	6.0%
bc-2	P	x	1.11	2.9%	3.2%	**3.6%**	3.5%
bc-3	P		1.15	0.2%	0.2%	**0.2%**	0.3%
bzip2-1	I		1.56	0.3%	0.5%	**0.5%**	0.6%
bzip2-2	I		1.59	0.0%	0.1%	**0.0%**	0.1%
bzip2-3	I		1.70	-0.3%	-0.3%	**-0.3%**	0.4%
clustalw	B		3.33	0.3%	0.3%	**0.3%**	0.3%
crafty	I	x	1.24	6.1%	6.3%	**6.2%**	6.3%
crc32	E		2.62	0.0%	0.0%	**0.0%**	0.0%
dijkstra	E		2.15	0.0%	0.0%	**0.0%**	0.1%
eon-1	I	x	1.23	17.0%	19.9%	**18.8%**	19.4%
eon-2	I	x	0.91	3.4%	3.6%	**3.4%**	3.9%
eon-3	I	x	1.00	6.6%	7.1%	**7.4%**	8.1%
epic	M		1.86	0.0%	0.0%	**0.0%**	0.0%
equake	F		0.83	0.0%	0.0%	**0.0%**	0.0%
facerec	F	x	2.06	6.7%	6.7%	**6.7%**	6.7%
fft-fwd	E		1.98	0.0%	0.0%	**0.0%**	0.0%
fft-inv	E		1.98	0.0%	0.0%	**0.0%**	0.0%
fma3d	F	x	0.97	1.9%	1.9%	**2.1%**	2.0%
ft	P		2.12	0.3%	0.3%	**0.4%**	0.4%
g721decode	M	x	1.76	1.5%	1.5%	**1.6%**	1.6%
g721encode	M	x	1.64	1.1%	1.1%	**1.2%**	1.2%
galgel	F		3.31	0.0%	0.0%	**0.0%**	0.0%
gap	I		1.27	0.7%	0.6%	**-0.2%**	0.7%
gcc-1	I	x	2.31	0.6%	0.7%	**1.0%**	1.0%
gcc-2	I	x	0.76	-6.3%	4.3%	**4.1%**	4.8%
gcc-3	I	x	0.82	2.6%	2.6%	**2.7%**	2.8%
gcc-4	I	x	0.96	2.3%	2.5%	**2.5%**	2.6%
ghostscript-1	E	x	1.40	14.7%	15.0%	**15.1%**	14.8%
ghostscript-2	M	x	1.40	15.3%	15.5%	**16.0%**	15.5%
ghostscript-3	E	x	1.67	10.1%	11.2%	**12.5%**	12.7%
glquake-1	G	x	1.04	4.0%	4.1%	**4.2%**	4.3%
glquake-2	G	x	1.17	3.3%	3.3%	**3.5%**	3.6%
gzip-1	I		1.79	0.0%	0.0%	**0.0%**	0.1%
gzip-2	I		1.37	5.0%	0.3%	**0.3%**	0.3%
gzip-3	I	x	1.49	1.1%	1.2%	**1.2%**	1.3%
gzip-4	I		1.80	0.0%	0.0%	**0.0%**	0.0%
gzip-5	I		1.34	0.0%	0.3%	**0.3%**	0.3%
jpegdecode	M		2.12	0.5%	0.5%	**0.7%**	0.7%
jpegencode	M	x	1.72	3.7%	3.7%	**3.8%**	3.7%
ks-1	P		1.22	0.0%	0.0%	**0.0%**	0.0%
ks-2	P		0.78	0.0%	0.0%	**0.0%**	0.0%
lucas	F		1.00	0.0%	0.0%	**0.0%**	0.0%
mcf	I		0.58	0.0%	0.0%	**0.0%**	0.0%

Table 3: Continuation of the data from Table 2.

Benchmark Name			Base IPC	LWT	St. Sets	St. Vectors	Perfect
mesa-1	M		1.49	0.0%	0.0%	**0.0%**	0.0%
mesa-2	F	x	1.50	17.9%	18.0%	**18.2%**	18.4%
mesa-3	M	x	1.06	1.3%	1.3%	**1.3%**	1.3%
mgrid	F		1.26	0.1%	0.1%	**0.1%**	0.1%
mpeg2decode	M		2.26	0.2%	0.2%	**0.3%**	0.4%
mpeg2encode	M		3.12	0.1%	0.1%	**0.1%**	0.1%
parser	I	x	1.26	6.9%	6.9%	**6.9%**	7.1%
patricia	E		1.09	0.1%	0.1%	**0.3%**	0.1%
perlbmk-1	I	x	0.95	10.0%	10.5%	**11.6%**	11.3%
perlbmk-2	I	x	1.22	1.7%	1.8%	**2.4%**	1.9%
perlbmk-3	I	x	2.37	2.5%	2.1%	**2.6%**	2.7%
phylip	B		1.42	0.0%	0.0%	**0.0%**	0.0%
povray-1	G	x	0.76	18.3%	18.6%	**18.7%**	19.1%
povray-2	G	x	0.98	8.8%	8.7%	**9.1%**	9.3%
povray-3	G	x	1.01	20.3%	20.8%	**20.8%**	21.3%
povray-4	G	x	1.03	11.8%	11.8%	**11.5%**	12.0%
povray-5	G	x	0.84	10.7%	10.9%	**11.0%**	11.2%
rsynth	E		1.62	0.1%	0.1%	**0.1%**	0.1%
sha	E	x	2.88	8.8%	9.0%	**8.8%**	9.0%
sixtrack	F		1.30	0.2%	0.2%	**0.2%**	0.3%
susan-1	E		2.27	0.2%	0.2%	**0.2%**	0.2%
susan-2	E		1.64	0.0%	0.0%	**0.0%**	0.0%
swim	F		0.80	0.0%	0.0%	**0.0%**	0.0%
tiff2bw	E		1.51	0.0%	0.0%	**0.0%**	0.0%
tiff2rgba	E		1.63	0.2%	0.2%	**0.2%**	0.2%
tiffdither	E	x	1.59	3.6%	3.7%	**3.7%**	3.9%
tiffmedian	E		2.46	0.1%	0.1%	**0.1%**	0.1%
twolf	I	x	0.83	1.3%	1.2%	**1.3%**	1.3%
unepic	M		0.40	0.0%	0.0%	**0.0%**	0.0%
vortex-1	I	x	1.56	31.0%	30.4%	**31.7%**	32.4%
vortex-2	I	x	1.06	37.6%	37.6%	**41.0%**	39.9%
vortex-3	I	x	1.45	35.0%	35.2%	**36.7%**	37.2%
vpr-1	I	x	1.06	5.0%	5.0%	**5.2%**	5.2%
vpr-2	I	x	0.65	3.5%	3.5%	**3.5%**	3.6%
wupwise	F		2.29	0.4%	0.5%	**0.5%**	0.5%
x11quake-1	G	x	1.44	4.4%	4.4%	**4.2%**	4.7%
x11quake-2	G		2.74	-0.1%	-0.1%	**-0.2%**	0.1%
x11quake-3	G	x	1.42	4.5%	4.5%	**4.3%**	4.9%
xanim-1	G		3.27	0.0%	0.0%	**0.0%**	0.0%
xanim-2	G	x	2.86	0.9%	0.9%	**1.0%**	1.0%
xdoom	G	x	2.06	4.7%	4.9%	**5.3%**	5.4%
yacr2	P	x	2.10	2.5%	2.5%	**2.6%**	2.5%

Benchmark Group		Base IPC	LWT	St. Sets	St. Vectors	Perfect
ALL		1.44	3.7%	4.0%	**4.2%**	4.2%
SpecINT	I	1.22	5.3%	6.1%	**6.3%**	6.5%
SpecFP	F	1.37	0.7%	0.7%	**0.7%**	0.7%
MediaBench	M	1.69	2.8%	2.8%	**2.8%**	2.8%
MiBench	E	1.84	2.4%	2.5%	**2.6%**	2.6%
Graphics	G	1.41	6.7%	7.0%	**7.0%**	7.3%
PtrDist	P	1.49	2.8%	2.9%	**3.1%**	3.0%
BioBench	B	2.17	0.1%	0.1%	**0.1%**	0.1%
Dep. Sensitive	x	1.31	7.7%	8.1%	**8.4%**	8.5%

different. The larger instruction window exposes more load-store reordering opportunities, and as a result, better dependence predictors are in fact required to manage the scheduling of these instructions. Figure 10 shows that there is considerable variation between the performance of the LWT predictor and the Perfect predictor for the large configuration. The floating point applications which showed similar speedups for the medium configuration regardless of the prediction algorithm used now show a slight improvement in performance when the Store Vectors or the Perfect predictor is used. The dependence sensitive applications are especially impacted at this larger configuration. As an example we highlight the performance of a memory dependence sensitive application $bc-2$. For the medium configuration, the LWT predictor had a 3% speedup over the Blind predictor whereas the Store Vectors predictor had a 3.6% speedup over the Blind predictor. In the large configuration however, the speedup of the Store Vectors predictor is approximately 7% whereas the speedup of the LWT table is less than 0.5% since the conservative LWT predictor is unable to accurately predict all the memory dependencies that are now exposed in this application. Due to some of the problems discussed in Section 3.6, the Store Sets algorithm only provides an additional speedup of around 1.7% over the LWT approach. Store Vectors, however, achieves performance levels much closer to the oracle predictor than the other techniques, although there still remains some room for improvement.

For the extra-large configuration, the performance difference between the memory dependence predictors is even more dramatic, as shown in Figure 11. There is now a much larger performance gap between the simple LWT predictor and the oracle predictor (2.1% verses 9.2% across all benchmarks, and 6.7% versus 14.9% on the dependence-sensitive applications). Per-application performance numbers for this configuration also show the benefit of using advanced memory dependence predictors like Store Sets and Store Vectors. Both *crc*32 and *dijkstra* post considerable performance losses when using the LWT predictor. Store Sets and Store Vectors in these cases match the performance of the oracle predictor. Additionally, among the integer applications, the *gzip* benchmarks also suffer performance degradations when the LWT predictor is used. The results, however, are quite encouraging when using the advanced memory dependence predictors like Store Sets and

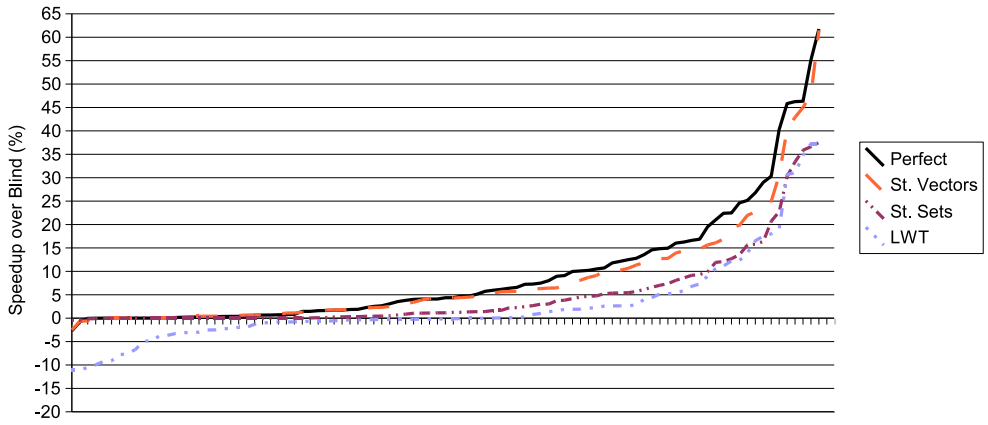


Figure 12: S-curves illustrating the range of performance behavior of the simulated memory dependence predictors across all benchmarks for the extra-large configuration.

Store Vectors. In fact, when considering all of the application results for the extra-large configuration, we see that where speculative memory dependence prediction does not help much (low performance difference between Perfect and Blind) Store Sets and Store Vectors are more robust and do not cause severe performance degradation as opposed to LWT which posts losses of more than 10% in several cases. These results highlight the fact that when there is something to be gained through prediction, Store Sets and Store Vectors do a much better job of reaching oracle performance than LWT, and when there is not much performance to be gained they are less harmful.

To illustrate this robustness property, we present performance S-curves for the extra-large configuration for the LWT, Store Sets, Store Vectors and Perfect predictors in Figure 12. The plots show the relative speedup over Blind prediction observed for the various benchmarks sorted from the lowest speedup (or greatest slowdown) to the highest speedup. From these curves we can see that the LWT predictor does see significant performance slowdowns (greater than 1%) for 25 out of the 94 benchmarks simulated with highest observed speedup of 37%. Store Sets does not see any major performance losses but stays flat for a considerable number of benchmarks with highest observed speedup of approximately 37%. Among all the proposed memory dependence predictors Store Vectors follows the trend of the perfect predictor more closely than the other simulated memory dependence predictors.

There are a few applications where LWT and Store Sets perform slightly better than

the Store Vectors algorithm. In these cases, there were more memory ordering violations overall in the blind speculation case. While the LWT is conservative in its general predictions, the Store sets' set merging heuristic sometimes causes Store Sets to also make more conservative predictions, resulting in slightly better performance than Store Vectors for these applications. For a small number of other applications, both Store Sets and Store Vectors are too aggressive and actually cause performance to be slightly worse than either the baseline case or the LWT. There are also a few cases where the LWT, Store Sets or Store Vectors outperform perfect dependence prediction due to the prefetching and early branch misprediction detection effects mentioned earlier.

Overall, these results indicate that for current processors, using a simple memory dependence predictor like the LWT should be sufficient, as evidenced by the simple PC-based dependence predictor used in the Intel Core 2 microarchitecture [19]. While some market segments are moving to a larger number of smaller, simpler cores, other mainstream product lines continue to invest more resources to expose more instruction level parallelism (e.g., Intel's "Nehalem" core [33]). Even in a heavily multi-cored world, some significant resources will still be required to attack the Amdahl's Law sequential bottleneck [31]. For these larger instruction window processors (or cores), our results lead us to conclude that more sophisticated memory dependence prediction algorithms will be needed.

3.5 Optimizations

In this section we study the performance impact of limiting the hardware budget for the Store Vector predictor tables, and then explore several optimizations to further reduce the hardware requirements without severely impacting overall performance. Since the performance trends of the large configuration mirrors that of the extra-large configuration, all the results in this section are presented only for the the medium and the extra-large configuration.

3.5.1 Sensitivity to Hardware Budget

Figure 13 shows the relative performance for all memory-dependence sensitive applications across a range of hardware budgets. The results show that even for a small hardware budget,

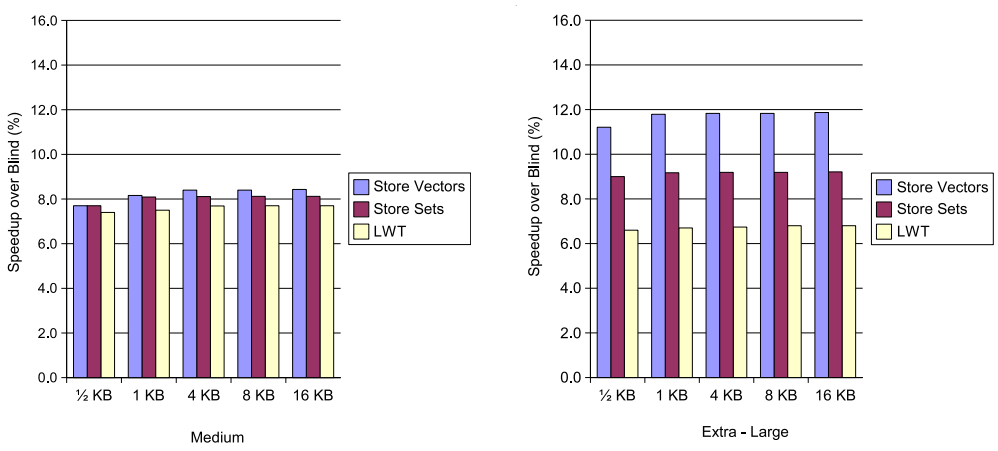


Figure 13: Speedup over blind prediction for the memory dependence-sensitive benchmarks of proposed memory dependence predictors across different hardware budgets.

sophisticated predictors like Store Vectors provide a significant performance benefit. In all of the proposed predictors, exceeding the hardware budget beyond 4 KB does not buy much additional performance. This is intuitive as the predictors hold dependence information for the loads. While there are many loads in a program only few of them collide with earlier stores and hence only information regarding these loads affects the performance of the application. If we decrease the hardware budget below 1 KB, however, we observe that the performance drops off. This is primarily a result of aliasing in the Store Vector table. For some applications with large working sets this affects performance. We now describe how we can reduce the hardware budget below 1 KB for the Store Vectors predictor without significant loss of performance.

3.5.2 Reduction of Store Vector Length

As described, the Store Vector length is tied to the number of entries in the store queue. While we found that this configuration performed well, it is possible to reduce the Store Vector length to decrease the hardware cost. For example, reducing the Store Vector length to only track the 16 most recent stores (as opposed to 32 as assumed in Section 3.3) would reduce the space requirements of the Store Vector Table (SVT) by one half. However, the sizes of the remaining hardware structures are still tied to the store queue size. The Load

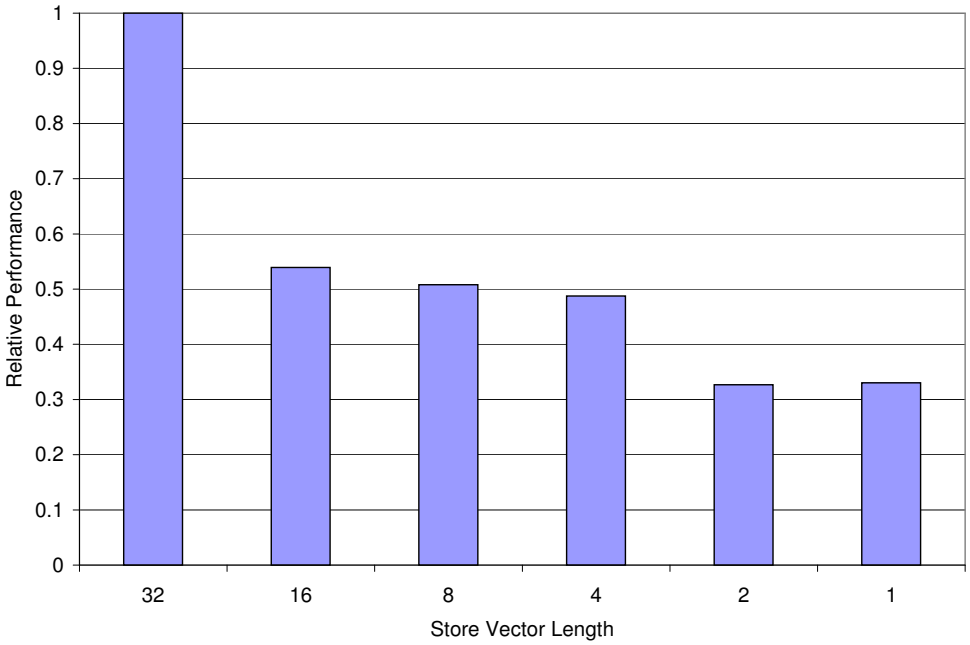


Figure 14: The relative performance impact of reducing the length of the Store Vectors (dependence-sensitive applications only). Zero represents the performance with blind speculation, and 1.0 is the performance of the baseline Store Vector predictor.

Scheduling Matrix still requires one column per store, and therefore the barrel shifter must also produce a bit vector matched to the number of store queue entries.

For our processor configuration, the benefit of Store Vectors drops off relatively quickly with shorter Store Vector lengths. Figure 14 shows the relative performance for different Store Vector sizes, where zero is the performance of blind speculation and 1.0 is the performance of the baseline Store Vector technique. From these results, we conclude that the less recent stores have a greater impact on load ordering violations. This intuitively makes sense as the compiler should choose to spill registers that will not be soon reused. If a data dependency exists between two nearby instructions, it is likely that the value will be kept in a register as opposed to being pushed out to memory. Even for subroutine calls, most functions have only a few arguments which can be passed through the registers. Our evaluation was conducted on the Alpha ISA which has a relatively large number of registers; the results and optimal memory dependence predictor configuration are likely to be different for an ISA like x86 where spills/fills and memory traffic in general is much more frequent.

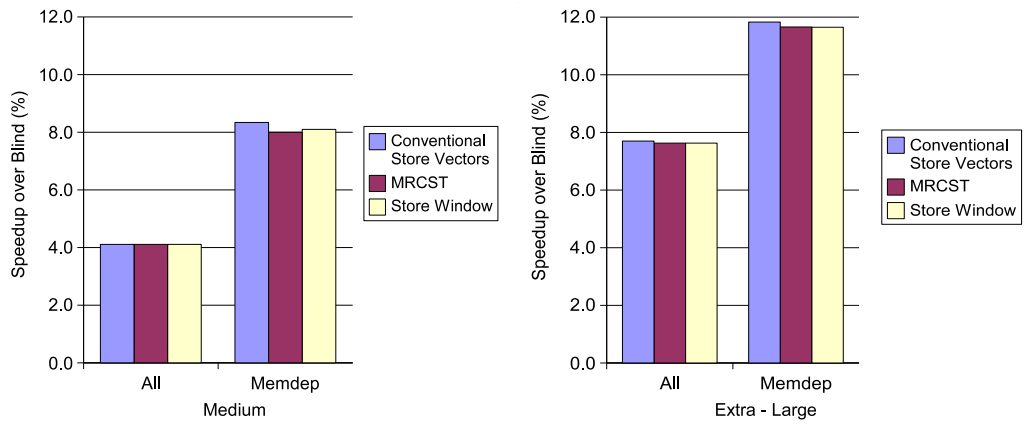


Figure 15: Different Store Tracking Strategies. The second bar indicates the MRCST strategy and the third bar depicts the performance of the store window method.

3.5.3 Most Recently Conflicting store

While Figure 14 indicates that the performance of Store Vectors drops when fewer stores are tracked in a conventional Store Vector technique, if we model a different style of tracking stores we may still be able to reduce the Store Vector length and not incur a great loss in performance. Our motivation in searching for different approaches to holding store information for each load comes from the observation that each Store Vector generally only has a few 1's. This indicates that only a few stores ever collide with the load and hence it may not be necessary to allocate storage for all potential stores in the Store Vector. The trick lies in being able to keep only relevant stores in the vector. We tried two such techniques, the first one being where only the most recent conflicting store (MRCST) would be kept in the SVT. The MRCST is that particular store that caused a memory ordering violation with the load the last time this load was issued out of order. This could potentially reduce the hardware cost of the Store Vector technique tremendously since each entry in the SVT now holds just one$\lceil \log_2 n \rceil$-bit store age-index in place of an n-bit vector. The reasoning behind this strategy is that since store-load dependencies generally follow a recurring pattern, just storing information about the most recent dependence should suffice. Another recent distance-based memory dependence predictor proposed exploits a similar observation and makes a load depend only on its most recent conflicting store [67]. The MRCST technique needs the index of the store to be kept in the SVT so that the correct store cell can be set in

the matrix. The second bars in Figure 15 correspond to MRCST technique in the medium and extra-large machines, respectively. The results are encouraging because even though the performance is lower than that of conventional Store Vector technique, it still outperforms Store Sets (which shows 8% and 9% speedup over Blind for sensitive applications in the medium and extra-large configurations respectively) at less than a quarter of its hardware budget. MRCST provides a very scalable SVT since the hardware required for it is much less than the conventional SVT. For the purpose of this experiment to model MRCST we used an SVT with 512 entries each holding a store age-index of 5 bits giving a total SVT hardware of 0.3125 KB. Thus for negligible loss in performance we get a 84% reduction in hardware from the original SVT (2 KB) evaluated in Section 3.4. The implementation complexity of this technique is also much lower. We can replace the barrel shifters needed to align the vector in the matrix as explained in Section 3.3.1 with a narrow-width adder and decoder.

3.5.4 Store Window

Extending this idea our next strategy deals with storing just a window of k stores where k is statically assigned like described before but the start of the window can be dynamically chosen. If for a particular load its 4th most recent store to its 8th most recent store are the stores generally causing a problem then storing only these in the SVT may be enough. Each store window was chosen to start at the store that corresponded to the load's MRCST as described previously. Thus each vector now needs to hold a store age-index indicating the MRCST and k bits after it which indicate the dependence information. In this experiment we keep track of 4 stores thus each vector holds a 5 bit age-index and 4 bits of dependence information. In both plots of Figure 15, the 'Store window' bar corresponds to this technique simulated in the medium and the extra-large machine respectively. In this method too, for negligible loss in performance we obtain nearly 70% reduction in hardware. The store window technique, however, does not provide much performance improvement over the MRCST as indicated in Figure 15.

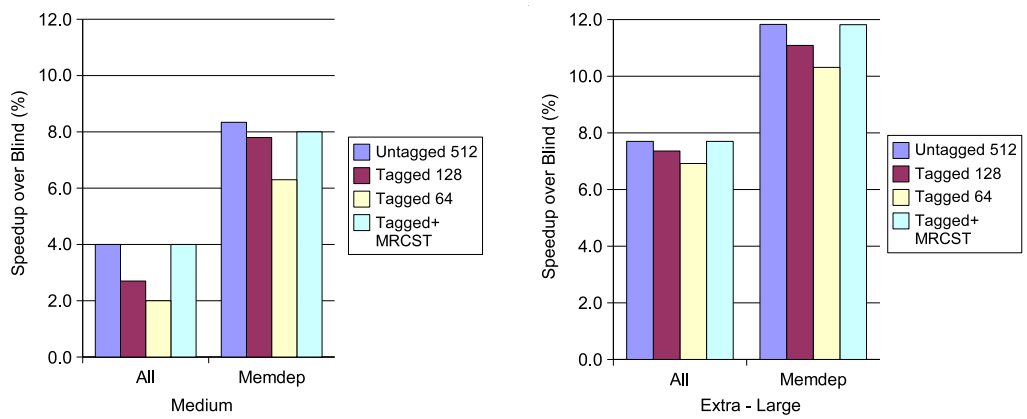


Figure 16: Performance of tagged SVT of different sizes compared with conventional untagged SVT. The first bar represents an 2KB untagged SVT. The second and third bars represent 0.625KB tagged SVT and 0.3125KB tagged SVT respectively while the fourth bar corresponds to a 0.1526KB tagged SVT storing only the MRCST.

3.5.5 Tagged SVT

After considering how we could best reduce the Store Vector length, we then moved on to determining whether some of the vectors could be eliminated altogether. On further examination it was found that not only were a lot of bits in some vectors zero, but many vectors only contained zeroes. This is not very difficult to comprehend as the SVT stores information for each and every static load encountered in the program. Many loads just do not have dependencies with earlier unresolved stores. Knowing that a vector is completely zero is important in order to ensure that loads are allowed to issue as soon as their addresses are ready but storing these zero vectors is wasteful and with just a slight change in the structure of the SVT these vectors could be eliminated while still keeping track of the respective loads. We augment the SVT with tags and only store non-zero vectors. A load miss in the table indicated that its vector was completely zero and the load could speculatively issue. This strategy of eliminating zero vectors greatly reduces the hardware budget while incurring practically no loss in performance. The reason for this is quite simple. All of the necessary information that enables Store Vectors to be an accurate memory dependence predictor is contained only in the non-zero vectors and hence storing only these vectors had minimal impact on the performance. For this simulation we assumed a partial tag of 8 bits and a conventional Store Vector of 32 bits to give a total vector size

of 5 bytes. In Figure 16 the first bar shows the performance of Store Vectors when using a untagged SVT (2KB), the second bar shows a tagged SVT with 128 entries (0.625 KB) and the third bar shows a tagged SVT with 64 entries (0.3125 KB). With the 128-entry tagged SVT we get 68% reduction in hardware and with the 64 entry tagged SVT we get 84% reduction in hardware. In fact the 128-entry tagged SVT performs as well as a 2 KB Store Sets predictor or a 2KB load wait table predictor (results shown earlier in Figure 9 and Figure 11. These results are consistent with our observation that most vectors are zero vectors and hence eliminating them does not significantly affect the performance. Finally we present the performance of the most optimized version of the Store Vectors algorithm, which involves using a 256-entry tagged SVT, that stores only the most recently conflicting store (fourth bar). This configuration provides comparable performance as the original Store Vectors algorithm with a hardware budget of 0.1563 KB. Additionally this configuration provides a 92% savings compared to the LWT and the Store Sets predictor with a performance difference of over 5% and 3% respectively. Even though this design stores the least amount of information as compared to the other designs, it stores the most pertinent information.

3.5.6 Effect of Incorporating Control Flow Information

In this experiment we use branch history information along with the load PC to index the SVT in the Store Vectors algorithm. In theory, Store Sets should have some more tolerance to memory dependence predictions that vary depending on the control path leading up to a load because the stores are all explicitly tracked by their PCs. With Store Vectors, a load might have a conflict with the third previous store when the program traverses one path, and the load might conflict with the fourth oldest store on another path. In this scenario, the Store Vector algorithm will mark bits three and four as conflicts, which may in turn cause the load to be overly conservative. We observe negligible improvement in performance, however, by the addition of control flow information. There are a few reasons why this effect does not have a great impact on performance. First, the load is only unnecessarily delayed if the non-conflicting store address resolves after the real store dependency. Second, even if

delayed, the load only impacts performance if it is on the critical path. Third, aggressive compiler optimization may reduce the effects of control flow on dependence prediction. A load aggressively hoisted and duplicated to both paths leading up to a control flow join effectively assigns the load to two different PCs which allows the predictor to identify the different control flow cases. If programs other than SPEC exhibit a high-degree of memory dependences within very branchy code, the Store Vector table can be modified in a gshare fashion to make use of some branch or path history information [48]. A memory dependence predictor that incorporates branch history information was also separately implemented in [68].

3.6 Why Do Store Vectors Work?

Store Vectors utilizes the predictability in relative distances between colliding loads and stores. Hence even if the control flow of the program differs, the relative position of a store that a load depends on is predictable. Another proposed work by Yoaz et al., made use of distance based information for determining which loads could be speculatively disambiguated [87]. The authors proposed using a collision history predictor where each entry would store a minimum allowable distance that the load can be advanced. This distance is based on past observed distances between conflicting loads and stores. However the mechanism of distance information collection and load scheduling is quite different from the store vector table and load scheduling matrix used by the Store Vector algorithm. In this section we specifically explain the advantages of Store Vectors over Store Sets and the LWT. In particular we will discuss scenarios where the Store Sets algorithm results in overly conservative predictions or unnecessary delays. The first reason why Store Sets can delay a load from issuing actually stems from delaying the issuing of stores. The desired behavior of a load with a predicted set of store dependencies is that the load waits for all of the dependencies to be resolved. Figure 17(a) shows a load instruction with four predicted store input dependencies. Ideally, the stores should be able to execute independently since memory write-after-write false dependencies are already properly handled by the store queue. To prevent loads and stores from having multiple direct input dependencies (which

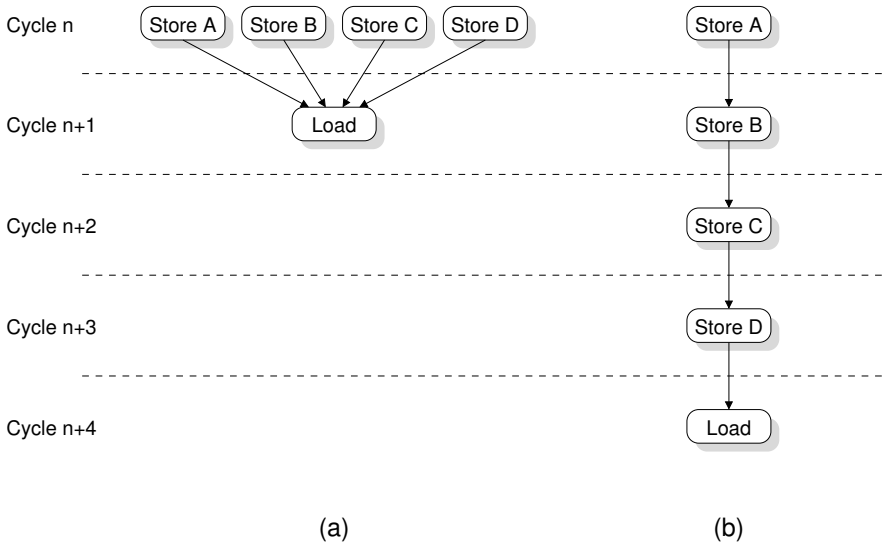
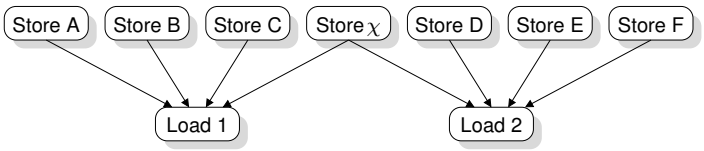


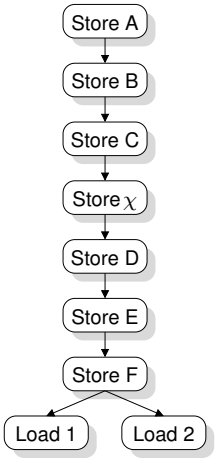
Figure 17: (a) Ideal load synchronization against predicted store dependencies, and (b) actual serialization with Store Sets.

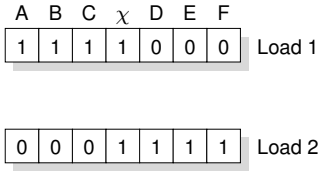
would require more CAMs), the Store Sets algorithm serializes execution of stores within the same Store Set. This is illustrated in Figure 17(b) where the load's execution has now been delayed by three extra cycles. With the Store Vector approach, individual stores have no knowledge about dependency relationships with other stores; in fact, the store does not even explicitly know if any loads are dependent on it. Stores may issue in any order, and they just obliviously clear their respective columns in the LSM.

The Store Sets merging update rules allow two or more different loads to be dependent on the same store. Consider the loads in Figure 18(a), where both loads have several predicted store dependencies, and both loads are predicted to be dependent on Store-χ. Without Store Sets merging, Store-χ can only belong to the Store Set of Load-1, for example. In this situation, Load-2 will not wait for Store-χ which results in ordering violations. With Store Sets merging, all of the stores associated with both loads will be merged into the same Store Set. Now Load-1 and Load-2 will serialize behind Store-χ. This prevents the ordering violation. Unfortunately, this can introduce substantial additional store dependencies. Load-1 must wait for all stores in Load-2's Store Set, and visa-versa. If all Stores A through F, and Store-χ are simultaneously present in the store queue, then Loads 1 and 2 will be considerably delayed as illustrated in Figure 18(b). With the Store Vector approach, each



(a)





(b) (c)

Figure 18: (a) Ideal load synchronization for two loads against their predicted store dependencies, (b) actual serialization of merged Store Sets, and (c) the Store Vector values which avoid unnecessary serialization.

load may have its own Store Vector that is capable of tracking dependencies independent of all other loads (modulo aliasing effects in the SVT). Figure 18(c) shows the corresponding Store Vectors for Loads 1 and 2 that do not result in the spurious serializations induced by Store Sets.

The load wait table though simple in design does not equal the performance of an oracle predictor due to its conservative nature. As described earlier the LWT does not store identifying information about the colliding load-store pair but only marks suspected loads. This marking then prevents the load from issuing before all previous unresolved stores. Thus LWT does not take into account the various dynamic instances of store-load pairs and makes predictions that can tend to be too conservative. Much like the untagged Store Vector table, the load wait table too, has many entries that are zero and do not hold any valuable information.

3.7 Implementation Complexity

Having described the various implementations of the predictors as well as presented the performance of these predictors in different processor configurations, we now compare the implementation complexities of these predictors. The basic units that handle memory instructions are the load queue and the store queue. These structures contain CAM logic to provide memory dependence checks and store forwarding. We now discuss how the load and store queues must be modified to support the different prediction algorithms along with any additional hardware structures/overhead that is required.

The Load Wait Table is the simplest dependence predictor that we evaluated, and as such requires the fewest modifications to the existing memory scheduling structures. In a processor without speculative load-store reordering, loads must wait until all previous (older) stores have resolved their addresses. A load's "readiness" is now effectively gated by this all-earlier-stores-resolved signal. To implement the LWT predictor, we simply compute the logic OR of this signal with the LWT's prediction (assuming a prediction of 1 indicates that the load is predicted to have no dependencies; an extra inverter takes care of the case of the reversed prediction encoding). Apart from the predictor table itself, which can be accessed early in the pipeline off the critical path, there are no other changes to the memory scheduling structures. While very simple to implement, our earlier performance results showed that the performance benefits of the LWT approach are limited when we consider larger processor configurations.

Store Sets can be implemented using either a CAM or RAM-based organization as explained in Section 3.2.2. Assuming a CAM implementation, the addition of a second set of CAMs in both the load and store queues represents a very large area and power overhead that impacts the scalability of the already difficult to scale load and store queues. These CAMs need as many ports as the maximum per-cycle store issue rate. Even for current load and store queue sizes, implementing these additional CAMs will not likely be able to meet clock frequency targets or would otherwise require very deep and complex pipelined accesses. These CAMs, however, are not as bad as traditional CAMs designed for load and store queues since they require smaller comparator widths and do not require age support.

When using a RAM-based implementation, the maximum number of loads per Store Set either makes the load queue difficult to scale (too many write ports, even though they are direct/RAM accessed) or requires some overflow handling. A third alternate implementation of Store Sets can use a matrix-based memory dependence scheduler similar to the one used in the Store Vectors algorithm. This matrix would track the dependencies between the instructions in a Store Set. The matrix, however, would have to be considerably larger as we need one row per load *and* one column per store since stores need to wake up other stores belonging to the same store set.

As previously discussed in Section 3.2, branch misprediction recovery introduces further complexities for cleaning up broken store-to-load pointers/indexes. The recurrence induced by intra-group Store Set dependencies also makes the implementation of the LFST difficult for a wide-issue processor without placing additional constraints of load/store dispatch/allocation rates. The speculative update of the LFST with wrong-path store instructions also requires additional complexity to repair the LFST's state after a branch mispredict detection. The same types of techniques employed for repairing the register alias table (RAT) could potentially be employed here. RAT/mapper checkpointing is a commonly used technique for rapid recovery of the renaming state [35], but this approach is more costly for the LFST. The RAT typically only contains a number of entries equal to the size of the architected register file, whereas the LFST contains 256 entries thereby requiring more storage per checkpoint. If the processor employs an "undo-list" approach for pipeline recovery where the recovery logic walks the ROB and incrementally undoes the RAT changes, then we can piggy-back on this logic to also repair the LFST entries at the same time.

In either case, the recovery needed to support Store Sets adds a level of implementation complexity that Store Vectors does not require.

The Store Vectors algorithm needs a scheduling matrix which has number of rows equal to the load queue size and number of columns equal to the store queue size. This matrix only needs one wire per column which is connected to the corresponding store queue entries for clearing the column when the store issues. Each row of the matrix needs a wired OR

or wired AND circuit (depending on whether the logic is implemented as active-low or active-high) that indicates when the load is ready to issue. The Store Vector Table (SVT) is a simple SRAM with the same port requirements as the basic LWT predictor. The SVT does not suffer from the dependency recurrence of Store Sets' LFST. The last significant component is a barrel shifter for rotating the vector to align it properly in the matrix. As the store queue size increases, the width of the shifter must also increase proportionately, which could make the shifter a critical timing path. Fortunately, this shifting operation can be very easily pipelined over several cycles. The SVT lookup and subsequent barrel shift can be started as soon as the decode stage has identified the instruction as a load (this may also require that the decode stage maintains a speculative STQ tail pointer, but this is also trivial to implement). For these reasons, the shifter does not impose any timing problems, although the area cost may be moderate since we need to implement one shifter per load (up to however many loads may be decoded per cycle). The shifter requirements may be further reduced by using a tagged SVT and then allowing only one load with predicted store dependencies to proceed to the shifter per cycle. This works well since the majority of loads are predicted to *not* have a dependency, which does not place much pressure on the shifter.

3.8 Conclusions

In this chapter, we study the predictability in memory dependences and exploit this common-case behavior to design a low-cost, scalable and high-performing memory dependence predictor which helps to improve the overall efficiency of the processor. Out-of-order load execution is necessary to realize the potential of large-window superscalar processors. Blindly allowing loads to execute, however, will result in ordering violations which may cancel out the benefits of supporting a large number of in-flight instructions.Our research has provided the design of a new memory dependence prediction and scheduling algorithm based on dependency vectors and scheduling matrices. An important result in this study is that by optimizing our base store tracking strategies (MRCST and Store Window) and predictor

management (tagged design), we are able to obtain nearly 90% hardware savings as compared to the original Store Vectors. Thus, depending on the given hardware requirements, Store Vectors can be easily modified and still perform very well. Our results also show that, while performing substantially better than previously proposed predictors, there remains some performance left between Store Vectors and an ideal predictor, which suggests that further innovation may help to extract even more performance.

CHAPTER IV

PEEP: APPLYING PREDICTABILITY OF MEMORY DEPENDENCES IN SIMULTANEOUS MULTI-THREADED PROCESSORS

4.1 Introduction

The previous chapter demonstrated that, in the common-case, the memory dependencies of load instructions can be accurately predicted. In this chapter, we focus on efficient resource-utilization with a view to maximizing the throughput of a simultaneous multi-threaded (SMT) processor with the help of memory dependence prediction [80]. We propose a technique that makes use of the predictability in memory dependences to anticipate stalls that are likely to occur due to memory dependences, and then apply this information to optimize the allocation of resources in the processor. Our resource allocation strategy specifically optimizes the fetch engine of the processor which is one of the most critical units in modern processors. We present results that show that when the fetch engine of a modern SMT processor is augmented with our PEEP technique, resource allocation becomes more efficient. This improves the processor throughput and performance which helps to improve overall efficiency.

While modern superscalars have been able to exploit instruction level parallelism (ILP) to improve single-threaded performance, there remain many resources in the processor that are under-utilized. One example of poor resource utilization is that a modern processor may provide functional units to execute 4-6 operations per cycle, but the sustained execution of most programs is only one to two instructions per cycle (IPC) at best. On an application stalled on a cache miss, for example, an out-of-order processor may not be able to find enough independent instructions to keep the functional units busy. During times of low IPC rates, other threads may have useful work that could be done.

To motivate our research design, we first study the efficiency of resource allocation in current SMT processors. SMT machines are designed to inter-mingle threads so as to maximize resource usage and improve overall throughput. As explained above, however, a thread that encounters a stalling condition (e.g., a cache miss with all other independent instructions already executed) can potentially tie up many of the shared resources for the entire latency of the stall. This effectively reduces the number of critical resources available to the non-stalled threads, and ends up reducing overall throughput. Past work has examined the impact of branch mispredictions and cache misses on SMT performance [37, 20, 82, 41, 16]. In this work, we study the interaction of memory dependence prediction with an SMT processor. From these observations, we propose a new class of SMT fetch policy filters, "PEEP", that exploit predictable memory dependences to avoid allocating resources to threads that are likely to stall on these dependences. We also use the predictability of dependence resolution latencies to mitigate thread starvation effects after dependency resolution.

The rest of the chapter is organized as follows: We give a brief background of the inefficiencies that exist in current fetch policies for an SMT processor in Section 4.2. We then present the design of PEEP, which augments traditional fetch policies with information about predictable memory dependences and their associated latencies in Section 4.3. We present detailed evaluation results that show the performance benefit as well as fairness impact of our proposed design in Section 4.4. We also use our technique in an alternative design that does not speculatively execute loads before unresolved stores to demonstrate that our proposed technique works because of effective fetch management rather than some second-order interactions between speculative load execution and the SMT microarchitecture. The design and performance evaluation of this approach is presented in Section 4.5. We present some additional sensitivity and scalability studies of our technique in Section 4.6 as well as present an industrial evaluation of our technique in Section 4.7. Finally we briefly summarize our work in Section 4.8.

4.2 Background

In this section, we first briefly review some commonly proposed SMT fetch policies and explain some of the program behaviors that can lead to performance degradations. We then discuss speculative execution of load instructions in the presence of earlier unresolved store addresses (speculative memory disambiguation) and its interaction with the SMT microarchitecture.

4.2.1 SMT Fetch Policies and Resource Stalls

The instruction fetch unit of an SMT processor has a huge influence on the overall throughput of the processor as well as the fairness of resource sharing among the different threads. On each cycle in which the processor front-end is not stalled, the fetch unit chooses one of the n threads to fetch instructions from. The decision process or algorithm for choosing which thread to fetch from is called the fetch policy. Common policies include Round-Robin and ICOUNT [83]. There are also some other polices that are described below.

While Round-Robin provides each thread with equal time on the fetch unit, it can still lead to significant reductions in overall throughput. Figure 19 illustrates a simplified view of an SMT processor's reservation station entries shared between two threads. The processor fetches from thread T_0 on even cycles and from T_1 on odd cycles. Assume that thread T_0 suffers a long-latency stall (such as a last-level cache miss) and does not have any instructions that are independent of the stalled instruction; thread T_0 will not be able to execute ($\underrightarrow{0}$) any more instructions until the stall has been resolved. T_1 executes all four of its instructions ($\underrightarrow{4}$) on alternate cycles. In the meantime, T_0 continues to occupy execution resources (represented by the white reservation station entries in Figure 19), and, even worse, the round-robin fetch policy continues to stuff more of T_0's instructions into the out-of-order core. Eventually, the entire set of reservation stations may be occupied by T_0's instructions thus stalling the entire pipeline until T_0's original stall has resolved.

In one of the early SMT studies, Tullsen et al. studied a variety of SMT fetch policies to exploit the fetch unit's choice of threads [83]. Tullsen et al. considered altering thread fetch priorities based on the per-thread counts of unresolved branches (BRCOUNT), data cache

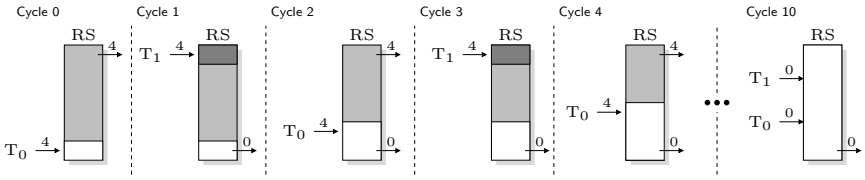


Figure 19: Shared reservation station entries in an SMT processor can be completely occupied by a stalled thread when using simple fetch policies.

misses (MISSCOUNT), and overall instruction counts (ICOUNT). The ICOUNT policy was shown to provide the best overall throughput out of the policies evaluated, and has become a de facto standard in academic SMT studies. The fetch unit tracks the number of instructions currently in the machine for each thread and fetches from the thread with the lowest number of instructions. As stated in the original study, ICOUNT provides three major benefits: it prevents any single thread from filling up the reservation stations, provides additional priority for threads that move instructions through the processor quickly, and generally provides a more balanced ILP mix between the threads to better utilize the processor resources. The ICOUNT policy works well when the processor does not have too many low-ILP/frequently-stalling threads. Since ICOUNT does not have any explicit knowledge of stalls, any completely stalled thread will eventually consume $\frac{1}{n^{\text{th}}}$ of the reservation stations. If a given application mix contains more than a handful of such threads, then a substantial number of processor resources will be tied up, thus reducing the overall throughput of the SMT processor.

One approach to addressing this attribute of the ICOUNT policy is to anticipate the stalls. Luo et al. studied the use of branch confidence prediction [34] to control the amount of control speculation per thread [37]. The intuition is that after fetching past a few low-confidence branches, the probability of actually fetching any useful instructions rapidly tends toward zero. El-Moursy et al. also considered the prediction of data cache misses to indicate which threads will likely experience more stalls [20]. Neither branch confidence prediction nor load miss prediction is easy, which leaves a fetch policy based on either of these predictors vulnerable to the mispredictions. The load-miss prediction policy augments the count of predicted load misses with a count of actual load misses to help offset the

difficulties in load-miss prediction. Unfortunately, when the processor discovers that a load has missed in the cache, the fetch unit may have already fetched many instructions dependent on this load miss. A hybrid load-miss predictor was also proposed in [87]. This predictor uses a majority vote to choose between predictions made by a basic predictor similar to El-Moursy's, a gshare predictor and a gskewed predictor.

There have been several other similar cache-miss related SMT fetch policies such as STALL and FLUSH [82], DC-PRED [41] and DCache-Warn [16]. Another recently proposed policy makes use of the available memory-level parallelism (MLP) in a thread [21]. The key idea is a thread that overlaps multiple independent long-latency loads has a high-level of MLP, and therefore this thread should be allowed to allocate as many resources as needed in order to fully expose this available MLP. In the event of an isolated load miss, however, this thread will be stalled from fetching further instructions and the remaining processor resources will be available to the other threads.

4.2.2 Memory Dependences and SMT

In the preceding sub-section, we described a few previously proposed techniques for predicting impending stalls before they happen. The prior work has focused primarily on branch and data-cache related stalls. Instructions in modern superscalar processors often experience a different kind of stall due to memory dependences. A load may have its address computed and be ready to issue to the data cache unit, but if there exists earlier store instructions whose addresses have not been computed, the processor will not know whether it is safe to allow the load to access the cache.

Some processors such as the Alpha 21264 [35] and the Intel Core 2 [19] allow load instructions to speculatively execute before all previous store addresses have been computed. A memory dependence predictor decides whether a load can execute as soon as its address is known, or whether it should wait until certain or all previous store addresses have been computed [50, 17, 79]. Accurate predictions allow loads to execute earlier, but mispredictions result in costly pipeline flushes. Prior work on memory dependence prediction has demonstrated that the relationships between load and stores are highly predictable,

and the predictability of these patterns has been exploited in other memory scheduling research [67, 68, 78, 15].

To the best of our knowledge, no prior work has explored the interactions between speculative memory disambiguation and simultaneous multithreading. In this sub-section, we briefly present the results of using memory dependence prediction in an SMT processor. The full details of our simulation methodology and workloads can be found in Section 4.4, but in short we model a four-threaded SMT processor with and without support for speculative memory disambiguation using a variety of multi-programmed workloads. Figure 20 shows the speedup of an eight-issue SMT processor using Round-Robin (RR) and ICOUNT fetch policies over an SMT processor with no speculative memory disambiguation. The workloads include different mixes of benchmarks with low, medium or high ILP levels (L/M/H) and programs that are sensitive (S) and not-sensitive (N) to speculative load reordering. For the configurations supporting memory dependence speculation, we implemented a 4K-entry load-wait table (like that used in the Alpha 21264 [35]) and an oracle dependence predictor (predicts exactly which store a load is waiting for and the load executes immediately after the store executes) for comparison. We also ran our simulator in single-threaded mode to measure the impact of memory dependence prediction on single threaded applications in our simulator. We found that for single threaded applications, perfect memory dependence prediction provides 7% performance improvement over no memory dependence prediction.

The results of our four threaded experiments are presented in Figure 20. The realistic load-wait table (LWT) provides 2.0% and 1.0% improvements in overall throughput for RR and ICOUNT, respectively.With an unachievable oracle predictor, the overall throughput increases by 4.1% and 4.0% over the baseline RR and ICOUNT. These relatively low performance benefits may at first be somewhat surprising given the value of speculative memory disambiguation in traditional single-threaded processors. The addition of such capabilities to the Intel Core 2 microarchitecture is evidence of the value of aggressive load reordering, at least for single-threaded performance [19]. The SMT results are not unreasonable, however, since the intuition is that in a single-threaded processor, a stall due to an unresolved

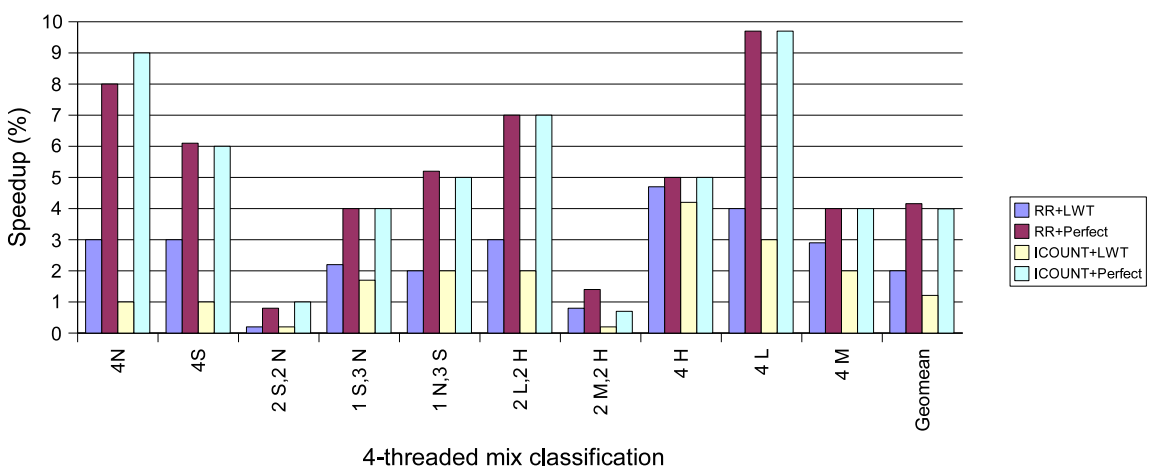


Figure 20: Impact of speculative load disambiguation on an SMT machine. The speedups (throughput improvements) are relative to SMT machines running round-robin and ICOUNT policies with no speculative memory disambiguation. LWT is the load-wait table memory dependence predictor. Perfect prediction uses an oracle predictor.

memory dependence can easily prevent a large number of instructions from making progress. In an SMT processor, there may be enough thread-level parallelism to largely compensate for a small number of load instructions being delayed for a few cycles. While speculative memory disambiguation and memory dependence prediction may not greatly increase the throughput of an SMT machine, it is important to ensure high single-threaded performance. One naturally asks the follow up question: apart from enhancing performance when only a single thread is present what other benefits memory dependence prediction can provide in SMT processors.

4.3 Adapting SMT Fetch Policies to Exploit Memory Dependences

The results of the previous section demonstrated that speculative memory disambiguation alone does not provide much performance benefit for SMT processors. In this section we describe a new technique that takes advantage of the fact that the memory dependences are highly predictable and uses this predictability for making instruction fetch decisions.

4.3.1 Proactive Exclusion

The central observation used here is that if the timings of an instructions' stalls are predictable, then this information can be directly exploited by the SMT fetch unit to better manage the shared processor resources. We say that a fetch policy uses *proactive exclusion* if it can stop fetching from a thread before the thread's stalling condition has been exposed in the out-of-order execution core. El-Moursy and Albonesi's fetch-gating on predicted load misses is a form of proactive exclusion [20]. In this work, we instead propose to implement proactive exclusion using a much more predictable property of programs: memory dependences.

Our PE_{mdep} (proactive exclusion based on memory dependences) is actually not a fetch policy in its own right, but rather it is an SMT fetch policy filter. That is, we can potentially apply PE_{mdep} to any other standard SMT fetch policy. PE_{mdep} starts with a memory dependence predictor, which could make use of any number of previously proposed algorithms [50, 17, 79]. When the fetch unit attempts to fetch instructions from thread i, it also consults the memory dependence predictor to find out if this thread will likely stall due to ambiguous memory dependences, as shown in Figure 21(a). If the answer from the predictor is "no dependence," then the underlying fetch policy continues operating as it otherwise would without PE_{mdep}. However if the predictor answers "dependence present," then the fetch unit finishes the current fetch and then removes thread i (T_0 in Figure 21) from the list of threads from which it can fetch instructions for (b). The fetch unit will then continue fetching, applying its original fetch policy to the current list of fetch candidates. When the load that was predicted to have a dependence issues, the out-of-order core signals the front-end that this dependence has been resolved and then thread i is reinserted into the list of fetch candidates (c). At this point, a policy like ICOUNT can help thread i catch up with the other threads, thus providing a reasonable amount of fairness.

By keying fetch decisions on a highly predictable attribute, the fetch unit can easily avoid fetching the instructions that immediately follow loads with dependences. This helps to reduce the number of reservation station entries occupied by stalling operations. Following a load with a predicted memory dependence, there may be some other operations that are

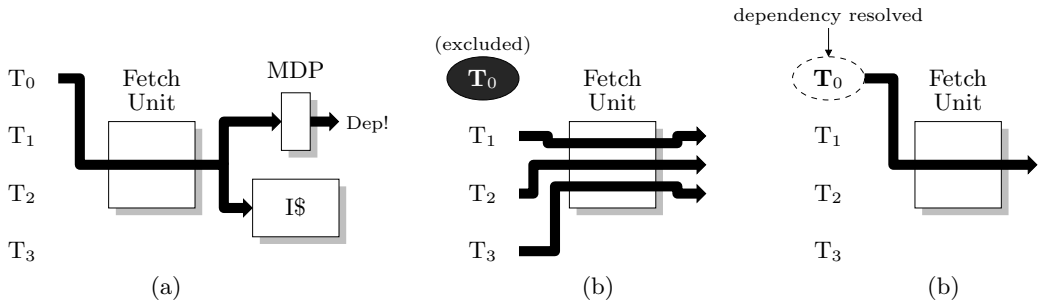


Figure 21: Demonstration of Proactive Exclusion. (a) When fetching a load, a predicted memory dependence causes the thread T_0 (b) to be excluded from the list of fetch candidates. (c) Only when the dependence has been resolved does the thread's fetching resume.

not data dependent on the stalled load. However, it is better for overall throughput to fetch from a different thread whose instructions are guaranteed to be independent of the stalled load rather than fetch instructions that only might be independent of the stalled load.

In the above description, we specified PE_{mdep} as checking the memory dependence predictor during fetch. Performing lookups at instruction fetch, however, would require making multiple parallel lookups from the predictor (although techniques used for making multiple branch predictions can help [66]), and the additional aliasing combined with the lack of decode information can lead to predicting that a memory dependence exists when the instruction itself might not even be a load (this is analogous to the phantom taken branch problem encountered when predicting multiple branches per cycle). The decode stage would need to provide an additional dependence-clear signal to notify the front-end of non-loads predicted to have dependences. For implementation purposes (and as modeled in our simulations), we do not actually consult the predictor until the instruction has been decoded and known to be a load. This could lead to a situation where a few additional instructions after the predicted-to-stall load also make it into the pipeline. There are potential negative and positive effects associated with these extra instructions. On the one hand, these instructions may tie up additional reservation stations and reduce overall throughput. On the other hand, these extra instructions provide the thread with a little extra work ready-to-go for when the load's dependency has finally been resolved. Without these instructions, the thread may temporarily starve because the delay from dependency resolution to the fetch

unit and back to the execution core can cause that thread to not execute any instructions for several cycles. Note that the number of additional instructions fetched from this thread is usually limited since the fetch unit will choose to fetch from other threads as well.

4.3.2 Leniency and Early Parole

At the end of the discussion of Proactive Exclusion, we discussed the potential problem of temporary thread starvation due to the delay between dependency resolution and when instructions from that thread make it back into the execution core. Figure 22(a) illustrates this scenario with a simplified view of the pipeline where a dependency has been predicted but is also quickly resolved. While a load may need to stall for ambiguous memory dependences, the time required to resolve this ambiguity may actually be fairly short. Proactively excluding the thread T_0 based on the predicted latency for instruction M hurts performance since the actual resolution latency is less than the resolution-to-fetch-to-issue delay, thereby starving T_0 for cycles i+5 through i+8 in Figure 22(a).

An interesting and useful attribute of memory dependences is that not only are their existences predictable, but their durations are predictable as well. We augment our PE_{mdep} technique with a PC-indexed delay predictor, indicated as ΔP in Figure 22(b). This structure could be implemented as an additional field in the existing dependence predictor entries. If the predicted delay is less than a fixed threshold, then we forgo the proactive exclusion of this thread under the assumption that the stall will not last long enough to make exclusion worthwhile. This is the concept of *Leniency*. In this example, the load M has resolved its memory dependences by the time its dependent instruction N has made it to execute, thereby avoiding any slowdown in this thread.

We measure the resolution delay as the number of cycles between when a load instruction is ready (effective address computed) and when it actually issues. The delay predictor is updated at load commit with the new resolution delay value based on the type of delay predictor being used. Our conservative delay predictor tracks the observed delay and records the maximum delay value ever observed. To prevent the table from completely saturating with only long-latency predictions, we periodically reset the table. We also consider a

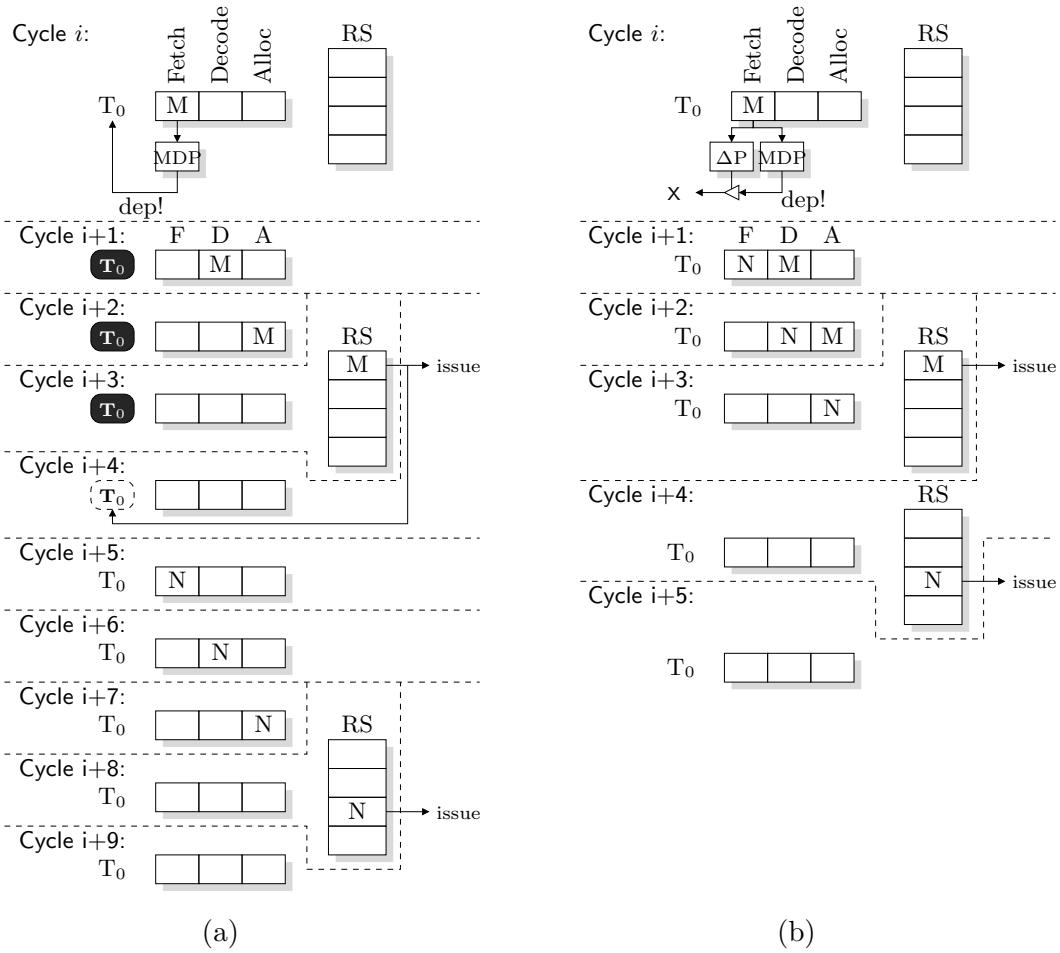


Figure 22: Timing diagram illustrating additional delays induced by (a) excluding a thread when its resolution delay is very short, and (b) avoiding its exclusion.

more aggressive delay predictor that simply predicts whatever the last observed delay was. Finally we also model an adaptive delay predictor that predicts the delay of a thread based on the last n observed delays and then uses the average of these delays as the next predicted delay. The adaptive policy takes into account differing program phases and provides a more accurate prediction of load resolution delays. Knowing the predicted stalls can help the fetch policy to make educated decisions regarding the exclusion of a thread predicted to have a memory dependence. As explained earlier, excluding a thread predicted to have a quickly resolving dependence can hurt performance instead of improving it. The overall goal is for the PE_{mdep} filter to be more lenient to threads with quickly resolving memory dependences.

In the case of a load with an actual long-latency resolution, PE_{mdep} (even with Leniency) still suffers from the problem of temporary thread starvation while waiting for new instructions to make their way down the pipeline. Since the dependence resolution delay is predictable, we can even avoid this temporary period of starvation for a thread. To facilitate this, we grant the excluded thread an *Early Parole* (EP) and start fetching new instructions before the actual resolution of its dependence. With accurate delay predictions, this allows the excluded thread's instructions to arrive in the out-of-order execution core "just-in-time" as the original memory dependence has been resolved. In this fashion, we can avoid clogging the reservation stations with stalled instructions while simultaneously ensuring the timely delivery of new instructions when needed. If the fetch unit receives notification of load dependence resolution before the delay has elapsed, then it immediately returns the excluded thread to its fetch candidate list.

The implementation of Early Parole is easy to implement on top of Leniency. Given a thread's dependence delay prediction, the fetch unit sets a counter for that thread, and decrements the counter on each subsequent cycle. When the counter reaches a threshold θ_{EP}, the fetch unit reinserts the thread into the list of fetch candidates. Note that this may occur before the thread's memory dependence has actually been resolved. When the Leniency threshold is equal to θ_{EP}, then Leniency effectively becomes a special case of EP where the predicted delay is less than θ_{EP} and so the thread would conceptually get excluded

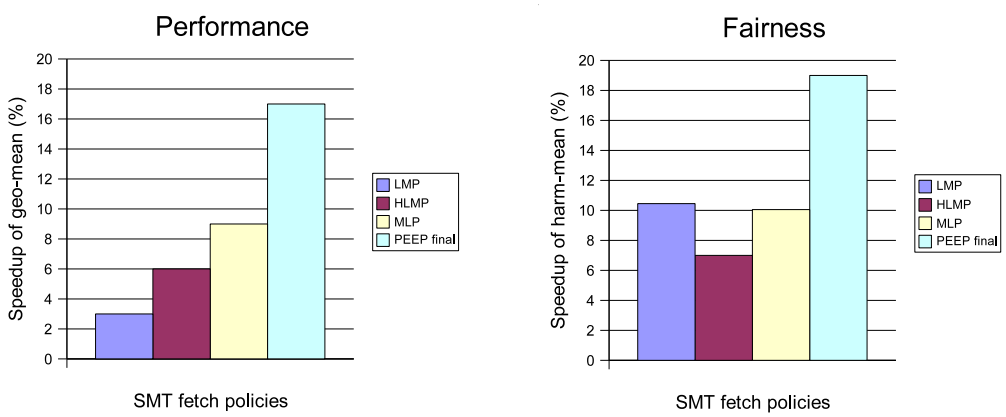


Figure 23: Performance and fairness overview of the SMT fetch policies. $\text{PEEP}_{\text{final}}$ represents our proposed fetch policy after the previously described delay prediction optimization

and then immediately paroled at the same time. In our experiments we found performance was best when both techniques are simultaneously employed, and so for the remainder of this discussion, the designation EP implies both Early Parole as well as Leniency. The next section presents and analyzes the impact of memory-dependence based Proactive Exclusion as well as Early Parole on SMT performance.

4.4 Evaluation

Figure 23 shows a summary of the primary experimental speedup over ICOUNT for PEEP. Compared to the load-miss prediction fetch policy using a simple, single-table load-miss predictor (LMP), a load-miss prediction fetch policy using a hybrid load-miss predictor (HMLP), and the MLP-based fetch policy (MLP), PEEP provides both greater overall performance/throughput (17%) as well as greater fairness (19%) as measured by the harmonic mean of weighted IPC metric [36].

In this section, after explaining our experimental methodology, we will show in steps how the individual techniques of PE and EP can combine to provide this speedup.

4.4.1 Methodology

For all of our experiments, we used the M-Sim 2.0 simulator [69] which is based on SimpleScalar 3.0d [3]. The simulator models an SMT processor modeling up to 8 threads.

Table 4: Simulated four-way SMT processor configuration.

Processor Width	8	Fetch-to-Issue	5 cycles
Scheduler Size (RS)	128	IL1 Cache	64KB, 2-way, 2-cycle
LSQ Size	256	DL1 Cache	32KB, 4-way, 2-cycle
ROB Size	512	L2 Cache	512KB, 8-way, 12-cycle
Integer ALU/Mult	8/4	Main Memory	200 cycles
FP ALU/Mult	8/4	ITLB	16-entry, FA
Store Ports	2	DTLB	32-entry, FA
Load Ports	4		

Table 5: Multi-programmed application mixes used for the four-threaded experiments.

Class	Benchmarks	Suite	Class	Benchmarks	Suite
(4 N)-1	art,bzip2,patricia,equake	F/I/E/F	(3 S,1 N)-1	vortex3,perbmk3,eonc,ammp	I/I/I/F
(4 N)-2	ammp,applu,swim,wupwise	F/F/F/F	(3 S,1 N)-2	eonc,gcc,mesa2,art	I/I/F/F
(4 N)-3	mcf,wupwise,epic,dijkstra	I/F/M/E	(3 S,1 N)-3	fma3d,vortex1,vpr1,ammp	F/I/I/F
(4 N)-4	patricia,lucas,gap,fft	E/F/I/E	(3 S,1 N)-4	crafty,tiffdither,vortex1,patricia	I/E/I/E
(4 S)-1	apsi,eonc,gcc2,perlbmk1	F/I/I/I	(4 L)-1	mcf,equake,art,lucas	I/F/F/F
(4 S)-2	gcc,yacr,twolf,tiffdither	I/P/I/E	(4 L)-2	twolf,vpr2,swim,parser	I/I/F/I
(4 S)-3	gcc2,perlbmk1,eonr,vortex1	I/I/I/I	(4 M)-1	applu,ammp,mgrid,galgel	F/F/F/F
(4 S)-4	twolf,gcc2,anagram,jpegencode	I/I/P/M	(4 M)-2	gcc,bzip2,eonc,apsi	I/I/I/F
(2 S,2 N)-1	vpr1,yacr,wupwise,mcf	I/P/F/I	(4 H)-1	facerec,crafty,perlbmk1,gap	F/I/I/I
(2 S,2 N)-2	apsi,anagram,patricia,dijkstra	F/P/E/E	(4 H)-2	wupwise,gzip3,vortex1,mesa1	F/I/I/M
(2 S,2 N)-3	bc-fact,bc-primes,swim,ammp	P/P/F/F	(2 L,2 H)-1	mcf,equake,mesa2,vortex2	I/F/F/I
(2 S,2 N)-4	gs,parser,fft,galgel	E/I/E/F	(2 L,2 H)-2	parser,swim,crafty,perlbmk3	I/F/I/I
(3 N,1 S)-1	galgel,art,ammp,crafty	F/F/F/I	(2 L,2 M)-1	parser,swim,gcc3,bzip2	I/F/I/I
(3 N,1 S)-2	mcf,lucas,swim,vortex1	I/F/F/I	(2 L,2 M)-2	art,lucas,galgel,gcc	F/F/F/I
(3 N,1 S)-3	art,bzip2,lucas,eonk	F/I/F/I	(2 M,2 H)-1	gzip3,wupwise,fma3d,apsi	I/F/F/F
(3 N,1 S)-4	applu,galgel,mgrid,anagram	F/F/F/P	(2 M,2 H)-2	vortex3,mesa2,mgrid,eonr	I/F/F/I

Table 4 lists the parameters for our simulated SMT processor. The simulator models several microarchitectural features in greater detail than the original SimpleScalar 3.0 model such as explicit register renaming/tracking of physical registers and speculative instruction scheduling with latency prediction. In M-Sim's SMT implementation, the reorder buffer (ROB) and load-store queue (LSQ) are hard-partitioned such that with four threads running, each is constrained to only one fourth of the ROB or LSQ sizes given in the table. This is similar to the Pentium 4's Hyperthreading implementation of SMT [45]. We modified the simulator to include speculative memory disambiguation using a memory dependence predictor based on the load-wait table (LWT) employed in the Alpha 21264 [35]. We also evaluated the final version of our design with an oracle predictor to determine the upper bound on the performance and to determine if the load-wait table predictor is too conservative.

We ran four-thread mixes from a variety of benchmark suites. Our benchmarks belong to the integer (I) and floating point (F) SPEC2000 suites, MediaBench (M), MiBench (E)

and the pointer-intensive applications (P) from Wisconsin [4]. The details of the four-threaded mixes are given in Table 5. For each benchmark, we classify the application as either exhibiting high (H), medium (M) or low-ILP (L), and memory dependence-sensitive (S) or not (N). The ILP-based workload mixes and the classification of memory dependence sensitivity are the same as those used in some other previous studies [71, 70, 79].

Per-workload performance numbers are reported as relative improvements over a baseline machine using the ICOUNT fetch policy with speculative memory disambiguation enabled. Aggregate performance numbers use the geometric mean, which has the property that the geometric mean of speedups is always equal to the speedup of the geometric means. We also quantify the impact of our proposal on overall fairness. To measure fairness, we considered the the harmonic mean of the weighted IPCs for all of the fetch policies. This metric provides a measure of both performance and fairness and is typically used in multi-threading research studies to quantify fairness [36]. We also report the standard deviation of performance degradation experienced by each thread relative to its stand alone IPC (i.e., if the thread had the entire processor to itself). If the standard deviation is large, then that means that some threads experienced larger slowdowns than others, thereby implying that the situation is unfair. This is a stricter measure of fairness as it does not explicitly account for overall system throughput.

For all configurations, we use a load-wait table dependence predictor with 4096 one-bit entries [35]. The table is reset every one million cycles to ensure that the predictor does not become too conservative. In configurations employing a delay predictor, we use a 4K-entry table of 7-bit counters, totaling to 4KB of state overhead when counting both the delay and dependence predictors.

4.4.2 Proactive Exclusion Only

We first present results for Proactive Exclusion (PE) by itself without the effects of Early Parole. All speedup results in Figure 24 are relative to the performance of the standard ICOUNT fetch policy. The load-miss predictor fetch policies have overall speedups of 3% and 6% for LMP and HMLP. Load misses are relatively difficult to predict, and so while

this approach provides some performance benefit, many mispredicted load misses lead to underutilization of the processor resources. The MLP-aware policy is able to expose a higher level of parallelism, which proves to be a better technique than using load-miss prediction, yielding a speedup of 9% over ICOUNT. The general trends are consistent with previously reported results. Applying our memory dependence-based proactive exclusion, the PE_{mdep} fetch policy performs quite well when compared to ICOUNT providing a geometric mean performance benefit of nearly 13% with five mixes showing over 30% performance improvement. Only one mix posts a performance degradation: the (2L-2H)-2 mix suffers from a 4% performance decrease, although this is avoided when we include the delay predictor. PE_{mdep} provides more robust performance gains, while LMP sometimes exhibits some significant losses.

4.4.3 Proactive Exclusion with Early Parole (PEEP)

Some of the benefit of reducing resource contention through PE is offset by the additional delay required to get new instructions from a stalled thread back into the pipeline. The use of Early Parole (and Leniency) significantly helps the problem of a slow restart after dependence resolution. Figure 25 shows the results of adding EP on top of PE_{mdep}. We evaluate delay predictors with conservative (largest delay observed), aggressive (most recent delay observed) and adaptive behaviors. Leniency is invoked if the predicted delay is less than or equal to five cycles (the length of the front-end pipeline), and Early Parole is granted five cycles before the predicted delay has elapsed. The conservative delay predictor provides an overall benefit of 16% over ICOUNT.

Our use of delay prediction only affects the fetch unit, and therefore inaccurate delay predictions only affect resource contention. As a result, the conservative delay predictor sometimes induces more stalling than is necessary. Using the aggressive predictor provides a slight performance advantage with effectively no additional cost or complexity as compared to the conservative approach. PEEP using an adaptive delay predictor provides an overall performance gain of 17%. This slight performance improvement, however, is likely not worth the additional hardware cost of tracking multiple delays per load. It is also interesting to

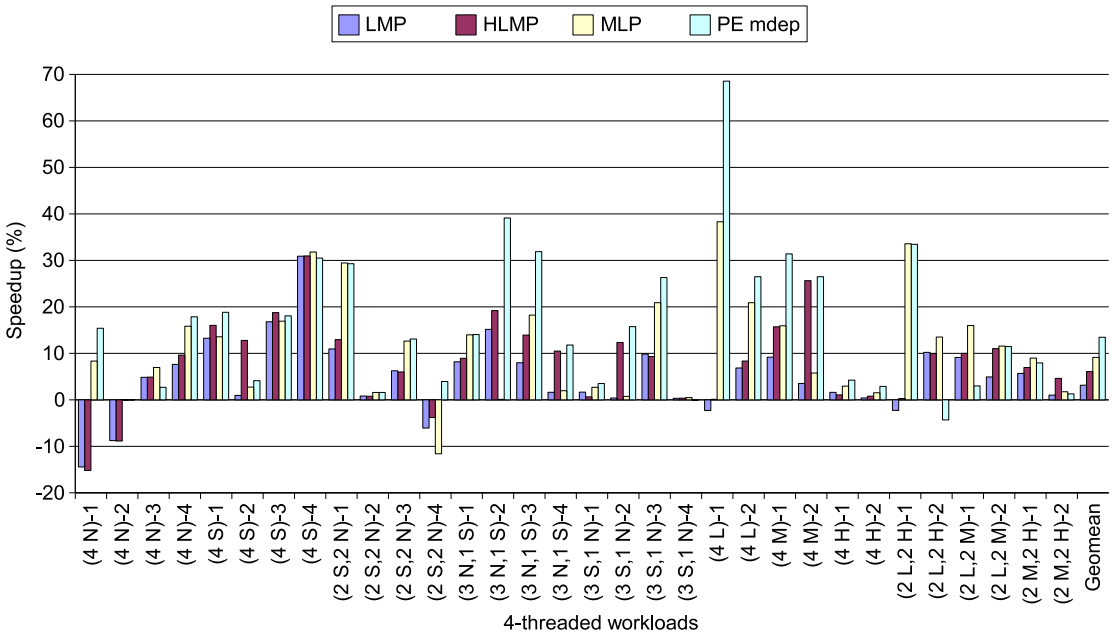


Figure 24: Speedup over ICOUNT of the basic SMT fetch policies as well as the Proactive Exclusion policy based on Memory Dependences without any delay prediction.

note that the one mix that showed a 4% performance degradation without the use of Early Parole now shows a 16% performance improvement with PEEP.

The performance benefits vary among the different workload mixes. For example, when all applications are not sensitive to memory dependencies, such as (4N)-2, then there are practically no long-latency load dependence resolutions for PEEP to take advantage of. In this situation, while we do not gain much performance, PEEP gracefully degrades to the underlying ICOUNT policy. The mix (4L)-1 achieves very large speedups (70%). On further analysis, we observed that there is a huge amount of load port pressure when the ICOUNT fetch policy is used. PEEP and MLP both reduce memory accesses from threads that are predicted to stall, thereby decreasing port contention. The port contention effect is further amplified by M-Sim's accurate modeling of latency-misprediction-induced scheduler replays. Any load directly dependent on an earlier mis-scheduled load (as occurs with pointer-chasing behaviors), may replay one or more times, with each attempt consuming more load issue slots. When the number of load ports was increased from four to eight for

the ICOUNT policy, this performance difference was reduced.

There are some application mixes that need some further detailed analysis. The first of these peculiar mixes is (2S,2N)-1. All configurations involving PEEP manage to provide a performance benefit in excess of 30%. The memory dependence sensitive benchmarks in this mix *yacr* and *vpr* have very predictable dependences and even the simple dependence predictor is able to exploit these. Additionally one of the applications in this mix *mcf* exhibits considerable levels of MLP which helps the MLP-aware algorithm to provide good performance, too. In the case of (3N,1S)-2, we have a good example of the potential performance benefit by accounting for memory dependence prediction to guide fetch of threads. *Vortex* happens to be a benchmark that has a very high memory dependence sensitivity whereas the other applications like *mcf* and *lucas* are highly insensitive to memory dependences. What this means is that the PEEP algorithm is able to very accurately fetch from these insensitive threads while *vortex*' is having a stall and then switch the fetching when the stall resolves. The delay predictors are able to further improve this performance since even the memory dependence resolution delays in vortex are predictable. We observed very few fetch stalls in the processor when the PEEP algorithm was being employed. The MLP-aware algorithm though is unable to exploit the maximum potential of this phenomenon.

To test the upper limit on the performance that PEEP could offer we evaluated both the ICOUNT fetch policy as well as PEEP with an oracle memory dependence predictor. We found that PEEP provides a 19% performance improvement over the ICOUNT fetch policy when an unrealistic oracle predictor is used, which is just 2-3% (depending on the delay predictor) more than when a load-wait table based predictor is used. This indicates that while the load-wait table is a slightly conservative predictor, it is probably sufficient in an SMT processor. These oracle predictor results are also important because they show that the main benefit of PEEP comes from being able to accurately predict the memory disambiguation stalls and their related latencies, rather than the fetch policy somehow indirectly reducing the number of ordering violations. Since the performance of the load-wait table based predictor and the oracle predictor is so close, this shows that the LWT predictor does not cause any significant artificial stalls by making loads wait unnecessarily

on independent stores. If this was the case the performance using the oracle predictor (which as described earlier only identifies true dependences) would have been higher.

An interesting observation is that while memory disambiguation delays are not as long as load miss delays on average, some resolution delays we noted were considerably long. The reason for this is that some store addresses have a dependence on earlier loads that miss in the cache, which in turn can force a later load to stall until the original miss returns and the store can compute its address.

One of the reasons why PEEP performs better than either the LMP or MLP approaches is that proactive exclusion does not kick in until we reach the load with the long predicted dependence resolution latency. Consider a load A that must access main memory, followed by a dependent store B, and then another load C. Store B cannot compute its address until A has returned from memory. In the mean time, if Load C has been predicted to *not* have a dependency on Store B, then it (and its dependents) can execute and make forward progress. In the case of LMP, the fetch policy would have stopped fetching from this thread immediately after fetching Load A, thereby missing out on the ILP available further in the instruction stream. If Load A is an isolated miss, then the same scenario would apply to the MLP-based policy as well. In the case that Load C has been predicted to be dependent on Store B, then the fetch policy would stop at Load C. The pipeline would, however, already contain some instructions past the original stalled Load A. This indirectly creates an effect somewhat similar to our Early Parole in that a small amount of additional work (only up to Load C) is already available in the out-of-order core to prevent starvation when Load A returns from memory. As execution finishes up on these instructions, the Early Parole effect for Load C should allow more instructions to arrive in the execution core just in time to prevent starvation when Load C executes.

4.4.4 Fairness Results

When dealing with multi-programmed workloads, high performance and throughput are not always the only important metrics. In many systems, maintaining fairness among threads is also of great importance. A greedy scheme that only benefits high-ILP applications or

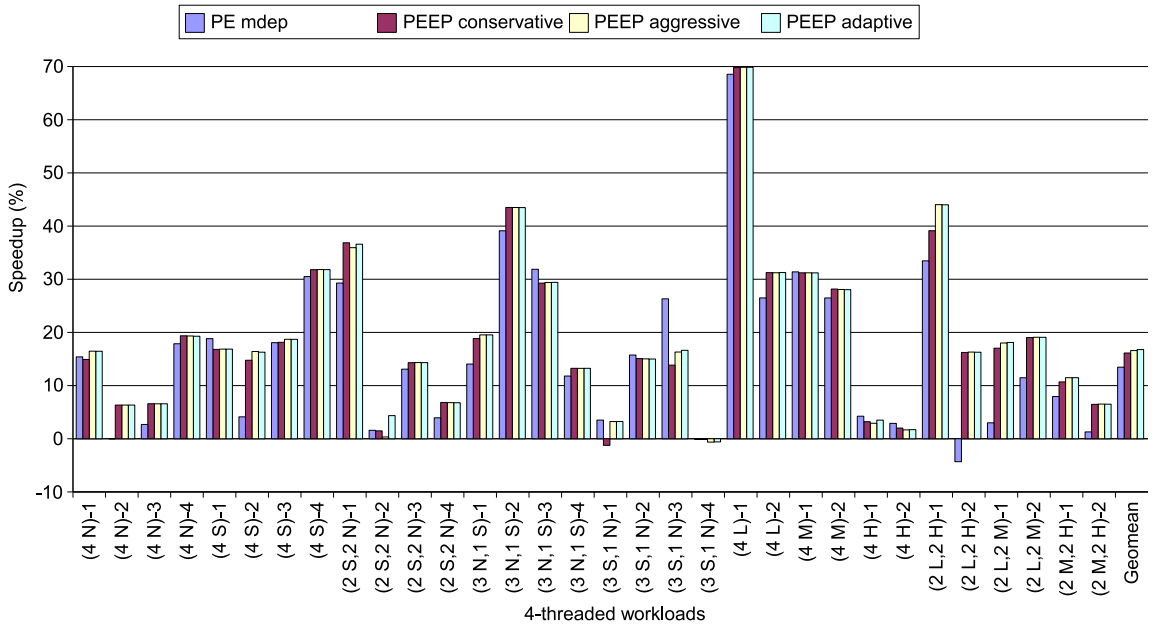


Figure 25: Combining Proactive Exclusion and Early Parole with different delay predictors.

applications having no memory dependences may over-penalize and starve other threads. This is one of the reasons why ICOUNT is considered to be a good policy for an SMT processor. ICOUNT provides relatively high throughput while ensuring that even slow-moving threads get their fair share of resources. Since PEEP runs on top of ICOUNT (or potentially any other fetch policy), PEEP effectively inherits ICOUNT's fairness properties. Immediately after an excluded thread has been reinserted into the candidate list, it would likely have fewer instructions in the pipeline, and therefore the underlying ICOUNT policy will heavily favor fetching from this thread. Figure 26 displays the harmonic mean of weighted IPCs (HMWIPC) for the four-threaded workloads. PEEP delivers greater performance than the other policies and results in better fairness properties as measured by the HMWIPC (20% higher than ICOUNT). The standard deviation of speedup (SDS) metric is a stricter measure of fairness because it does not take performance into account like HMWIPC. Our results indicate that both ICOUNT and PEEP running on top of ICOUNT exhibit very low slowdown standard deviations (0.11 and 0.17 for ICOUNT and PEEP, respectively).

4.5 PEEP on an SMT Processor without Speculative Memory Disambiguation

The PEEP results presented so far demonstrate strong performance gains for a relatively simple modification to basic SMT fetch policies. However, the idea is based on the assumption that an SMT processor has support for speculative memory disambiguation in the first place. While memory disambiguation and load-store reordering are known to improve performance and would likely be included in future processors, there have also been SMT based processors in the past that have not supported it. For example, the Intel Pentium 4 with Hyperthreading supports a version of SMT, but does not execute loads out-of-order ahead of earlier unresolved store addresses [19].

In this section, we evaluate an alternative design based on the main PEEP idea. We allow the SMT processor to make dependency predictions and consequently predict load dependence resolution delays, and we use PEEP to control the fetch policy. However, *controlling fetch is the only thing these predictions are used for.* The out-of-order execution core does *not* allow loads to issue in the presence of earlier unresolved stores, even if the load has a predicted delay of zero. Delay still has the same definition as in Section 4.3; that is, the time from when a load's address has been computed until it actually issues. Any predicted non-zero delay counts as a predicted memory dependence, but Leniency is still used to prevent exclusion when the delay is less than the given threshold (five cycles for our experiments). In an SMT processor without speculative memory disambiguation, load instructions will on average have to wait longer for all earlier store addresses to resolve. Employing PEEP in this context does not attempt to reduce this latency; it cannot. Rather, PEEP only targets the reduction of scheduler congestion effects when such stalls are present. The intuition here is that a large fraction of the performance benefit comes from being able to predict when these stalls occur and more importantly how long they take to resolve. These two pieces of information are what guide the processor to make intelligent decisions about which threads to fetch from, which threads to throttle and for how long. The predictor cost is reduced slightly since we no longer need to keep the one bit entry corresponding to the load wait table. The remaining delay predictor counter still consumes 3.5KB of state. The

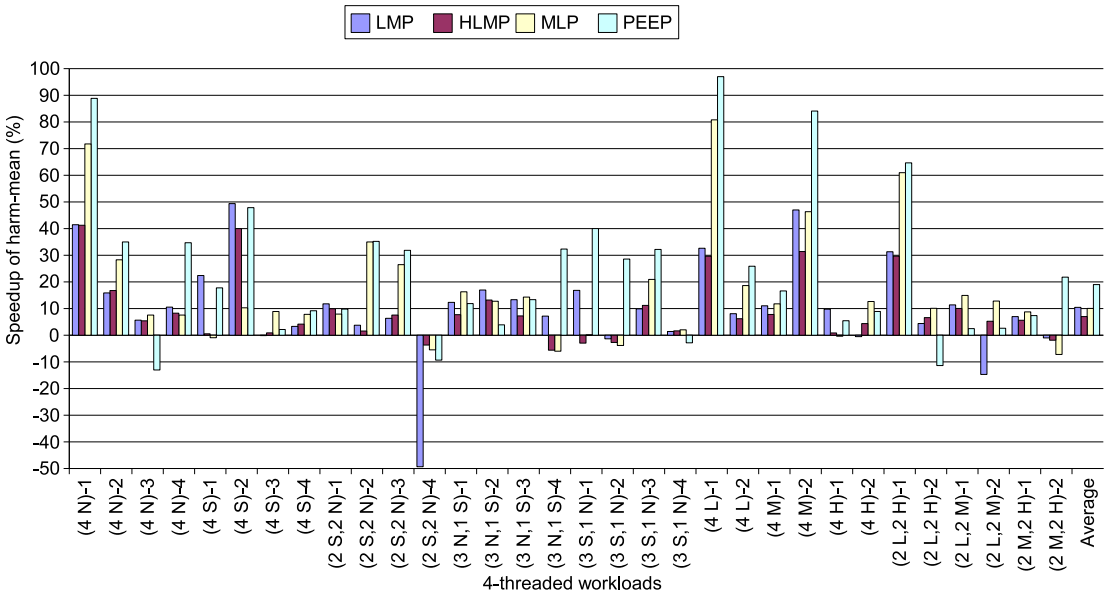


Figure 26: Fairness as measured by the harmonic mean of weighted IPC.

primary advantage is that we can potentially keep most of the benefits of PEEP without requiring the pipeline to implement speculative load-store reordering, especially in an SMT processor that does not already incorporate the same. In the remainder of this section, We refer to this version of PEEP without speculative load reordering as PEEP*.

Figure 27 shows the performance of PEEP* compared to a baseline SMT processor using ICOUNT, also without speculative memory reordering. While not allowing loads to issue out-of-order with respect to earlier unresolved stores potentially reduces ILP, it helps to avoid pipeline flushes on mispredictions. The relative results are very close to the original PEEP design indicating that a fetch policy that is cognizant of impending memory disambiguation stalls helps to improve the throughput of an SMT machine considerably. From these results, we draw two conclusions. First, predicted memory dependences are very effective indicators of impending stalls that an SMT fetch policy should be aware of. Second, the benefits of our proposed technique are purely a result of better fetch management and they are not dependent on any explicit support for speculative memory disambiguation in the out-of-order core.

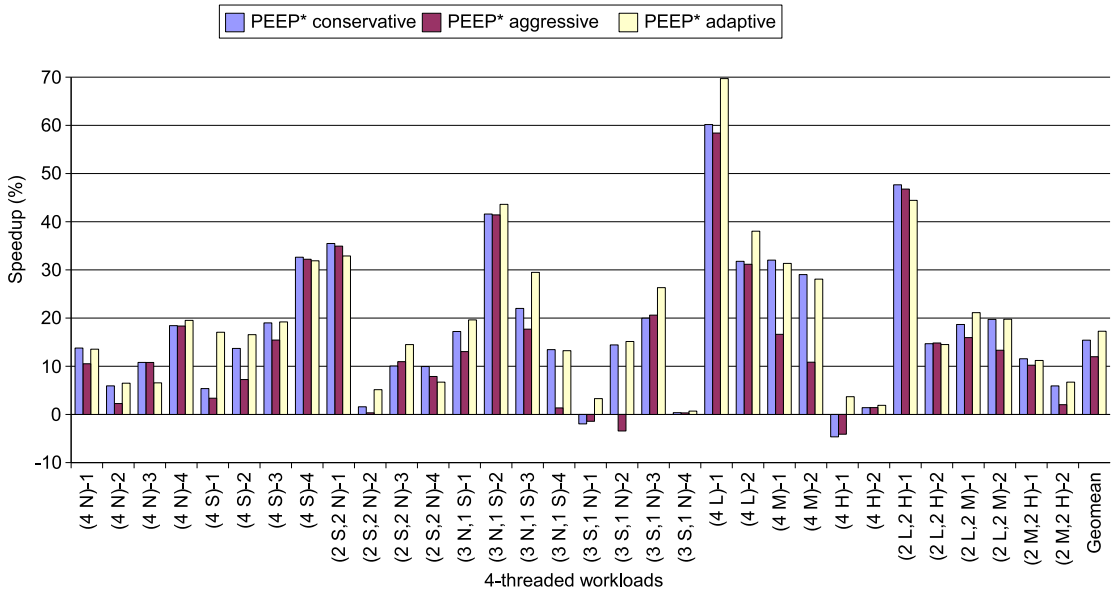


Figure 27: Benefit of memory dependence prediction without speculative load disambiguation.

4.6 Sensitivity Analysis

So far, we have explained and demonstrated the merits of PEEP/PEEP* on a specific set of benchmarks and microarchitectural assumptions. In this section, we present additional experimental results to show that PEEP can provide strong results across a range of microarchitectural assumptions. We also study the impact of PEEP in conjunction with other fetch policies.

4.6.1 Sensitivity to Multithreading

Our baseline four-threaded SMT microarchitecture was chosen to provide a moderate level of aggressiveness. We now study the impact of scaling down the processor configuration and workloads. We consider a smaller processor operating only on two-threaded workloads. Table 6 shows these workloads, which combine the same classes of applications as before. The processor configuration is similar to that described earlier in Table 4, with per-thread resources (e.g., ROB entries) reduced to handle only two threads. Figure 28 shows the corresponding performance results for LMP, HLMP, MLP, and PEEP/PEEP*.

By its nature, PEEP throttles the fetch of a thread predicted to have dependencies

Table 6: Two-threaded workloads.

Classification	Benchmarks	Suite	Class	Benchmarks	Suite
(2 N)-1	art,bzip2	F/I	(2 L)-1	mcf,equake	I/F
(2 N)-2	ammp,applu	F/F	(2 L)-2	twolf,vpr2	I/I
(2 N)-3	mcf,wupwise	I/F	(2 M)-1	applu,ammp	F/F
(2 N)-4	patricia,lucas	E/F	(2 M)-2	gcc,bzip2	I/I
(2 S)-1	gcc2,perlbmk1	I/I	(2 H)-1	facerec,crafty	F/I
(2 S)-2	gcc,twolf	I/I	(2 H)-2	wupwise,gzip3	F/I
(2 S)-3	apsi,eonc	F/I	(1L,1H)-1	mcf,mesa2	I/F
(2 S)-4	gcc,yacr	I/P	(1 L,1 H)-2	parser,crafty	I/I
(1 S,1 N)-1	vpr1,wupwise	I/F	(1 M,1 H)-1	gzip3,fma3d	I/F
(1 S,1 N)-2	apsi,patricia	F/E	(1 M,1 H)-2	vortex3,mgrid	I/F
(1 S,1 N)-3	bc-fact,swim	P/F	(1 L,1 M)-1	parser,gcc3	I/I
(1 S,1 N)-4	parser,galgel	I/F	(1 L,1 M)-2	art,galgel	F/F

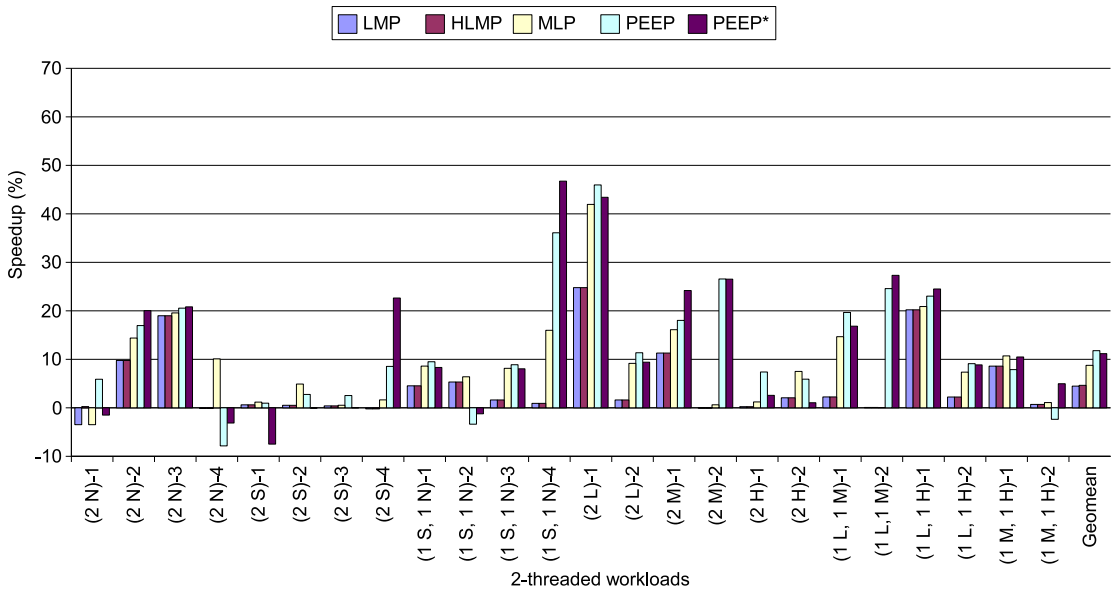


Figure 28: Performance of the SMT fetch policies for two-threaded workloads.

while allowing other independent threads to utilize all the processor resources. This means that with more threads, PEEP has more opportunities to fetch from a non-stalled thread. Conversely, with fewer threads such as in the case of our two-thread workloads, if one (or worse both!) threads suffer from a memory-dependence related stall, then there are far fewer options for finding useful work to cover the dependency resolution latency. As a result we do not expect PEEP to provide the same level of performance gains in this scenario. Figure 28 shows, however, that the results are not too discouraging either. The reason is that not all stalls occur simultaneously. If only one of the two threads has a predicted memory dependence, then PEEP allows the other thread to make use of practically all of the processor's resources. Only when both threads have predicted memory dependences will the processor's throughput be severely reduced, which does not happen too often.

4.6.2 Scalability of PEEP

In this study, we evaluated PEEP/PEEP* on a larger, more aggressive SMT microarchitecture. Figure 29 shows the performance evaluation. We modeled a four-way SMT configuration containing a 2048-entry ROB, 512-entry LSQ, 300-entry RS and 10 memory ports. Larger instruction windows allow the processor to consider more instructions per thread, which in turn increases the likelihood that some load instructions will be stalled waiting on ambiguous memory dependences. In this context, it becomes even more critical for the SMT fetch policy to account for these dependences to avoid tying up processor resources with those instructions in the forward slices of the stalled loads. As seen from the figure, using PEEP and PEEP* for this large SMT processor configuration yield average throughput improvements of 20% for both approaches. Additionally the overall trends are very similar to those reported in Figures 25 and 27. As machine sizes scale upward, however, the difference between PEEP and MLP decreases as larger windows expose more memory-level parallelism.

4.6.3 PEEP Combined with Other Fetch Policies

We now examine the performance of PEEP used in conjunction with the LMP and MLP policies. When combining policies, we model a logical "OR-ing" of policies. That is, the

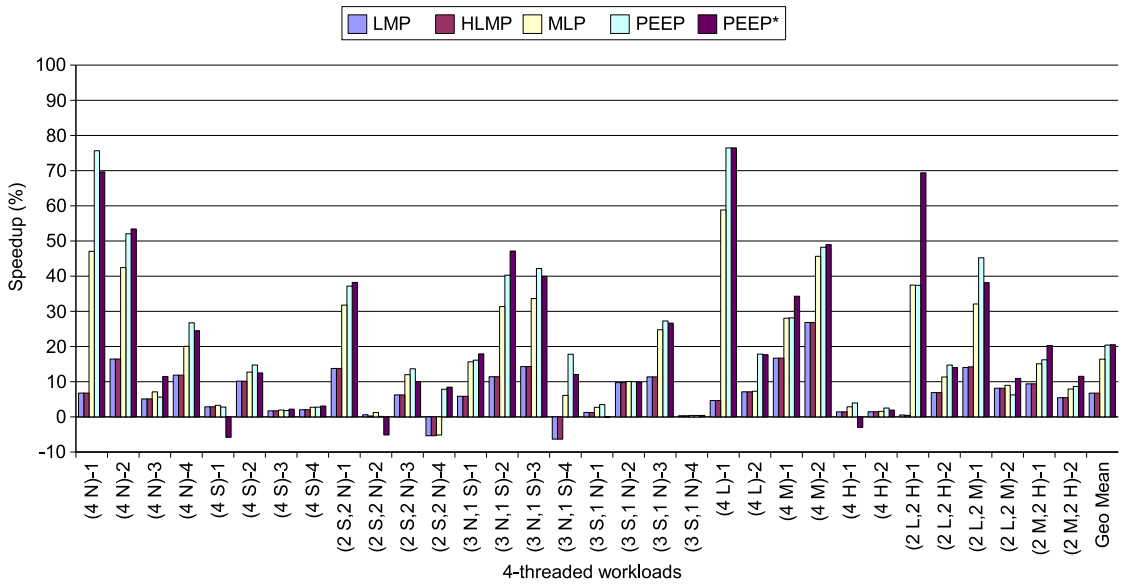


Figure 29: Performance of the SMT fetch policies on a large, aggressive processor configuration

aggregate policy removes a thread from fetch consideration if any of its constituent policies chooses to exclude the thread. The delay predictor is still used for PEEP to determine the number of cycles the thread would be gated for in case of memory-dependence related stalls. The results are presented in Figure 30. As expected, most of the performance benefit comes from PEEP and in most of the workloads the performance is nearly the same as that of PEEP with an adaptive delay predictor. Since load misses are hard to predict we do not see the PEEP+LMP technique have much of an advantage over PEEP. In some workloads, PEEP+MLP is able to predict an increased number of stalls, thereby making this policy work better. Finally we also simulated all three: PEEP along with a load-miss predictor and an MLP-aware policy; this hybrid policy seemed to marginally outperform just the PEEP adaptive policy.

4.6.4 Delay Sensitivity

The effectiveness of the PEEP fetch policy may be impacted by the accuracy of predicted delay, and the responsiveness of the PEEP mechanisms may also be sensitive to the pipeline depth of the processor frontend.

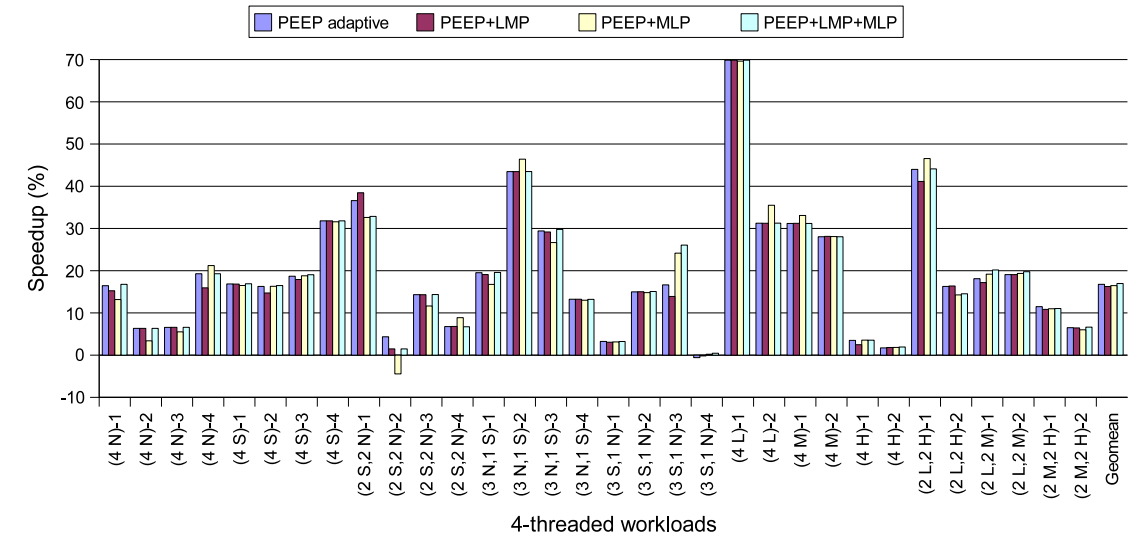


Figure 30: Performance of PEEP combined with other fetch-gating policies

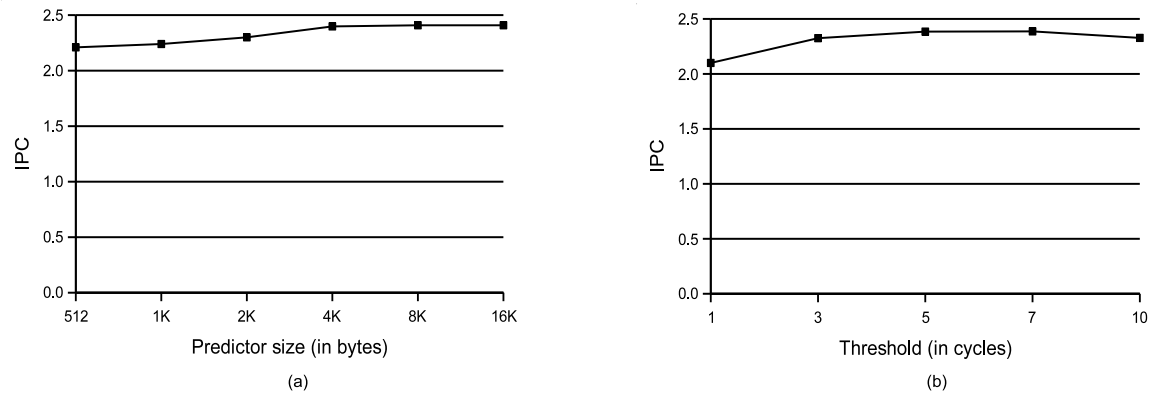


Figure 31: Sensitivity of the policy to different predictor sizes and delay thresholds

We first study the impact of the delay predictor's size. A smaller predictor may lead to more frequent conflicts in the predictor tables, thereby leading to inaccurate delay predictions. Figure 31(a) shows that increasing the size of the delay predictor beyond 4KB does not increase performance by any appreciable amount. Decreasing the predictor size below 4KB causes performance to drop off due to frequently inaccurate delay predictions.

Next, we study the impact of the delay threshold θ_{EP}. Figure 31(b) shows the IPC impact as we vary the threshold. Not surprisingly, the optimal setting for θ_{EP} is equal to the length of the frontend pipeline (5 cycles/stages). If the predicted delay is less than

the pipeline length, then it intuitively makes sense that it will take longer to exclude and then reinstate the thread. Similarly, to minimize the impact of waiting for dependents to get dispatched into the out-of-order core after a load's memory dependence has been resolved, an early parole of a number of cycles equal to the frontend pipeline depth allows the dependent instructions to arrive "just in time." This assumes that the fetch unit fetches from the reinstated thread immediately after it gets returned to the candidate list, but this is very likely as the underlying ICOUNT mechanism will likely favor this thread.

We also explored the impact that the front-end pipeline depth had on the various fetch policies. Since the fetch policies described and evaluated through this chapter (including PEEP and PEEP*) are fetch-gating policies, it is important to see how these policies behave with shorter and longer pipelines. When we fetch-gate a thread and the processor has a long front-end pipeline, this latency would be exposed when the thread is refetched. For the LMP and MLP policies, this latency could result in a significant performance loss if the issue queue is not filled with useful work from other threads. However for PEEP/PEEP*, since we have an Early Parole mechanism that starts refetching some cycles before the instructions are actually needed, and the threshold is decided based on the front-end pipeline depth, long-latency pipelines should not impact the performance too much. In Figure 32 we see the results of all the policies evaluated. We see that ICOUNT itself performs quite poorly when the pipeline depth is increased. As a result the other fetch policies have more or less the same performance improvement over ICOUNT as observed for the base five cycle latency from fetch to issue, however both PEEP and PEEP* are able to improve their performance over ICOUNT due to the delay predictor in place.

4.7 Evaluating PEEP in an Industrial Simulator

In this section, we study the performance of PEEP in an industrial simulator. We use an execution-driven, cycle-level IA-32 simulator that models an aggressive, out-of-order, two-threaded superscalar processor. The simulator model has been validated against hardware for correctness and accuracy and is currently used by a leading processor manufacturer to evaluate processor designs. The simulator models a two-threaded SMT processor with

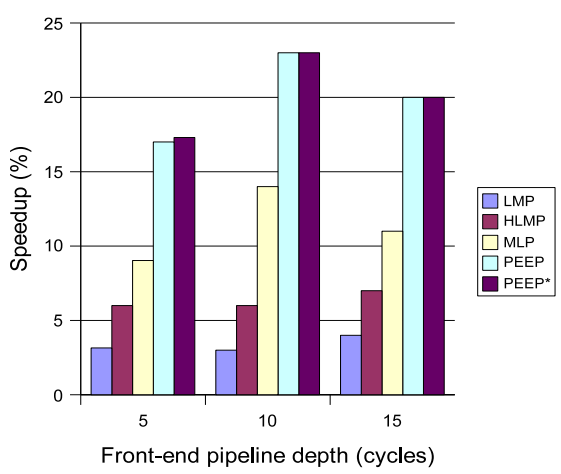


Figure 32: Performance of the SMT fetch policies with larger front-end pipelines

a 128-entry reorder buffer, 48-entry load queue, 32-entry store queue, and 54 reservation stations. The memory dependence predictor incorporated in the simulator is an advanced memory dependence predictor that tracks per-load dependence history and uses saturating counters and confidence thresholds to make predictions regarding ambiguous memory dependences [19]. In this simulator, we incorporated PEEP as a fetch policy filter that runs on top of the base fetch policy which is a round-robin based policy. We analyzed over 150 benchmarks for this study which are taken from the Multimedia (Mult), Server(S), Office(O), Digital Home(DH), and Workstation (W) benchmarks suites. We further divide these benchmarks into three categories based on their sensitivity to memory dependences as being either non-sensitive (NS), medium-sensitive(MS) or highly-sensitive(HS). To form the two-threaded workloads we form the following types of benchmark pairs: NS-HS,NS-MS,MS-HS,HS-HS. We present results for ten pairs from each type (40 total benchmark pairs). Due to the sensitivity of the information, we are unable to present the individual benchmark names in this dissertation.

Figure 33 presents the performance results for PEEP for 40 workload mixes relative to a baseline that does not incorporate PEEP. This figure shows that on average PEEP has an 8.4% performance improvement over ICOUNT when implemented in the industrial simulator. Out of all the 40 simulated workloads, 5 show performance degradations. Only one mix, (1 NS,1HS)-10 experiences more than 5% performance degradation. As indicated

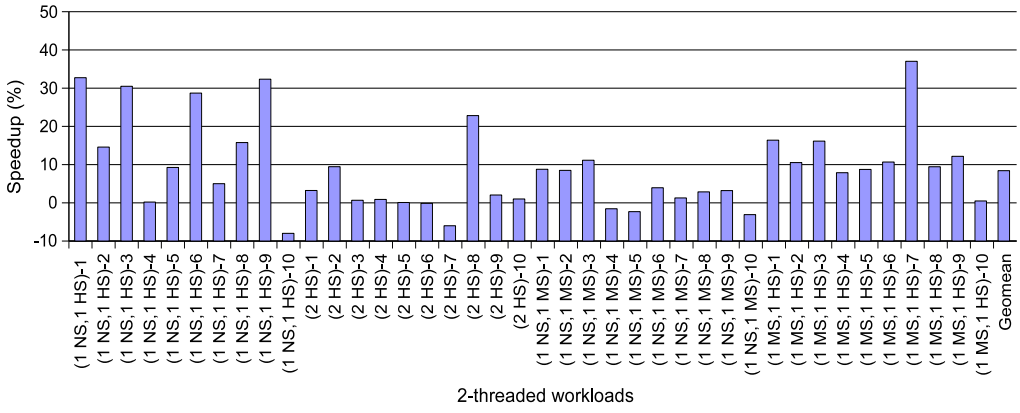


Figure 33: Per-mix performance results as obtained in an IA32 simulator

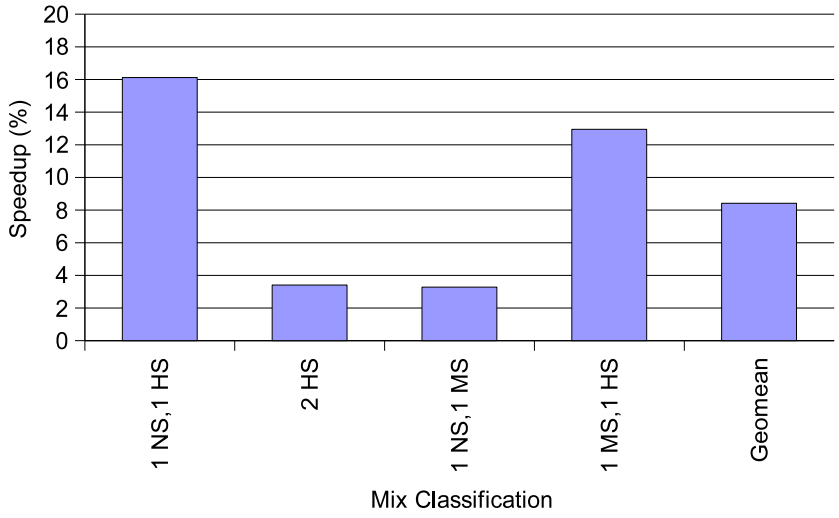


Figure 34: Aggregate performance results per-mix type

in Figure 34, the best performing mixes are the ones where one application is a sensitive application and one is a non-sensitive application. This observation is consistent with the results reported in the M-Sim simulator.

4.8 Conclusions

In this chapter, we have exploited the common-case behavior of predictable memory dependences in programs to improve the performance of SMT processors thereby improving their efficiency. We used the concept of Proactive Exclusion (PE) to remove threads from an SMT processor's fetch unit's work list when threads are likely to experience long-latency stalls. We also proposed an Early Parole (EP) mechanism to restart fetching of previously excluded threads in an anticipatory fashion such that the instructions arrive at the out-of-order execution core right as the previous stall resolves. In particular, we took advantage of the predictability of load-store memory dependences as well as the predictability of their resolution delays. While our design focused on memory dependence prediction, the PE and EP techniques can potentially be applied to any stalls in SMT processors that are easily predictable. We analyze the performance of PEEP with both a simple load-wait table based predictor as well as an oracle predictor to help determine efficient performance/cost trade offs.

PEEP targets the fetch engine of an SMT processor, and augments it with predictable stall information so that the fetch unit is able to prioritize threads accordingly. This enables the threads to make maximum use of available resources, which considerably improves overall throughput and performance. Furthermore, we see that PEEP does not compromise on fairness in the fetch policy as indicated by the harmonic mean speedups and the standard deviation of speedups for all the policies. PEEP accomplishes this with the use of a simple, light-weight memory dependence predictor which keeps the design complexity low. All these factors allow PEEP to be an effective design for resource management in SMT processors which enables it to improve overall processor efficiency.

CHAPTER V

FIRE-AND-FORGET: IMPLEMENTING SCALABLE STORE TO LOAD FORWARDING

5.1 Introduction

In this chapter, we focus on the issue of scalable data forwarding to improve the scalability and reduce the power consumption of the processor while minimizing the performance impact [78]. Conventional implementations of the load queue and the store queue, which support out-of-order memory scheduling and data forwarding, are CAM-based and scale poorly for the sizes required for current and future large-window, high-ILP processors. They are inefficient due to the increased power consumption as well as the impact on access latency, which could in turn impact the clock frequency of the processor. In our work, we present a design for these queues that exploits the common-case behavior of predictability in data forwarding patterns. Our approach to designing the load and store queues allows a more scalable and power-efficient implementation and mitigates many of the problems that impacts conventional implementations of these critical structures. We present results that show that the Fire-and-Forget technique of store-to-load forwarding provides power and energy savings as well as a slight performance improvement over the conventional forwarding approach which helps to improve overall processor efficiency.

Our main technique is to predict the relative location of a store's data-dependent load (if such a load exists at all) and speculatively forward the data. After forwarding to a predicted load queue entry, the store never again worries about data forwarding and simply waits to commit and write its result back to memory. Hence, we call the technique "Fire-and-Forget." In addition to simplifying the memory scheduling logic, this approach enables a store to forward a value before its corresponding store address has been computed and/or before the receiving load entry's effective address has been computed, thus enabling memory cloaking. As a result, our Fire-and-Forget memory scheduling microarchitecture

provides better performance than previously proposed techniques for simplifying the load and store queues, and at the same time Fire-and-Forget requires substantially less hardware. Specifically, our base design supports a load queue which requires no CAM-based search logic, and it **completely eliminates the store queue**. We also present am optimized Fire-and-Forget design which does not even require a load queue.

The rest of the chapter is organized as follows: The next section provides a brief review of current load and store queue implementations and describes the most recent proposals for simplifying the hardware. Section 5.3 describes our Fire-and-Forget forwarding mechanism, and Section 5.4 provides our performance evaluation. Section 5.5 presents some case studies of applications that have interesting behavior. Section 5.6 briefly describes the energy and power impact of using Fire-and-Forget over the traditional associative queues. Section 5.7 and Section 5.8 present two optimizations to the basic design of Fire-and-Forget which help improve its performance as well as eliminate the load queue. Section 5.9 compares Fire-and-Forget with another LQ/SQ-less data communication mechanism and Section 5.10 presents a summary of our findings and discusses future work in this area.

5.2 Background

5.2.1 Conventional Load and Store Queues

A conventional processor partitions the memory scheduling logic into two separate queues: the load queue (LQ) and the store queue (SQ). Load instructions wait in the load queue until the processor determines that the load's dependencies have been resolved and that the load is ready to issue. When a load issues, it retrieves a value from the data cache, and it also performs an associative search of the store queue (to be described shortly) to check if there are any earlier stores to the same address that have not yet written their results back to the cache. Store instructions wait in the store queue for their effective address and data operands. When both address and data become ready, the store broadcasts both of these to the load queue. Every valid load queue entry monitors the store broadcast bus, and captures the forwarded data value if its address matches the store's and it comes

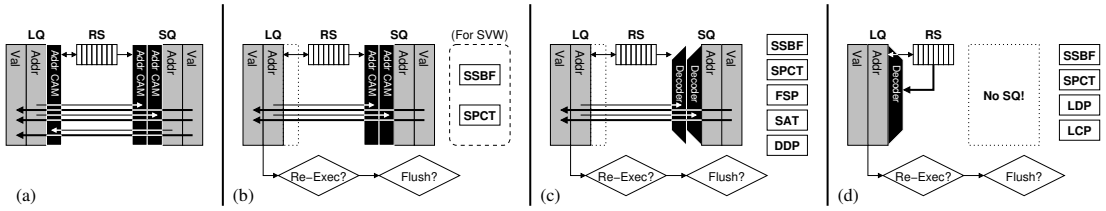


Figure 35: Memory scheduler designs: (a) conventional fully-associative load and store queues, (b) non-associative load queue with optional SVW filtering, (c) Store Queue Index Prediction, and (d) Fire-and-Forget store forwarding.

later in program order than the store.[1] In a conventional LQ/SQ implementation, both stores searching for data-dependent children and loads searching for parents require content addressable memories as illustrated in Figure 35(a). It is important to note that the LQ/SQ CAMs are much more complex than the CAMs used in the reservation stations (RS) for scheduling non-memory instructions. In particular, the RS CAMs match on relatively small tags ($\lceil \log_2 \text{Physical RF Size} \rceil$, typically 6-8 bits) whereas the LQ/SQ typically match on memory addresses (32-64 bits), although tricks such as staggered/partial tag matching and virtual-indexing/physical-tagging can be used to reduce the overhead. Furthermore, the LQ/SQs also broadcast and compare some form of age identifier because there may be more than one load (store) from (to) the same address.

5.2.2 Load Re-Execution

An alternative to the fully-associative load and store queues is *load re-execution* prior to commit. The idea is to allow loads to speculatively issue out of program order, and then re-execute the load later in the pipeline to verify the results of the speculative execution. When a load commits, it is the oldest instruction in the processor, implying that any earlier stores have already written their results to the cache hierarchy.[2] Therefore, any value retrieved from the cache hierarchy at commit will be correct. If the re-executed load value matches the preliminary speculative value, then the load's dependents also received the correct value

[1] Some LQ designs assume that it does not hold the forwarded value. Instead the load reads it from the store queue and propagates it to its dependents. For the purpose of this work we assume that the LQ has additional storage for holding the data value. We also account for this extra storage in our hardware budget calculations.

[2] For a superscalar processor that commits multiple instructions per cycle, this commit-time re-execution also needs to check for any earlier stores retiring in the same cycle. However, the complexity of this logic is quite reasonable as it only needs to scale with the commit width rather than the LQ or SQ sizes.

and the processor's state is correct. If the re-execution reveals an inconsistency in the load's value, however, then it is possible that its dependents have also consumed incorrect values. In this situation, we flush the pipeline and refetch the instructions following the load. An interesting aspect of load re-execution is that it is a *value-based* approach [13]. Loads that execute in an order that would normally be considered as an ordering violation or incorrect with respect to the consistency model may still be treated as "correct" if re-execution reveals that the speculative *value* of the load is the same as the real value. This may occur due to Silent Stores [40] or just dumb luck (e.g., many values in a processor are zero [88], and so reading a zero from the wrong store, the wrong address, or anywhere else may still result in the correct value!).

Procrastinating the load checking until commit removes considerable complexity from the LQ and SQ hardware. However, naïve load re-execution forces every load to effectively issue twice. This places a large burden on the limited data cache ports and tends to have a substantial negative effect on performance due to pipeline flushes and cache port contention. Much work based on load re-execution has focused on reducing the complexity of the LQ and/or SQ while attempting to reduce the overhead of re-execution. We now present a brief review of techniques based on load re-execution. We describe several works which progressively remove more and more of the LQ and SQ hardware.

5.2.2.1 Non-Associative Load Queue

To address the power concern of the load queue's associative search, Cain and Lipasti proposed the Non-associative load queue (NLQ) [13] which uses load re-execution. When a load issues, it performs an associative search on the store queue to find any earlier stores to the same address, with a hit resulting in data being forwarded from the matching store to the load. However, a store does not "execute" in that even when its address and data are ready, the store makes no attempt to communicate this information to other instructions. The store queue entry simply holds on to its operands until it reaches commit, and then writes its value to the data cache. As a result, the NLQ approach removes all of the associative CAM logic from the LQ, as shown in Figure 35(b). The NLQ work also proposed a simple

heuristic to reduce the re-execution overhead. In particular, a load that issues when there were no unresolved store addresses will have received the correct value, and therefore need not be re-executed at commit. Cain and Lipasti also proposed two additional heuristics that guarantee correct operation with respect to multi-processor consistency.

5.2.2.2 Store Vulnerability Window

To improve on the non-associative load queue design, Roth proposed the idea of a Store Vulnerability Window (SVW) [59]. The SVW technique is a re-execution filter based on the observation that when a load issues, the processor only contains a small number of stores that could even potentially conflict with the load. For each load, this technique tracks this small set of stores (i.e. its SVW), and a load only re-executes at commit if any store from its SVW had executed after the load. Any other store not in the load's SVW will not cause re-execution because, by construction, the store cannot have a conflict with the load. For the purposes of our work, the reader can simply consider SVW as a very effective means of reducing load re-executions when using a NLQ (Roth showed an 85% reduction).

To implement the SVW scheme, Roth proposed to assign each store instruction a unique, monotonically increasing identifier called the *store sequence number* (SSN). In practice, the SSN will have a finite bit-width and some mechanism for handling SSN overflow/wrap-around. For each dynamic load, its vulnerability window can be represented by the SSN of the youngest older store that already committed when the load was dispatched (i.e. the most recent store to which this load is not vulnerable). A store with a lower SSN (an older store) has already committed its results to the cache prior to this load instruction entering the out-of-order core of the processor, and therefore the load cannot be vulnerable to this store. In addition to defining a SVW for each load, the overall SVW scheme also makes use of an address-based *store sequence Bloom filter* (SSBF) that conservatively filters unnecessary load re-executions. The SVW scheme also used a Store PC Table (SPCT) that performs memory dependence prediction to reduce the frequency of load-store ordering violations. It is important to note that the auxiliary structures (SSBF and SPCT) are all off of the critical load and store execution paths and therefore do not impact the scalability of the LQ and SQ.

5.2.2.3 Store Queue Index Prediction

SVW provides a means of implementing the non-associative load queue (NLQ) while keeping the re-execution overhead well under control. However, the SVW_{NLQ} (SVW on top of NLQ) organization still requires associative search logic in the store queue. In particular, when a load executes, it still searches the SQ to find earlier stores that have already computed their addresses from which data may be read. Sha et al. proposed a novel approach called *Store Queue Index Prediction* (SQIP) that works on top of SVW_{NLQ} to remove the associative logic from the store queue [67]. Prior work on memory dependence prediction has already shown that there are stable patterns between loads and the store instructions they receive their values from [52, 17, 79]. That is, a static load that is dependent on an earlier store is usually dependent on the same static store for each dynamic instance of that load, and furthermore the relative position of the store will also be the same (i.e., the load is always dependent on the i^{th} previous store). As shown in Figure 35(c), SQIP replaces the SQ's costly associative logic with much simpler direct-mapped indexing.

Store queue index prediction (SQIP) starts with SVW_{NLQ} with all loads directly reading their values from the data cache (no SQ search). SQIP uses two tables to track loads and stores: the *Forwarding Store Predictor* (FSP) and the *Store Alias Table* (SAT). The FSP is a set-associative table indexed by the PC of a load instruction, where each entry stores a small set of partial PCs for store instructions that recently forwarded values to the load. The FSP also contains valid bits, partial tags, and a training/confidence counter. The SAT is a direct-mapped untagged table indexed by a store's PC. A SAT entry contains the SSN of the most recent store to hash to the entry. The Store PC Table (SPCT) tracks pairs of loads and stores that have had ordering violations. Similar to the underlying SVW scheme, SQIP's extra hardware structures (FSP, SAT, etc.) are all accessed off of the critical path of load-store execution.

An Example: For each part of the example in Figure 36, we only include the fields of the SQ and LQ entries that are used in that part. Figure 36(a) shows that when a store

commits, it indexes the SPCT with the address it wrote to (Y) and records its own PC (4000). Next, a later load that should have received its data from the store at PC=4000 re-executes at load commit and reveals that it did indeed receive the wrong value (b). The load now reads the SPCT to find out which store it should have received a value from, and then records the identity of this store in its FSP entry. A later instance of the same store (c) writes its SSN into the SAT during the register rename stage. The SAT is not a large structure, but it requires additional ports and checkpointing to repair it after pipeline flushes. When we next encounter the load (d), the load checks the FSP and finds that it had a previous conflict with the store at PC=4000. The load then accesses the SAT to find the SSN of the most recent instance of the forwarding store (SSN=27). From this SSN, the load can directly compute the store queue index (SQI) by taking the SSN modulo the SQ size (e). When the load issues, it can use the predicted SQ index to directly access a single SQ entry (i.e. no associative search). In this case, the load's address matches the address in the predicted store queue entry, and so the load uses this value instead of the value in the data cache. Note that underneath the SQIP scheme, SVW is still running to guarantee the overall correctness of execution. Sha et al. also proposed a Delay Distance Predictor (DDP) that reduces the frequency of pipeline flushes due to difficult-to-predict loads. Overall, they demonstrate that SQIP performs within 0.6% of a realistic fully-associative SQ.

5.2.3 Other Related Work

Our Fire-and-Forget approach can be viewed as a logical progression in the line of load/store processing optimizations as illustrated in Figure 35(d). However, there have been other proposed techniques for optimizing the load and store scheduling structures. Address-Indexed Memory Disambiguation (AIMD) uses a structure that acts like a small, speculative L0 cache [76]. Stores speculatively write to this cache along with sequence numbers that are similar in spirit to Roth's SSNs, and ordering violations can be detected which forces loads to re-execute. Note that with AIMD, the load execution occurs in the main out-of-order core which requires already issued instructions to somehow be put back into the scheduler. Also, the use of sequence numbers alone does not allow for the relaxation to the

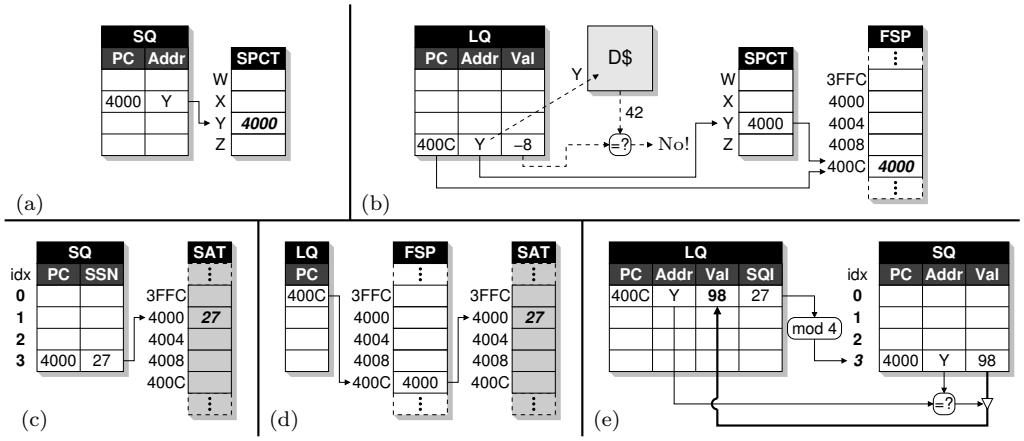


Figure 36: Store Queue Index Prediction example illustrating (a) a store updating the SPCT, (b) a misforwarded load using the SPCT to update the FSP, (c) a later instance of the same store updating the SAT, (d), a later instance of the load using the FSP and SAT to compute a SQ index, and (e) the load using the predicted index to directly access the predicted SQ entry.

value-centric approach of commit-time load re-execution. Sethumadhavan et al. proposed a partitioned LSQ design [65]. Each segment of the LSQ still uses CAMs for address matching, but having fewer entries per segment relaxes timing constraints. They also use Bloom filters to filter unnecessary CAM activity when it can be proven that a match will not occur. Previous work has also capitalized on the observation that few stores actually forward their values. Roth [58] and Baugh and Zilles [8] independently proposed store queue organizations that partition the SQ into one piece that buffers stores for in-order retirement, and a smaller piece that forwards values to loads (the forwarding store queue or FSQ). The expensive CAM logic is limited to only the small FSQ, which keeps the hardware costs under control. Some form of load re-execution is still required to guarantee correct load execution. The difficulties with address-indexed LSQs are that address/bank/port conflicts can occur, allocation of entries is more complicated as it typically occurs at execution, and finding the right instructions to squash on a pipeline flush is not straightforward. Recently, Sha et al. proposed a design in which store-load forwarding can be performed without a store queue [68]. We will discuss this proposal in some detail in Section 5.9 since the goals of this work and Fire-and-Forget are quite similar; however, the intuitions used and designs explored are quite different.

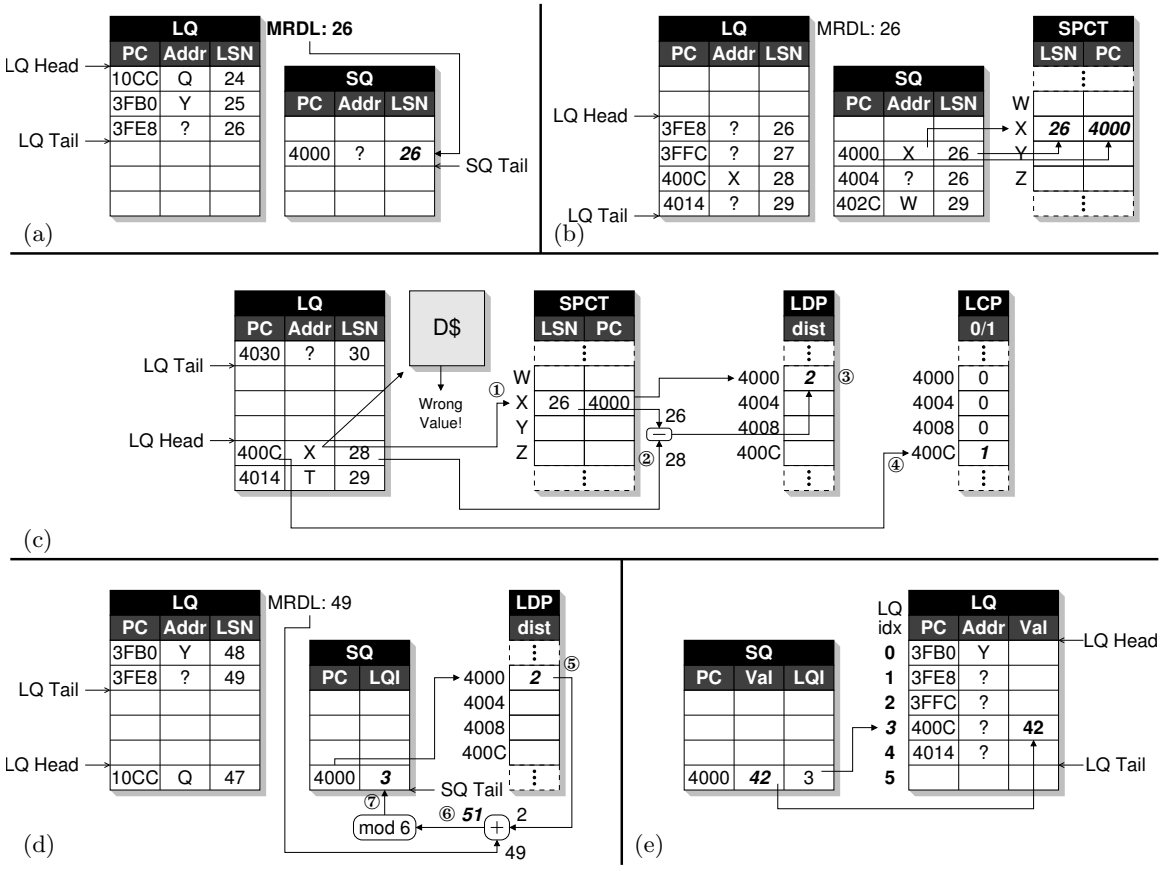


Figure 37: Fire-and-Forget example illustrating (a) a store tracking the LSN at dispatch, (b) the store updating the SPCT at commit, (c) a misforwarded load tracking its distance for future use by the store, (d) a later instance of the store computing the index for the predicted LQ entry, and (e) forwarding a value to the LQ entry.

5.3 Fire-and-Forget Store-to-Load Forwarding

We now describe our Fire-and-Forget (FnF) store forwarding design. Sha et al.'s Store Queue Index Prediction (SQIP) exploited the fact that a load that receives a forwarded value usually receives its value from the same store. In this work, we take a subtly different view of this observation: a store that forwards a value usually forwards it to the same load. While this may seem like two sides of the same coin, the slight change in perspective leads us to a different memory scheduling scheme that ultimately removes not only the CAMs or decoders of the store queue, but eliminates the store queue completely!

5.3.1 Load Queue Index Prediction

Store queue index prediction takes a *load-pull* approach to store-to-load data communication. When the load has computed its address, it checks its predicted source SQ entry and then possibly pulls a value from this store. We propose a *store-push* approach that can be considered as the dual of SQIP. For each store, we make a prediction for the LQ index that the store should forward its value to.

Similar to SQIP, Fire-and-Forget starts with SVW_{NLQ} and augments it with three auxiliary tables. The first is a modified SPCT that tracks a store's PC as well as its position relative to the LQ entries. The second is the *load distance predictor* (LDP) that tracks the relative distance from a store to the load it should forward to. The last table is the *load consumption predictor* (LCP) that determines whether a load should make use of a forwarded value. In the following text, we first explain how FnF works with a stripped down SQ, and then we explain how to remove the entire SQ.

We define a *load sequence number* that is analogous to SVW's SSN. The processor assigns a unique and monotonically increasing identifier to each dynamic load. When the processor dispatches a store instruction, the store records the LSN corresponding to the most recently dispatched load (MRDL), as shown in Figure 37(a).[3] Although we do not place stores in

[3]MRDL here can also be thought of as the most recently allocated store. Dispatch here refers to the pipeline stage of allocate and not issue. This distinction is important as when the store is dispatched its MRDL may not have issued nevertheless the store does not need to wait for this load to issue and go ahead and determine the load that it needs to forward to.

the load queue, this MRDL number indicates where in the load sequence the store would have hypothetically been inserted. When a store commits, it writes its MRDL and PC number into the SPCT, shown in Figure 37(b). The LSN in the SPCT effectively records the position of the hypothetical insertion point of the store in the load queue. Later, when a load re-executes and discovers that it received an incorrect value, it will use its effective address to check the SPCT to see if there was a recent store to this same address, shown in Figure 37(c:①). Assuming the load finds a valid SPCT entry, the processor subtracts the recorded MRDL from the load's current LSN to compute the distance in load queue entries from the hypothetical store insertion point to the consuming load (②). The processor then stores this distance in the LDP based on the *store*'s PC (③). The load also updates the PC-indexed LCP so that in the future it will know that it should receive a value from a store forwarding (④). Each entry of the LCP is a single bit, and we only update the LCP on a load re-execution-induced pipeline flush. The bit is set to zero if the load should have accessed the cache, and one if the load should have used a forwarded value. When the processor next encounters this same static store, it searches the LDP and finds a non-zero load distance, shown in Figure 37(d:⑤). The store computes a predicted LSN to forward to by taking the current MRDL and adding the distance from the LDP (⑥). Similar to SQIP, taking this predicted LSN modulo the load queue size provides the actual load queue index (LQI) of the target load (⑦).[4]

As described above, FnF forwarding could be used with a non-associative SQ. A store would wait in its respective SQ entry until it is ready to issue, and then it simply forwards its data to the predicted load queue entry, shown in Figure 37(e). Note that the data movement from stores to loads is the sole responsibility of the store instructions. *A load never searches, polls or otherwise accesses the store queue* to try to find data (i.e. no CAMs, no decoders, no way at all for the load to access the SQ).

When a load dispatches, it first checks the LCP to predict whether it should use a forwarded value. If the value is already present, then the load can use it right away,

[4]This example uses a LQ size of 6 entries, but a practical implementation would have a power-of-two-sized LQ, thus making the modulo a trivial computation.

otherwise the load stalls until a value arrives. If the LCP indicates that the load will not receive a value from an earlier store, then the load can access the data cache as soon as it has computed its effective address. Note that for a load to consume a forwarded store value, it must go through two levels of "filtering": a store must predict to forward to this load, and the load must independently predict to make use of the value. It is possible that some earlier store will attempt to forward a value to this LQ entry, but the load can always simply ignore it. If the sourcing store's load index prediction is correct, the store will eventually send its value to the load. However, if the sourcing store had a load index misprediction, or no earlier store even predicted to forward a value to this load, the load could potentially deadlock in a situation where it is waiting for a value that will never arrive. To guarantee forward progress, a load will issue to the data cache when there are no longer any earlier unexecuted stores. The regular SVW mechanism is still in place to guarantee that the load's "final answer" is correct; however, SVW does not protect against speculatively forwarded values. Our implementation of FnF forces all loads that used forwarded values to re-execute, but this does not have a substantial impact on performance because only some of the loads use forwarded values. FnF could be modified to have stores forward their SSNs as well to enable the forwarded-to loads to better use SVW's SSBF.

Similar to the baseline SVW_{NLQ} and SQIP, the three auxiliary tables/predictors used by Fire-and-Forget (SPCT, LDP, LCP) are all accessed off of the critical path of load-store execution. As a result, none of these structures affect the scalability of the core memory scheduling hardware. The SPCT needs one write port for each store that can commit each cycle. A load re-execution that reveals a misforwarded load requires the load to read the SPCT. Even though we may commit multiple loads per cycle, we only process the first detected misforwarding and therefore the SPCT only needs a single read port. Processing only a single misforwarding implies that the LDP only needs a single write port as well. The LDP needs one read port for each store dispatched per cycle, although one port can be combined with the write port because a write and read will not happen in the same cycle.[5] The LCP requires as many read ports as loads dispatched per cycle, and as

[5] A write to the LDP only occurs after a misforwarding, which implies that the processor has been flushed

many write ports as loads committed per cycle. However, this slightly higher port count fortunately corresponds to the smallest of our three tables. Compared to SQIP, our FnF table management is simpler because all updates occur at commit and therefore never need any sort of recovery. SQIP updates its SAT in the rename stage which requires additional SAT ports and storage for logging/checkpointing to support recovery after pipeline flushes.

5.3.2 Complete Store Queue Elimination

The previous discussion explained why FnF removes the need for any sort of SQ access circuitry for load instructions. At this point, we question what benefit the remaining logic in the store queue actually provides. Load re-execution guarantees that loads eventually receive the correct value, and FnF provides a means of forwarding values from stores to loads, and therefore the SQ only buffers the stores and their results in program order for eventual in-order writeback to the cache. There is another structure in the processor that already tracks all instructions in program order: the reorder buffer or ROB. Since the ROB maintains a list of *all* instructions in program order, it necessarily includes all stores in program order. We still need to buffer the results of the stores as well. In some microarchitectures (e.g., Pentium-Pro/Pentium-M [72, 27]), each ROB entry contains a physical register to store the result of that instruction. Normally, this physical register goes unused for store instructions because the store's value is kept in the corresponding store queue entry. In FnF, we propose to make use of this otherwise neglected resource to record the store's value.[6] After this change, *all* of the SQ's functions have been outsourced to other mechanisms, and so we can completely do away with the store queue!

To properly execute store instructions, some hardware resources are still required. Similar to a store's address-and-data (STA-STD) μop decomposition in modern x86 microarchitectures, we allocate two separate scheduler (RS) entries for each component of the store instruction, as shown in Figure 38(a).[7] In this example, we assume that an instruction's

and several cycles must elapse before any new stores have been fetched and are ready to read the LDP.

[6]For a microarchitecture that uses a separate physical register file such as the Intel Pentium 4 or the Alpha 21264, we could allocate a physical register to the store without updating any RAT entries.

[7]The Pentium-M's μop fusion could be used to pack both STA and STD into the same RS entry, but we do not explore this optimization in this work.

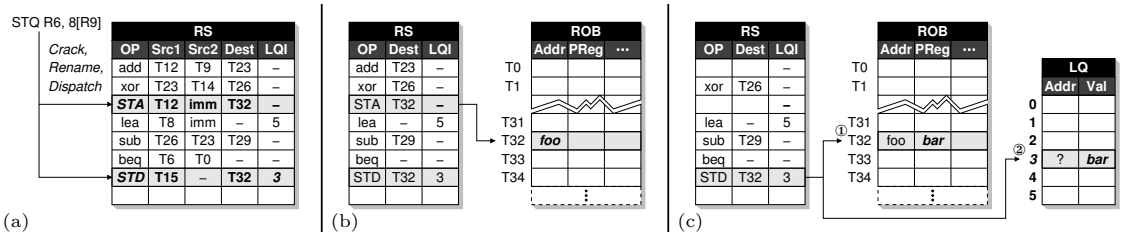


Figure 38: An example of Fire-and-Forget without a store queue showing (a) cracking and dispatch of a store, (b) store address tracking, and (c) store data tracking and speculative forwarding.

destination physical register is integrated into its ROB entry like in the Pentium-Pro [72]. When the STA μop executes, it computes its effective address and saves it in its ROB entry for its eventual cache update at commit (b). When the STD μop executes, it also writes its value to the ROB entry for eventual writeback (c:①). In addition, if the store has a predicted load to forward to, it uses its load queue index and performs a direct write (RAM) to the corresponding load queue entry (②). This speculative forwarding occurs in a completely blind fashion: the destination LQ entry may not have predicted to wait for a forwarded value. This is the one and only chance a store has to directly forward a value to a load and afterward makes no other effort for proper forwarding: hence the name *Fire-and-Forget*.

In the FnF mechanism without a store queue when a store-data issues it forwards a value to the predicted LQ index and then forgets about it. This value needs to be stored somewhere. In designs that assume that the LQ includes storage for forwarded values [59] FnF does not incur any additional overhead. However based on the feedback we received, atleast in some of the Intel processors and other proposed designs [8] the LQ does not hold the value since this forwarded value is still kept in the SQ which is searched every cycle by the loads. Since FnF does the forwarding based on when the store's data is ready and there is no store queue, this value may not be present in the processor when the load is ready to issue unless it is kept in the LQ entry. However for a 128-entry LQ, 8 bytes of data stored in each entry corresponds to an overhead of only 1KB which when added up is still far less that the area overhead of SQIP.

5.3.3 Speculative Memory Cloaking

An interesting attribute of FnF is that the act of speculatively forwarding based on a predicted LQ index can independently occur from any effective address computations. Neither the sourcing store nor the receiving load need to know their respective addresses. So in addition to eliminating the store queue, FnF provides a simplistic implementation of speculative memory cloaking [53]. Although previous work argued that speculative memory cloaking was "not worth the effort" [43], that study primarily argued its position from the perspective of needing to add sophisticated load-store memory dependence prediction and requirements for invasive changes to the register rename logic. That study also did not consider non-unit store-to-load forwarding latencies that are commonplace in most modern LSQ implementations. However, we believe that the speculative memory cloaking facility provided by FnF is worth the effort because it effectively comes for free once one has implemented the basic FnF scheme.

FnF's speculative memory cloaking enables some curious scenarios. A store can forward its value to a LQ entry that does not even contain a valid instruction because the previous load that occupied this LQ entry had already committed, but a newer load has not yet been dispatched. In this case, the store still performs its forwarding and deposits its value in the unoccupied LQ entry. Eventually, a load will be dispatched to this LQ entry; if this new load's LCP says that the load should use a forwarded value, then the load can "consume" the value that was left for it by the earlier store (which itself may have already committed). In this case, the load can execute immediately after dispatch, even before it has computed its effective address. Another variant of this scenario is when a store forwards to an LQ entry that contains a wrong path load. When the processor later detects the branch misprediction, it flushes the LQ entries containing wrong path instructions. However, there is no need to flush the forwarded value out of the LQ entry. When the processor eventually refills that LQ entry with a correct-path load, that load may use the left-over forwarded value. This implements a form of *control independent* speculative memory cloaking.

5.3.4 Corner Cases and Loose Ends

In this sub-section, we briefly clarify the behavior of Fire-and-Forget in several uncommon cases for completeness. In the vast majority of scenarios, a store forwards its value to zero or one LQ entry. In the infrequent case where a store should forward to more than one LQ entry, FnF will simply suffer from a misforwarding or non-forwarding.

The speculative forwarding creates the potential for a scenario where a store forwards past the logical end of the LQ (the LQ head) to a load that actually precedes the store in program order. To prevent sending values "back in time", the store also forwards its MRDL; if the load's LSN is less than or equal to the store's MRDL, then it should ignore the value. Similarly, if the store forwards fewer bytes than the receiving load reads (i.e. partial forwarding), then the load ignores the forwarded value.

It is possible to have more than one store forward a value to the same LQ entry. In our implementation, each forwarding overwrites the previous contents (subject to the LSN wrap-around check described above), and a load that consumes a forwarded value simply uses the value that was present at the time the load issues.

A conventional ROB entry may not contain a field for recording a store's address (since this was kept in the SQ), and so we can either add this field as shown in the example of Figure 38, or we can allocate two ROB entries for each store instruction: one for the STA and one for the STD. In our simulations, we make store instructions allocate two ROB entries for the STA and STD μops instead of adding an address field to every ROB entry. This does not increase the storage requirements of the ROB, but it can place more pressure on the ROB entries.

5.4 Evaluation

Similar to the work on Store Queue Index Prediction, our goal is to provide performance that is competitive to realistic fully-associative store queue designs while providing the implementation benefits of not needing a store queue at all.

5.4.1 Methodology

We use cycle-level simulation to evaluate the performance of the Fire-and-Forget forwarding technique. We use the MASE simulator [38] from the SimpleScalar toolset [3] for the Alpha instruction set architecture. We modified MASE to include pre-commit load re-execution, and we also implemented the SVW filter, the SQIP-based non-associative store queue, and our FnF mechanism. We simulated an aggressive out-of-order microarchitecture similarly configured to the baseline machine used in the SQIP study. Specifically, the baseline configuration has a 512-entry reorder buffer, 300-entry issue queue (RS), 128-entry load queue, 64-entry store queue, and a 20-stage pipeline depth.The processor is 8-wide throughout (fetch/dispatch/issue/commit), with a 4K-entry gshare branch predictor and 4K-entry/4-way BTB, 4 integer ALUs, 1 integer multiplier, 4 data cache ports (2 load, 2 store), 2 FP adders, 1 FP complex unit, 16KB/4-way/3-cycle L1 data and instruction caches, 512KB/8-way/10-cycle unified L2 cache, 250-cycle main memory latency, 64-entry/fully-associative instruction and data TLBs.

We simulated a variety of applications from SPEC2000 (all programs), MediaBench [39], MIBench [29], Graphics applications including 3D games and ray-tracing, and pointer-intensive benchmarks [4]. All SPEC applications use the reference inputs. For applications with multiple reference inputs, we weight the separate runs such that each individual benchmark weighs equally in the final comparisons.[8] To reduce simulation time, we used the SimPoint 2.0 toolset to choose representative samples of 100 million instructions [56].

Sizing SQ Structures: Our FnF processor and the SQIP approach both make use of SVW as the underlying load re-execution filter. We make use of slightly larger SVW structures than used in the SQIP study [67] because we use the reference input sets for SPEC which results in larger memory footprints for several benchmarks. In particular, we use a 4K-entry SSBF with 16-bit SSNs (8KB state total) with a similarly configured 4K-entry SPCT using 8-bit partial store PCs (4KB). The total non-SQ overhead for the baseline SVW_{NLQ} is

[8] For example, `eon` has three input sets, and so each would receive a weight of 0.33, whereas `mcf` has only a single reference input and we assign it a weight of 1.0. The encoding and decoding phases of the MediaBench applications are regarded as separate programs.

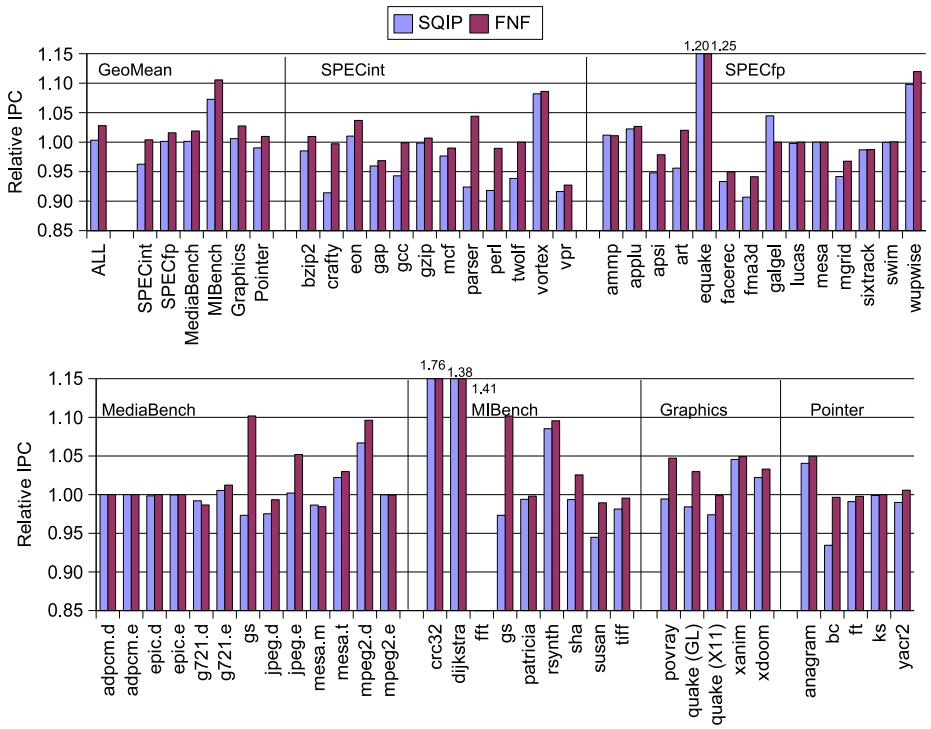


Figure 39: Relative IPC performance over a fully-associative SQ.

12KB. We used the SQIP structure sizes suggested by Sha et al. [67]: a 256-entry SAT ($\frac{1}{2}$KB), a 4K-entry FSP (10KB) and a 4K-entry DDP (12KB). The total non-SQ overhead of SQIP *over SVW* is $22\frac{1}{2}$KB, plus the additional state required for SAT logging.

Fire-and-Forget has three tables in addition to the SVW structures. We add a field to the 4K-entry SPCT to records the MRDL for each store. The MRDL is a LSN which we set to be the same size as the SSNs (16 bits), and therefore adds 4K-entries $\times$ 16 bits = 8KB to the SPCT. The LDP stores load distances, which need not be larger than the number of LQ entries. The baseline configuration uses a 128-entry LQ and we allot 4K-entries for the LDP, so $3\frac{1}{2}$KB of hardware is sufficient. The last structure is the LCP, which is a simple array of 1-bit entries. We use a 2K-entry LCP that only requires $\frac{1}{4}$KB. The total non-SQ overhead of FnF *over SVW* is $11\frac{3}{4}$KB, which is 48% less state than SQIP not even accounting for the elimination of the store queue or the SAT logging overhead. For designs that do not include storage in the LQ for values there is the additional overhead of 64-bit data associated with each LQ entry which adds 1KB to the original LQ hardware budget.

5.4.2 Performance Results

For performance comparisons, we use a baseline processor equipped with a non-associative load queue, with pre-commit load re-execution to validate load values, and the SVW re-execution filter as described in Section 5.3. This baseline is a processor that uses the SVW_{NLQ} scheme, which includes a fully-associative store queue. Figure 39 shows the relative IPC rates for both SQIP and Fire-and-Forget compared to SVW_{NLQ}, grouped by benchmark suite/category. The per-group geometric means show the average IPC rates for each group.

Our results verify the claims of Sha et al. that Store Queue Index Prediction provides comparable performance to the baseline without the need for a fully-associative store queue. Although the performance impact varies between applications, the overall performance is as good as (actually, 0.4% better than) the fully-associative SVW_{NLQ} implementation.[9] Figure 39 also shows that our Fire-and-Forget technique (second bar in the figure) provides a performance improvement of 2.9%. The per-benchmark variations correlate strongly to SQIP's variations, which makes sense because both approaches attempt to capitalize on the same phenomenon of the predictability of store-load forwarding relationships. Overall, our results show that SQIP meets or exceeds the performance of the fully-associative SQ on 31 out of our 55 applications, while the rate for FnF is 45 of 55.

For some applications, both SQIP and FnF perform better than the baseline SVW_{NLQ} configuration. In these situations, this is primarily due to a reduction in the number of ordering violations and the subsequent pipeline flushes (e.g., due to SQIP's delay distance predictor). In a few cases, FnF also performs better than SQIP. The performance benefit of FnF primarily comes from higher predictor accuracy and a reduction in the store queue capacity pressure since the number of stores in SVW_{NLQ} and SQIP is limited by the SQ size, but in FnF it is limited by the size of the ROB.[10] Speculative cloaking can provide

[9] The original work by Sha et al. reported a 3.3% IPC reduction, but that was in comparison to an ideal oracle-scheduled LQ/SQ design. Our SVW_{NLQ} is similar to their "associative-3" configuration, for which their results also show that SQIP provides about the same performance. We do not expect our results to match perfectly for a variety of reasons including differences in benchmark sets, compilers, sampling methodologies, etc.

[10] Or half the ROB if STA and STD are split between two entries as discussed in Section 5.3.

some performance benefit as well. There are a few applications where both SQIP and FnF perform worse than the baseline due to forwarding mispredictions. In the next section, we will examine the behavior of a few applications in greater detail to better understand how and why Fire-and-Forget works.

5.4.3 Scalability of FnF: Performance analysis over a medium configuration

The results presented in the earlier sub-section were obtained using a rather large configuration. For purposes of fair comparison of FnF with SQIP we used the configuration presented in the original paper that proposed SQIP [67]. We also simulated a less aggressive 128-ROB, 32-RS and 32/32-LQ/SQ configuration. In this sub-section, we show that FnF scales even when a more 'realistic' processor configuration is used. Figure 40 presents the relative performance of SQIP and FnF over an associative store queue equipped with the SVW filter. As the machine is scaled down, the trend for both SQIP and FnF stays the same across the various benchmark suites. We see that the difference in performance between an associative store queue with SVW and the other mechanisms goes down slightly especially for the MiBench suite. This is probably due to the fact that a smaller machine exposes fewer forwarding opportunities and thus all the mechanisms perform similarly. Additionally, the difference in the access times for 32-entry associative and indexed store queues is smaller than that between 64-entry associative and indexed store queues. A specific example of this is the `djikstra` benchmark which showed a significantly lower performance improvement in the smaller machine as opposed to the larger machine.

5.5 Specific Case Studies

5.5.1 A Good Example

The benchmark *gs* (ghostscript, which appears in both MediaBench and MIBench although compiled separately and run with different inputs) shows one of the biggest differences in performance between SWIP and FnF. While SQIP's performance is reasonably close to the SVW_{NLQ} baseline (<3% difference), FnF manages to achieve about a 10% speedup. While many benchmarks exhibit reasonably high forwarding rates [67], accurate prediction

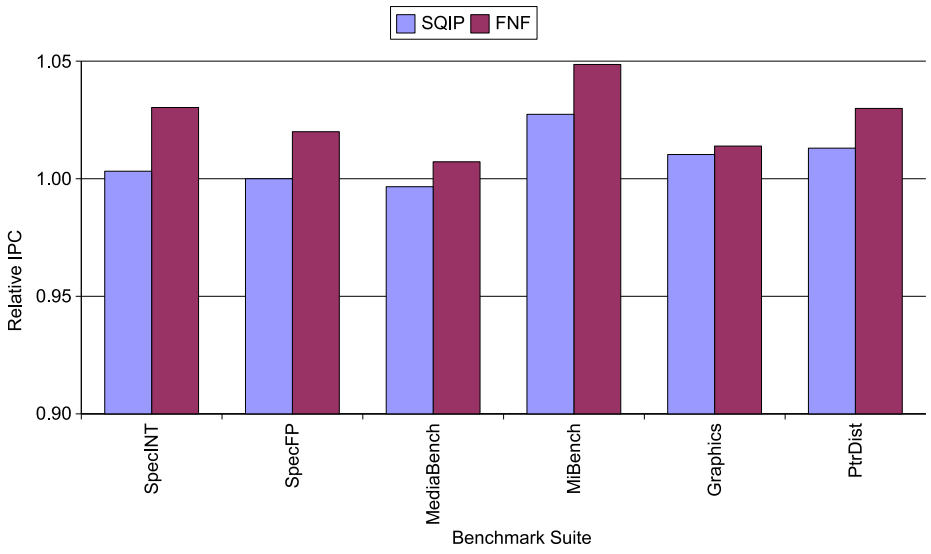


Figure 40: Relative performance of FnF and SQIP over the SVW mechanism for a medium sized machine

of these forwardings (whether using SQIP or FnF) typically does not improve performance (it just keeps the performance close to that of the associative SQ). For *gs*, FnF manages to achieve a 98.4% forwarding accuracy (i.e. of all loads that consumed a forwarded value, almost all received the correct value), which avoids the vast majority of pipeline flushes. We also tracked the frequency with which a load that was predicted to consume a forwarded value, received that value before it had even computed its effective address. Almost all (99.97%) of `gs`'s forwarded-value-consuming loads issued ahead of their respective address calculations. Applications like *gs* with store-load communication patterns that are amenable to speculative cloaking can derive a significant performance benefit. Across the entire range of applications, however, the opportunity for cloaking address calculations is limited and in general it plays a smaller role. Nevertheless, FnF provides this functionality "for free", and so it provides a good cost-benefit tradeoff even if it only helps a few applications.

5.5.2 A Neutral Example

The program *bc* from the Pointer applications is an interesting case to contrast against *gs*. We ran *bc* with three different input sets, but only two exhibited interesting levels of forwarding, and so the following discussion is in the context of only those two runs. The

bc program achieves high forwarding accuracy, but most of the receiving loads consume the value after their addresses have been computed. In this scenario, FnF does not provide any direct benefit over the other techniques that wait until addresses have been computed. We also observed that 6.9% of $bc's$ forwarded-value-consuming loads actually received a value from the wrong store (either a store with a different address, or not the most recent store to the correct address), but still ended up receiving the correct value. The combination of frequent value locality [88] (especially for the value zero) and the value-based approach of the load re-execution mechanisms makes this possible.

5.5.3 An Extreme Example

For the majority of applications, our FnF technique achieves performance that is within ±5% of the SVW_{NLQ}baseline. However, for a few benchmarks, the performance falls far outside this range. This is particularly true for the MIBench embedded benchmarks. While MIBench may not be representative applications for typical high-performance systems, we still find them useful to include as they exercise the processor in different ways to expose corner-case behaviors, and many of the programs are small enough to facilitate straightforward source-level or assembly-level analysis. For example, *crc*32 results in a very large performance speedup for both SQIP and FnF. We analyzed the simulation statistics and the *crc*32 source code and assembly and found that there is a repeating load-op-store sequence to the same address of the form `(*x)++`. This is embedded in a very tight loop with a highly predictable store-to-load distance. Due to the small size of the loop, the processor will typically have multiple loop iterations in-flight at once, resulting in multiple instances of stores to the same address. When the SVW_{NLQ} configuration issues the dependent loads, the associative scan of the SQ will often find a matching store, but this match is usually for a store in the wrong loop iteration (i.e. a load in iteration i should read from the previous store in iteration i-1, but if that store address had not yet been computed, then the load could have matched against the store from iteration i-2 or earlier). The large number of wrong-iteration forwardings results in a substantial number of pipeline flushes, whereas both SQIP and FnF correctly identify the relationships between specific loads and stores

and thereby avoid misforwardings.

5.5.4 A Misbehaving Example

The worst performing application is *fft*, also a small program from the MIBench suite, which experiences a 20% slowdown for both SQIP and FnF. The main loop of the program only contains about two dozen unique store instructions, but two of these stores happen to have PCs that end up aliasing in the SPCT and they execute in a strictly alternating fashion. Furthermore, one store always forwards to the immediately following load, whereas the other store is separated from its consuming load by a code hammock, such that the distance in either SQ or LQ entries between the two instructions varies depending on the dynamic control flow. We verified that this performance anomaly disappears when we increase the size of the SPCT. Most applications actually have more aliasing than *fft*, but in these cases there are enough unique forwarding behaviors that both the SQIP and FnF predictors can be negatively trained to not speculate so much. The problem with *fft* is the strict alternation between the two stores with different behaviors. Another solution for *fft's* performance deviation is to implement a 2-way set-associative SPCT, which could provide more robustness against pathological memory communication patterns.

5.6 Power, Area and Latency considerations

While the earlier sections showed the performance improvement and the implementation ease of FnF, in this section we concentrate on the power and energy benefits of FnF. To estimate the various metrics we used CACTI 4.1 which was modified to simulate the various structures [81]. For all calculations we use a 65 nm technology and 0.9V supply voltage. As a baseline we simulated a 64-entry fully associative SQ and a 128-entry non-associative LQ augmented with the SVW filter. For the SQIP design a 64-entry indexed store queue was simulated along with the other structures required for forwarding as described in Section 5.2.2.3. The energy savings provided by an indexed store queue is about 42% when compared to a traditional associative store queue (due to partial tags the partial address CAM of a fully associative store queue is only 12 bits and not the entire 40 bits of physical address). Our simulations indicate that the implementation of FnF provides an energy

savings of about 50% compared to the baseline case. This result is not surprising as there is no store queue at all in FnF and thus the power and energy expended is only due to the various structures described in Section 5.3 which is not much. This analysis of FnF includes the extra storage required in the load queue to hold the forwarded values. The power requirements of the FnF mechanism is 13% less than that of the SQIP mechanism. Note that since the store queue energy accounts for less than 2% of the entire processor energy [67] the energy impact of both the SQIP and the FnF designs was not further explored. Since the per access load latency using a scheduling mechanism like FnF is difficult to compute(most of the structures are accessed off the critical path and can be overlapped) it is not presented here.

5.7 Speculative Stores to Improve FnF Efficiency

Techniques like FnF and SQIP work well because of the infrequency in store-load forwarding. In fact, the Load Distance Predictor or LDP in the FnF technique only holds entries for those stores that forward values to loads; hence fewer forwarding occurrences can enable us to either improve the accuracy of prediction or reduce the size of the predictor. In this section, we explore an extension to the FnF technique that allows store instructions to write their values to the data cache earlier in the pipeline rather than at commit. This speculative write reduces the probability that a load would receive stale data from the cache when it issues, thus reducing the probability that it requires forwarding from an in-flight store. Using this approach we may be able to maintain forwarding information for fewer stores which can help improve the prediction accuracy and overall performance of the FnF technique or reduce the size of the predictor.

Our proposed technique involves determining exactly when the store is ready with its data and writing the data to cache at that time. This is in contrast to the conventional cache write mechanism, which happens only when the store instruction commits (actually a cache write happens only when the store reaches the head of the store buffer which may be several cycles after the store commits). Since the store write is executed several cycles early, our Speculative Stores (SS) technique has the potential to reduce the store-load forwardings

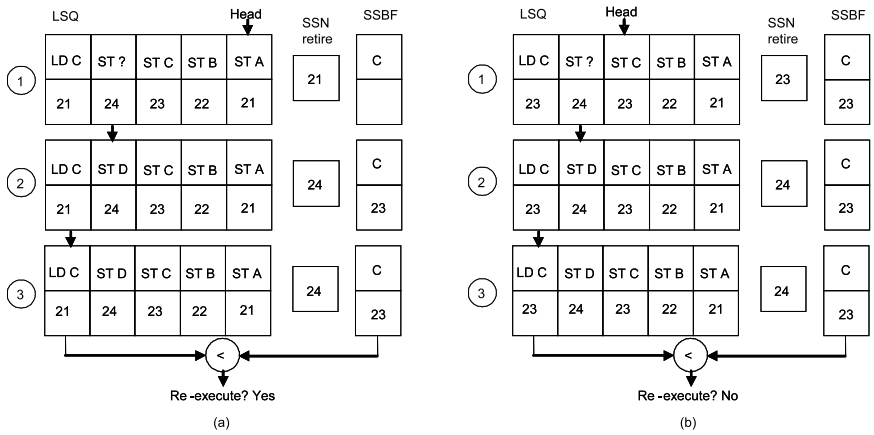


Figure 41: Working Example for Speculative Stores. (a) shows the working when simple SVW is applied and (b) shows the example when SVW is augmented with SS.

thereby improving the stability and performance of the predictor, the hardware required for the predictor and even reduce the number of load re-executions (the size of the store vulnerability window reduces since the oldest committed store is now a younger store). This cache write, however is speculative and conditions such as branch mispredictions need to be taken into account. To implement SS, we augment each cache line with a speculative bit. This bit is set on a cache write that is performed speculatively. When a store actually commits, a check is performed to see whether a younger store speculatively wrote to the same address. If no younger store speculatively wrote to the same address then only the speculative bit needs to be cleared to indicate that this data is not speculative anymore. This is done to ensure a speculative cache line is not evicted to L2 cache on a miss, where it can be observed by all the other processors in a multi-processor environment.

Our technique of SS is built on top of and makes use of the SVW technique. To understand the design, let us consider a working example for both SVW and SS. Figure 41(a) shows an example when just the SVW technique is applied and (b) shows an example when the SS technique is applied in conjunction with SVW. In snapshot 1, in the SVW scheme, the last store to retire when LD C is fetched is ST A with SSN 21. Thus, this becomes the SVW of the LD. Since the load executed in the presence of an older unknown store it must undergo the test for re-execution. When ST C with SSN 23 commits, in snapshot 2, it writes its SSN to the SSBF at address C. At the time of the re-execution test in snapshot

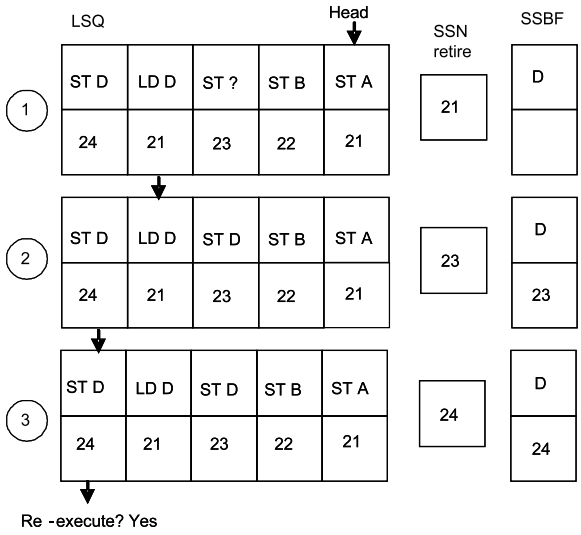


Figure 42: Store Re-execution in SS.

3, the test indicates that the load's SVW is older than the last store to have written to the SSBF and hence the load must re-execute. In the SS scheme, we write stores to the cache faster; hence at the time the load is fetched the last store to speculatively write to cache is ST C with SSN 23. Thus the load assigns 23 as its SVW and it reads the value from the cache. As indicated by the figure this is in fact the correct value for the load hence this load escapes re-execution.

Some issues regarding WAW violations can come up when we consider out-of-order cache updates. To maintain correctness we constrained our system to commit stores in order with respect to each other, but out of order with respect to other instructions. Even with this restriction there are scenarios where complications could arise. Consider one such scenario as shown in Figure 42. In this scenario LD D executes in the presence of older unknown store and hence must check for re-execution. ST D with SSN 24 cannot write to cache due to in order commit of stores with respect to each other. As the execution progresses, (snapshot 2) ST D with SSN 23 resolves its address and writes its data to cache. Later ST D with SSN 24 also writes to cache. In the second snapshot, when the load re-executes it would now get data from the wrong store (SSN 24). Hence, ST D with SSN 23 must first re-execute and write its value to cache before the load re-executes. Now LD 3 re-executes and gets the correct data. However, after this in snapshot 3, ST D with SSN 24 too has

to re-execute to make sure the data in the cache is updated. Hence in the SS scheme, we now have one load re-execution and two store re-executions as opposed to just one load re-execution in the SVW scheme.

As seen above, we may have a scenario where a younger store speculatively writes to the cache before the earlier store actually commits. Here we need to re-execute the older store at commit to ensure the cache has the correct value. To implement store re-execution, we need to associate a counter with every address. This counter can be stored in the SSBF itself. On a speculative write to an address, the counter is incremented. When the store actually commits the counter is checked to see if is > 1. If so, then it means that there is a younger store that has speculatively written to this address and then the store has to re-execute to ensure the cache has correct data. The counter is then decremented to indicate that a speculative store has been committed. If the counter is 1 after decrementing, then the last store is marked for re-execution as well to make sure the most recent data will be in the cache (since the re-execution test will not catch this store). These store re-executions may compete with actual cache accesses and load re-executions for bandwidth, and might reduce some of our performance benefit.

We now discuss the benefits of this FnF extension. The third bar in Figure 43 corresponds to the FnF technique augmented with Speculative Stores or SS. As shown in the figure, SS only slightly improves the performance of FnF (1% on average). There are some applications which do benefit due to early store commit. *eon*, *lucas*, *g*721.*decode* and *g*721.*encode* all see over 5% performance improvement over FnF. This performance benefit comes primarily due to reduced forwardings and improved accuracy. On the other hand, some of the applications experience quite a few store-reexecutions which causes their performance to slightly drop. *mcf* from the SPECint suite experiences 5% performance degradation when SS is applied due to the increased data cache port pressure. Figure 44 shows the performance impact of reducing the predictor sizes for both, the FnF technique and SS technique (FnF augmented with SS). The data points on the x-axis indicate decreasing predictor sizes in terms of number of entries. Our baseline size (which is also the first data point in this graph) has a 4K-entry LDP and a 2K-entry LCP. The slowdowns

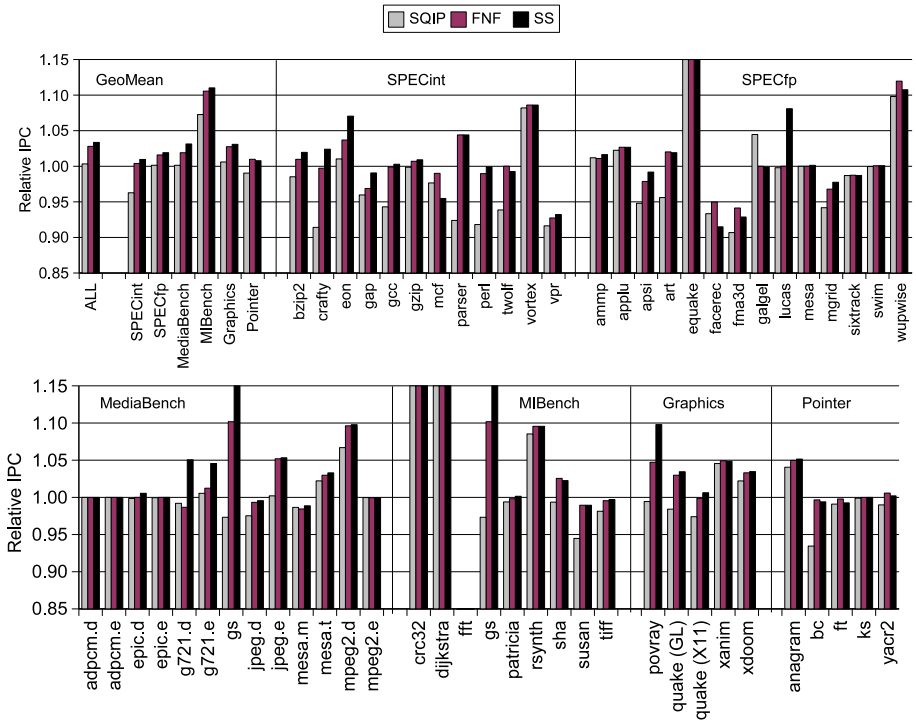


Figure 43: Impact of Speculative Stores on FnF. The third bar shows the performance of the SS technique.

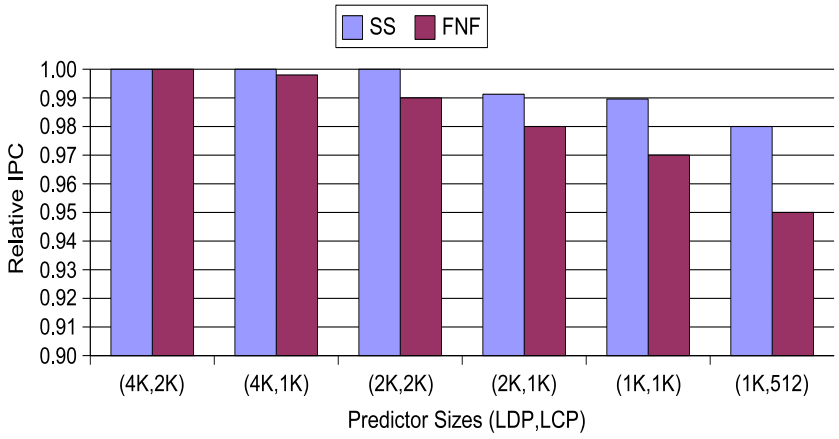


Figure 44: Performance of SS and FnF(base algorithm) relative to baseline sizes

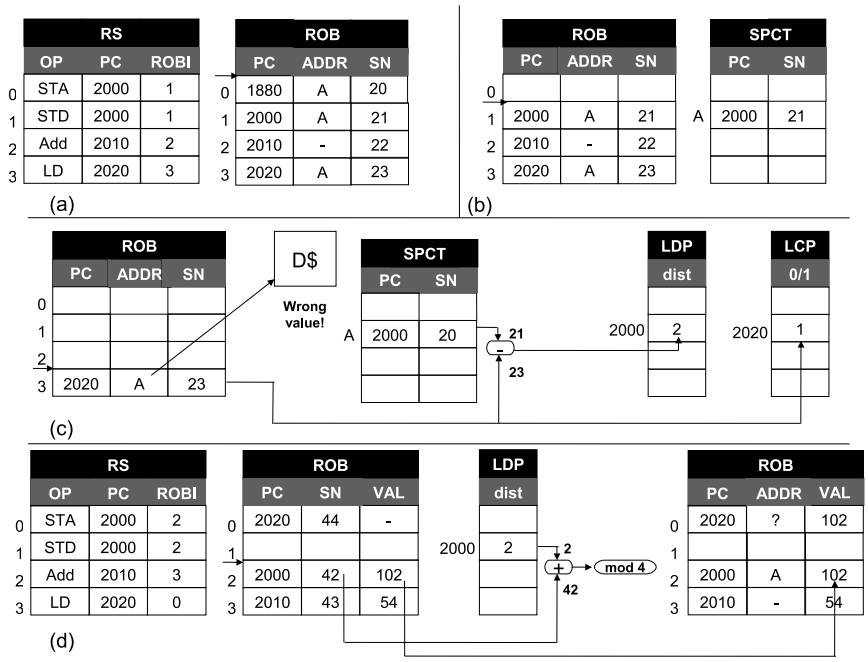


Figure 45: Implementing Fire-and-Forget with the ROB. Note that the SQ and the LQ have been eliminated.

denoted on the y-axis are relative to the specific technique (SS or FnF) at the baseline configuration. As we can see from the figure, SS performs well even at small predictor sizes whereas the performance of the base FnF falls off when the LDP and LCP are scaled down below 2K-entries. This shows that while at the baseline, SS may not help much, if smaller hardware budgets are required, then augmenting FnF with speculative stores can be considered. SS also reduced the load re-executions experienced by the processor by 2%.

In SS, however, there is an area cost of adding the speculative bits in the data cache as well as a power cost for the store re-executions that would eat into the area and power benefits provided by the base FnF design. Due to the small performance improvements and considerable area and power overheads, we believe that while being able to write stores speculatively to the cache is an interesting design which provides some benefits at small predictor sizes, it is probably not a worthwhile addition to Fire-and-Forget.

5.8 FnF_ROB: Load/Store Scheduling without a Store Queue or a Load Queue

In this section, we present an alternate design for Fire-and-Forget that eliminates the need for both the SQ and the LQ. Our implementation of FnF, so far, predicted forwarding distances in terms of LQ entries. Each entry in the LDP had a distance value that was equal to the distance from the corresponding store's most recently dispatched load or MRDL to its dependent load as described in Section 5.3. We assigned load sequence numbers (LSNs) for each dynamic load that were used to compute the distance. In this proposed optimization we will use the ROB, which already has an entry for each load and store, to compute forwarding distances.

The algorithm for using FnF with the ROB is similar to the LQ. We define an *instruction sequence number* or SN which is analogous to the LSN or SSN, except that this sequence number is a unique and monotonically increasing number assigned to every instruction. Since the LQ and SQ are not involved in data forwarding in this FnF optimization, we do not need to keep track of LSNs or SSNs. The distance between a store and its forwarding load, in this optimization, includes *all* instructions that fall between these two instructions (as opposed to the original FnF implementation which measured distances in terms of loads). When a store is dispatched, its instruction sequence number is recorded in the store's ROB entry. When the store commits, this value is recorded in the SPCT (indexed by store address) along with the store PC. When a load re-executes later and an ordering violation/missed forwarding is detected, the load will access the SPCT using its effective address, retrieve the corresponding store's sequence number, and subtract it from its own sequence number to compute the forwarding distance which is then stored in the Load Distance Predictor or LDP. The next time this static store is encountered, it computes the ROB entry that it needs to forward a value to by adding the predicted distance (from the LDP) to its sequence number.

We now explain this algorithm with the help of an example as shown in Figure 45. Part (a) shows four instructions that have been allocated. The oldest instruction in the ROB has a sequence number (SN) equal to 20. The next instruction is a store with PC

2000 (which is split into the STA and STD μops), followed by an ADD and then a load instruction with PC 2020. These instructions are allocated RS as well as ROB entries. Unlike earlier examples in this chapter that showed the STA and STD μops occupying separate entries in the ROB, in this figure we assume that that both these μops are fused into a single ROB entry. This assumption is used for simplicity and does not change the algorithm for computing forwarding distances. Since this is the first instance of the store 2000, no forwarding information is known at this point and hence this store will not forward any value when it executes. The ROBI field in each RS entry is the corresponding ROB index of the micro-op occupying that RS entry. In (b), the store reaches the head of the ROB (as indicated by the arrow), commits, and stores its PC (2000) and its SN (20) in the SPCT accessed by store address (A) so that, if required, future forwarding distance can be computed. In (c), the load (PC 2020) re-executes before commit and detects it received a wrong value due to a missed forwarding. This load accesses the SPCT with its address (which is the same as the store address) and retrieves the SN of the store (20) as well as its PC. The processor then subtracts the store's SN from the dependent load's SN to compute a distance of two. As explained earlier, this distance now also includes the ADD instruction that fell between the store and the load. This distance is stored in the Load Distance Predictor (LDP), accessed by the store PC, and the Load Consumption Predictor(LCP) is updated to reflect that in the future load 2020 should wait for a forwarded value. In (d), the learned distance is used to facilitate store-load forwarding. A later instance of store 2000 with SN 42 is allocated. This store accesses the LDP using its PC and finds that it needs to forward its value to a load which is two instructions away from it. The processor adds the store's SN to the predicted forwarding distance and mods it with the ROB size (4) to compute the ROB entry of the load that should receive the forwarded value. When the store data micro-op executed from the RS, the store's value (102) is copied into the ROB entry corresponding to load 2020. Although not shown in the figure, the load can now broadcast this value to its dependent instructions. As in the original FnF implementation, the load address is not yet computed at the time of forwarding, which provides an opportunity for speculative memory cloaking. Even more importantly, the entire forwarding process is

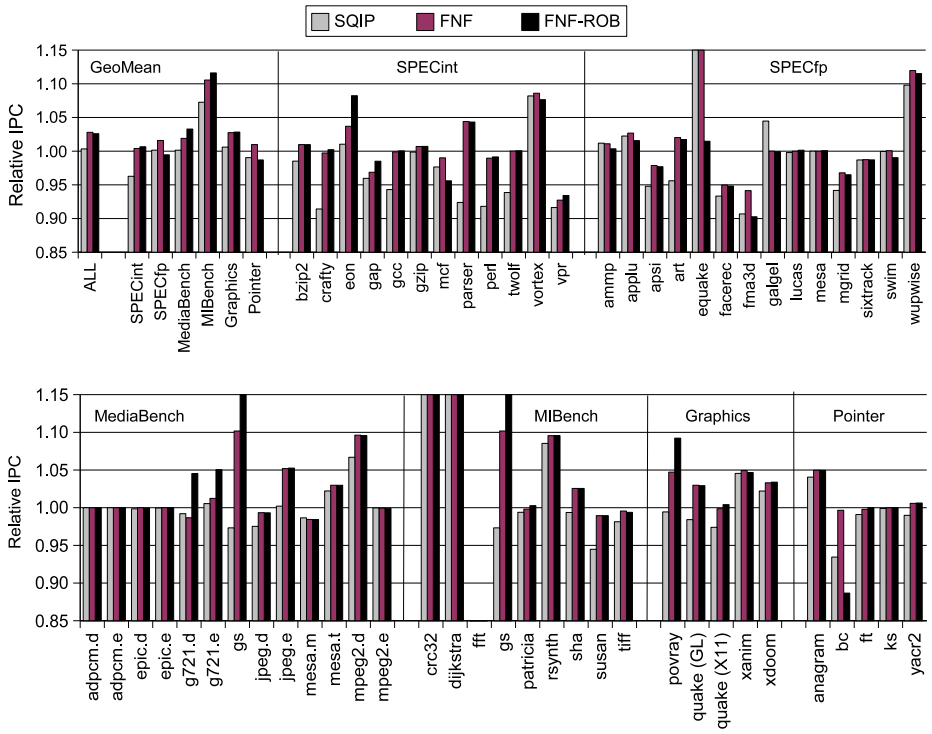


Figure 46: Implementing FnF without a LQ or a SQ. The third bar shows performance when FnF is implemented using only the ROB.

carried without a store queue or a load queue.

While this approach allows us to completely do away with both the LQ and the SQ, since we use the ROB, there are more instructions that lie between a forwarding store-load pair. This could lend some inaccuracy to the prediction which could affect the performance. Additionally branches that fall between a store and its forwarded load (not shown in the example here) may also impact the stability of the predictor. To evaluate the performance of the FnF-ROB technique we use the same baseline configuration as detailed in Section 5.4.1. All of the additional structures required for the SVW technique as well as the SQIP and FnF prediction algorithms use the same parameters as described in Section 5.4.1. The FnF-ROB design does not use a LQ or a SQ, hence the number of loads and stores that can be allocated is limited by the ROB size. We compare this technique to the original FnF predictor as well as the SQIP predictor. The third bar in Figure 46 corresponds to the relative performance of the FnF technique implemented using a ROB that completely eliminates both the LQ and the SQ. On average, we see that the performance is nearly the same as the base FnF

design. This is very encouraging, since as explained earlier, there could be some prediction inaccuracy due to the larger number of instructions between a store and load as well as impact of control flow. Most of the applications, however, still have high enough prediction hit-rates that offset these problems. Some applications even perform better when using only the ROB to perform the functions of the LQ and the SQ. Since the ROB is larger than the LQ, more loads can be fetched into the instruction window, which can expose more ILP. Many of the MediaBench and MiBench applications see significant performance improvements due to this phenomenon. Additionally the reduction in area and power due to the elimination of the LQ makes this FnF optimization a very attractive design.

5.9 NoSQ

Sha et al. have independently proposed another store-load communication design, NoSQ, that did not require a store queue [68]. While the goals of both techniques are similar, that of eliminating CAM-based associative logic with scalable designs that provided the same functionality, the intuitions used and designs proposed are quite different. Fire-and-Forget used the predictability in forwarding distances as well as the capability to reuse other structures such as the ROB to implement store-load forwarding without a store queue or a load queue. NoSQ, on the other hand, uses the principles of speculative memory bypassing, a previously proposed technique, that dynamically converts store-load forwarding into register communication. Data communication through memory takes place through a store-load pair. A producer instruction writes data to a register (PROD); this data is written to the memory by the store instruction which is then read by a load instruction and then forwarded to the consuming instruction that uses this data (CONS). SMB extends the conventional register renaming process to convert this PROD-store-load-CONS chain into a PROD-CONS chain [52, 53].

The loads that are eliminated from the chain are thus called "bypassed loads". NoSQ proposed, using SMB to predict whether a load could be bypassed as well as the bypassing store. Once such bypassing loads are speculatively identified, they skip the out-of-order core completely and are handled in the back-end. Non-bypassing loads obtain their value from

the data cache. Furthermore, since data communication now happens directly from the producer to the consumer, stores do not need to be executed in the out-of-order core either. NoSQ extends the commit pipeline to calculate the store and bypassed loads' addresses, commit store instructions to cache as well as verify correct data communication by using SVW-based filtered load re-execution. Thus while Fire-and-Forget still maintains store-load forwarding and uses the observation that forwarding is a rare occurrence, NoSQ removes all store-load forwarding in the out-of-order core by predicting which memory communications can be converted into register communications. Since stores are executed in the commit stage, NoSQ is able to eliminate the SQ by holding the required entries in the ROB. Loads do not receive any values from stores either in NoSQ; hence NoSQ does not really require even a LQ.

The basic difference between FnF and NoSQ is the guiding principle in both these data communication approaches. As mentioned earlier, NoSQ makes some non-trivial modifications in the register renamer to link the PROD to the CONS. Specifically the register alias table or RAT is extended in a Store-Sets fashion to allow the output physical register of the PROD is linked to the input physical register of the CONS. FnF, as described in this chapter, only uses the ROB to hold the store and load entries and that too does not modify the structure of the ROB. Also, due to the complete elimination of value forwarding from a store-load in NoSQ, the prediction mechanism to identify bypassing loads and stores needs to be very accurate. Their work shows that the predictor alone requires 10KB of hardware. In comparison the predictor tables in FnF occupy less than 4KB of hardware. NoSQ, however, completely eliminates out-of-order execution of stores and is able to reduce the number of loads that access the cache (most of the bypassed loads never access the cache). Hence, while both techniques achieve the objective of store-load communication without using a load queue or a store queue, if one does not want to incur the additional complexity of RAT modifications and memory bypassing required by NoSQ, FnF is likely to be the more feasible approach.

5.10 Conclusions

In this chapter, we have exploited the common-case property of predictability in data forwarding patterns to improve the scalability of the processor thereby improving its efficiency. Our optimizations to our base Fire-and-Forget (FnF) design have different impacts on the performance and efficiency of the algorithm. Speculative Stores as a concept is very attractive and has the potential to significantly improve the performance. Unfortunately, effects of store re-executions and the added hardware for maintaining the counters reduce its actual performance benefit. FnF using a ROB on the other hand, surprisingly performs very well. Not only does this design eliminate both the SQ and the LQ, it also improves the performance of some applications by removing the LQ and SQ size constraint and allowing more ILP to be exposed. This optimization simplifies the hardware and is easily scalable to larger microarchitectures as well (since only the ROB needs to be scaled up).

Fire-and-Forget provides several simplifications to the processor's load-store scheduling hardware. We have already pointed out the elimination of the LQ and SQ, but there are other implementation benefits of FnF. All of our predictor/tracking structures are accessed off of the main load and store critical paths and therefore are not likely to impact processor timing/clock frequency. Furthermore, FnF only updates these tables in program order at commit, hence no recovery or checkpointing mechanism for the predictor tables is required. The area and energy savings provided by the FnF technique of store-load forwarding (due to the elimination of the store queue) in conjunction with the performance improvement and scalability provided by the predictor help to improve the overall processor efficiency.

CHAPTER VI

LCP: LOAD CRITICALITY PREDICTION FOR EFFICIENT LOAD INSTRUCTION PROCESSING

6.1 Introduction

In this chapter we exploit the common-case behavior of predictability in load instruction criticality to improve the performance while reducing the power consumption of the processor, thereby improving its efficiency [77]. Conventional load optimization techniques are applied to all loads in a uniform manner. Optimizing every load, however, may not always provide performance benefits. Consider an addition instruction with two operands, both of which come from loads. The left operand always hits in the cache, while the right almost always misses. If we can accurately predict the value of the left operand, for example using a value prediction optimization technique, it will have no impact on performance because the dependent addition will still have to wait many more cycles for the right operand to return from memory. To improve upon this situation, researchers have proposed techniques to dynamically predict which loads are most likely to be critical, and then only attempt to perform optimizations for these loads [14, 23, 84]. In our work, we design a simple yet effective load criticality predictor and then propose and evaluate several optimizations that use this criticality information. Our approach to load criticality prediction uses a heuristic that is easy to implement in hardware making the predictor cost fairly low. We present results that show that our optimizations that use LCP or load criticality prediction significantly improve the performance of the processor while decreasing the power and area costs for several hardware structures.

While prior research has shown that the criticality of instructions, and loads in particular, can be accurately predicted, the hardware applications of criticality prediction so far have been limited to specialized or niche microarchitectures. The first primary application area was for dynamic value prediction [14, 23, 84], which after years of intense research

interest does not appear to have any traction with respect to adoption in a commercial processor.[1] The other primary application area is for clustered microarchitectures with cross-cluster bypass penalties [60, 23, 84] or asymmetric power or performance characteristics [64, 22, 44, 7], neither of which will be used in any known future *main-stream* commercial processors.[2] Other proposed applications include the Non-Critical Load Buffer [24] and the Penalty Buffer [6], both of which augment the DL1 cache with an auxiliary buffer that stores data accessed by non-critical loads. Although the proposed buffers are relatively small, adding any new structures to the DL1 processing path is very complex as memory accesses must get routed to multiple structures, results require extra multiplexing, and it also represents one more structure that must be snooped for maintaining cache coherence.

In our research, we use load criticality information in six simple criticality-based applications that can be directly applied to modern, conventional x86-based microprocessor organizations. Our applications are divided into three categories based on where in the load's execution pipeline they are applied. The first class involves the data cache port of a processor where we use criticality predictions to make a single-load-port data cache perform almost as well as an ideal dual-load-port version. The second class deals with the memory disambiguation and store-to-load forwarding. The third class includes applications that modify the placement of data in the Level 1 data cache (DL1) to increase the hit-rates of critical loads. While the load-port optimization for the DL1 has the greatest performance benefit, once the overhead for the criticality predictor has been paid for, implementing any (or all) of the remaining optimizations are effectively "free." The additional techniques only involve minimal microarchitectural changes beyond those already used for the load-port optimization. By simultaneously combining several of our criticality-based optimizations, not only is the performance benefit higher (nearly 16% over a conventional processor), but the cost of the hardware predictor also gets amortized across all of these multiple optimizations.

[1] Even one of the original researchers in value prediction concedes that the technology will not likely see commercial adoption [42], although the work has catalyzed an intense research thrust in value-aware optimizations and microarchitectures.

[2] The closest exception is the Alpha 21264 which has two integer execution clusters with a +1 cycle bypass penalty between clusters; however, there has been no real further development of this microarchitecture ever since the Alpha line was terminated.

The rest of the chapter is organized as follows. Section 6.2 reviews the concept of load criticality and provides a brief overview of the most relevant prior research. Section 6.3 describes the design and operation of our proposed load criticality predictor and Section 6.4 describes our simulation infrastructure. We then propose and evaluate several applications in Sections 6.5-6.7, and we consider the combination of these techniques in Section 6.8. We also compare our load criticality predictor with another previously proposed criticality predictor in this section. Section 6.9 summarizes how our LCP technique improves processor efficiency and presents some conclusions.

6.2 Background

In this section, we first briefly review the criticality of instructions and explain the conditions that make some loads more critical than others. We then briefly describe some of the most relevant prior studies and highlight some important observations.

6.2.1 Instruction (and Load) Criticality

An instruction is *critical* if increasing its latency directly causes the overall execution time of the program to increase as well.

Critical loads are simply load instructions that are critical. The reason we focus on loads is that they often have long latencies due to cache misses in the L1 or L2, and therefore are far more likely to lie on the critical path of execution. The criticality of loads can manifest in many ways. First, there is the simple case where the load is on the critical path in the traditional dataflow sense, as described earlier. Second, delaying loads that have a mispredicted branch as a child (or a grand-child, etc.) directly delays the detection and subsequent recovery from the branch misprediction. Third, when a load is the oldest instruction in the processor and suffers a long latency cache miss, the reorder buffer may become full and new instructions (regardless of whether they are data dependent on the load or not) will not be able to allocate resources and execute. On the other hand, not all loads are critical. A load that hits in the L1 cache may still have a latency of several cycles, but there may exist other paths through the data-flow graph that render this load non-critical; that is, the processor may be able to extract enough instruction-level parallelism to hide or

overlap the latency of the load.

The observation that some instructions are critical while others are not has been known for quite some time [74, 6]. More recently, work has gone into carefully quantifying the properties of the criticality [73, 85] and slack of instructions [22], as well as the dynamic predictability of criticality [23, 84]. In the next sub-section, we focus on ways to predict instruction criticality.

6.2.2 Predicting Instruction Criticality

6.2.2.1 Dataflow-Based Prediction

Fields et al. proposed a token-passing algorithm that develops dependence chains and learns the critical path of the program [23]. The predictor builds dependence graphs based on last-arriving chains of instructions assuming that the critical path is likely to reside along the last-arriving edges. The profiling technique described by Fields et al. determines the critical path by starting at the last instruction and traversing the graph backwards until it reaches the first instruction. The corresponding hardware implementation uses an array that tracks certain tokens that are distributed randomly at different instruction points. A set of pre-decided rules builds a chain of last-arriving edges from the root instructions (where the tokens are planted) along which tokens are propagated. All tokens that are propagated are checked for "liveness" after a certain number of instructions. If a token is alive when checked, the instruction at which the token was inserted is on the path of a last arriving edge and is deemed to be critical. While very accurate due to its explicit tracking through the program's dataflow graph, this approach is difficult to implement in hardware due to the management of the token-based array, token free list, and the simultaneous tracking of multiple tokens.

6.2.2.2 Criticality Prediction from Implicit Criteria

Several research proposals for criticality predictors make use of information other than explicit dataflow to *infer* criticality in a more heuristic-based manner. Fisk and Bahar proposed two indicators of load criticality [24]. First, loads that cause the processor utilization to fall below a certain threshold are considered to be critical. The second heuristic considers

the number of instructions that are added to the load's "dependency chain" over the course of a miss. Determining whether an instruction belongs to the forward slice of the load may require traversing multiple levels of the dependency graph which is a rather complex task to perform in hardware. To measure the number of dependents added during the course of a miss, the predictor needs to first track the number of dependencies at the time when a cache miss is detected, and again when the miss returns, to check for additional dependencies. Their proposed implementation adds counters to the load queue and MSHRs, plus some extra state to track which counter should be updated. Particularly troublesome for implementation is the need to communicate or broadcast consumer information from the allocation stage to the MSHR entries behind the DL1 cache. Their predictor also tracks whether any branches are dependent on the load (which could require tracking a large number of dependencies in each load's forward slice), in which case the load is automatically classified as critical. One interesting aspect of these predictors is that a load miss acts as the trigger to start tracking the criticality of an instruction, thereby reducing the number of instructions that need to be tracked.

Tune et al. also proposed several criticality predictors based on simple heuristics [84]. The heuristics consider the age of the instructions, the number of dependents, and the number of dependents made ready. While the heuristics are easy to describe, they are not all easily implementable in hardware. For example, the "QOldDep" criteria marks each instruction in the scheduling queue (RS) that is dependent on the oldest instruction that is in the queue as critical. This requires first knowing the oldest instruction (which may change every cycle), and then somehow broadcasting the information to all other instructions so that they can be marked as critical if they are dependent. Another heuristic, "QCons" checks each issuing instruction and marks the one whose result is used by the most instructions in the queue that cycle as critical. Implementing this in hardware would be expensive as it requires some way to immediately detect the number of physical register tag matches and then compare which instruction caused the most.

Srinivasan et al. proposed a more specific heuristic-based predictor that considered the instruction type of the dependents [73]. Their scheme proposes three conditions to classify a

load as critical: 1) any of its dependents is a mispredicted branch, 2) any of its dependents is a load that misses in the cache, 3) the number of independent instructions issued (within a certain number of cycles) following this load is below a pre-determined criticality threshold. The predictor table used in this proposal is comprised of dependency vectors, one per ROB-entry which tracks the number and type of instructions waiting for a specific load. This dependence vector increases with both the ROB size and the LQ size and can be a considerable area overhead.

6.3 Our Load-Criticality Predictor

Many of the previously proposed criticality predictors, while effective, are fairly complex to implement directly in hardware without severely impacting one or more major functional unit blocks (e.g., the out-of-order scheduling logic). In this section, we describe our simple yet effective criticality predictor used in this study.

6.3.1 Basic Operational Description

As a base principle for our load criticality prediction algorithm, we use the observation made by Tune et al. [84] and Fisk and Bahar [24] that a load instruction with many dependents is likely to be on the critical path. A simple statistical explanation for this is that an instruction with many consumers has that many more chances for one of them to be on the critical path, thereby making the original parent more likely to be critical as well. For our algorithm, the prediction hardware includes two main components: a Consumer Collection Logic (CCL) and a Critical Load Prediction Table (CLPT).

6.3.1.1 Consumer Collection Logic

The CCL is responsible for collecting the number of direct consumers (i.e., the load's children, but *not* its grand-children or other deeper descendants) that each load has while the load is in the instruction window. The CCL inspects every allocated instruction: if the instruction is a load, the CCL sets a bit associated with the logical register that this load writes to, indicating that the instruction which wrote to this register is a load. This bit is concatenated to the physical register mapping stored in the register alias table (RAT).

Additionally, the CCL resets a counter in the ROB entry corresponding to the load. If the instruction is not a load, then the CCL reads the input operand mappings from the RAT and determines if the parent is a load. Note that intra-group dependencies are also automatically handled by the existing intra-group dependency check and bypass logic, and we simply piggy-back on the regular RAT reads and writes so we do not introduce the need for any additional RAT ports. If the source is a load, we increment the counter associated with load's ROB entry. For implementation purposes, the counters in the ROB may be stored in a separate table so as to not impact the critical timing paths through the ROB.

6.3.1.2 Critical Load Prediction Table

When a load instruction commits, the consumer count that is associated with its ROB entry is written to the CLPT, which is a simple, untagged, PC-indexed table of k-bit counters. For each load that we allocate, we use the load's PC to access the CLPT to determine the number of consumers it had the last time we saw the same load. Based on the consumer count associated with its entry, a load is predicted to be critical or not.

For a single application of load criticality, each entry of the table would only need to have a bit to store whether the load is critical or not, which in turn would be determined by whether the number of consumers exceeded some pre-specified threshold. In this work, we examine multiple applications of load criticality where each technique may have different criticality criteria (i.e., different thresholds). By storing the consumer count directly in the CLPT, we can reuse the same prediction structure across all of our optimizations, even if each optimization requires a different criticality threshold.

6.3.2 Example

The following example demonstrates how the CCL collects consumer information, and how the CLPT gets updated. Consider the four instructions listed in Figure 47. Even though the CCL is responsible for controlling all of the data collection, we do not explicitly show it in the figures for simplicity. For each load, its ROB entry has a counter that tracks the number of immediate consumers. Every RAT entry has a load bit that indicates if the instruction

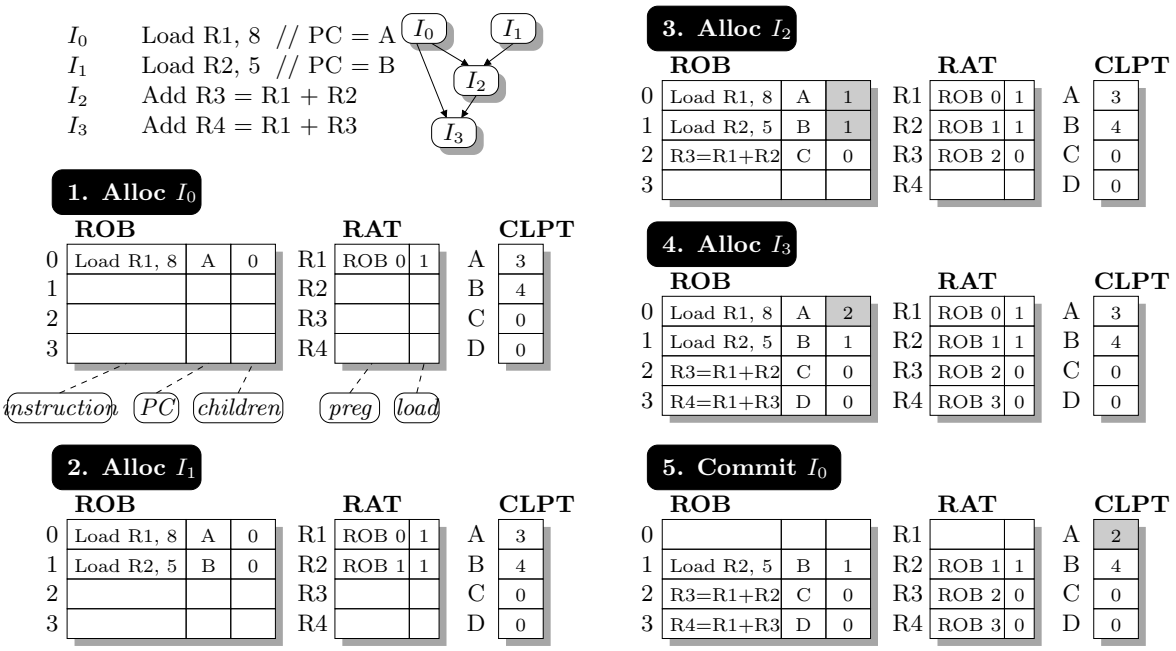


Figure 47: Example showing the collection of consumer counts and update to the prediction table.

that writes to the corresponding logical register is a load. Each snapshot corresponds to a specific operation on a single instruction. The example starts with the CLPT with some counter values that would have been collected the last time these loads were executed. All updates in a given step are highlighted with bold text.

1. Alloc I_0 In this step, the first load, I_0 is allocated. Since it is the first instruction, it is assigned ROB index 0. This ROB ID also serves as the renamed physical register (*preg*). The load bit is set in the RAT indicating that the instruction that wrote to register R1 is a load. The consumer counter in the ROB entry is set to 0. In parallel with this, the load PC A is used to look up the CLPT. The entry in the CLPT has a value of 3 indicating that the last time this load was seen, it had three consumers. Depending on the criticality threshold, this instruction may be marked as critical.

2. Alloc I_1 In this step, the second load, I_1 is allocated. It is assigned ROB ID 1, and the consumer counter in the ROB entry is set to 0. In parallel with this, the load PC B is used to look up the CLPT, which returns a value of 4 indicating that the last time this load was seen, it had four consumers.

3. Alloc I_2 In this step, the first add is allocated. First, the RAT entries that correspond to the source operands of the add are checked to see if either source is a load (note that this step would have occurred for I_0 and I_1 as well, but in our example neither of these two loads had register input operands). For this add, the RAT entry for the left operand indicates that its parent is a load. Using the ROB ID stored in the RAT, we now increment the corresponding child counter in ROB entry 0 (shown shaded in the ROB). Similarly, the right parent is also a load, and so we also increment the respective counter in ROB entry 1 (also shaded).

4. Alloc I_3 In this step, the second add is allocated. Checking the RAT, we see that the left source is a load, and so the counter in ROB 0 is incremented. The right operand is present in the ROB, but since the load-bit in the RAT indicated that the right parent is not a load, no further actions are taken. This add instruction is inserted into the ROB at entry 3.

5. Commit I_0 In this step, load I_0 commits. We read its counter value from the ROB and copy this to the corresponding CLPT entry to reflect its new consumer count which is 2. This new consumer count value will be used to predict the load criticality the next time Load A (or any other load that aliases to the same predictor entry) is encountered.

6.3.3 Issue-Rate Filtering of the Predictor

In the preceding sub-sections, we described and demonstrated the basic operation of our load criticality predictor. While the number of in-flight consumers of a load is a strong indicator of its importance, the load's criticality also depends on the dynamic processor conditions while the load is in the instruction window. Consider a load with five direct consumers in the window. Such a load will be considered critical if the criticality threshold is four. If there are enough independent instructions in the window that can execute in parallel with this load however, then the latency of the load may not significantly impact the performance of the processor. We restrict our algorithm to only collect consumer counts when the processor is in a period of low processor utilization or a "critical period." The idea is that when the issue rate of the out-of-order processor is low, then the probability

of finding one or more long-latency, critical loads is higher. Similarly, when the issue rate is high, most loads will unlikely be critical; even those that are critical will likely have relatively low *tautness* (the number of cycles that can potentially be saved before another path becomes critical [85]).

Restricting the predictor's training and usage based on issue rate has two advantages. First, we do not need to monitor loads during periods of sufficient performance. This can reduce the storage requirements of the CLPT because collecting data for fewer loads reduces the aliasing in the CLPT. Consider a program with two alternating function calls f(A) and f(B). Due to data-dependent control flow within the function, f(A) may have six loads that always hit in the cache while f(B) only has two loads that usually miss. Without filtering, these eight loads would need at least eight CLPT entries to store all of their criticality information. Since f(A)'s execution maintains a high instruction throughput due to the cache hits, criticality-based optimization of the loads in f(A) are unlikely to provide much benefit, and so storing the corresponding criticality predictions in the CLPT wastes space. If the CLPT updates only occur during periods of low performance, then updates from f(A) will be entirely filtered out, and only the two updates from f(B) will be stored in the CLPT. The result is that the CLPT only needs two entries to store this information.

Another secondary advantage of filtering CCL/CLPT activity is that it may not be necessary to explicitly account for branch or path history when making criticality predictions. The two loads in f(B) may have the same PCs as two of the loads in f(A). If the predictors are always updated, then it may be necessary to include, for example, branch history to distinguish between the different instances of the loads. At least in this example, our filtering mechanism would simply eliminate the loads from f(A), eliminating the path-ambiguity from the predictor.

To determine the critical period when load instructions should be tracked for criticality, we make use of the *issue rate* or the functional unit utilization of the processor. Every cycle the number of instructions issued is monitored and every s cycles a global or average issue rate for the processor is noted (an average can be easily computed if s is chosen to be a power of two). If the processor utilization falls below a pre-specified target issue rate, this

implies that a significant portion of the processor resources are sitting idle waiting for one or more stalls to resolve. When the global issue rate falls below this target issue rate, we enable the tracking of load consumers and subsequent updates to the CLPT.

We also include a 2-bit confidence counter for each entry in the CLPT. This counter is incremented if the difference between the previous and current number of consumers is less than Δ=2, and decremented otherwise. In other words the counter is incremented if the number of children is reasonably stable, and decremented if the number varies too much. If this confidence counter is low, then we assume the load is critical regardless of the number of consumers stored in the CLPT entry. To maintain implementation simplicity, we do not repair the CLPT (or update consumer counts) on pipeline flushes, as the design overhead to repair consumer counts due to wrong-path consumers is non-trivial. This could (slightly) impact the accuracy of the criticality prediction, but our confidence counters can protect against these cases.

We use the following parameters for our load criticality predictor. Our largest criticality threshold used is twelve, and therefore the child counters associated with each ROB entry each only use four bits. The exact threshold values were empirically chosen; for different microarchitectures or cache hierarchies, different thresholds may be used. The CLPT has 1024 entries, each of which consists of a four-bit field to store the observed number of children, and then also the two-bit confidence counter. We also include another one-bit field that will be described in Section 6.5. The total amount of state required for the CLPT is 1024 entries $\times$ 7 bits per entry = 896 bytes. Even when including the extra bit per RAT entry and the per-ROB entry 4-bit consumer count, the total state is still under 1KB. Our simulated processor has a peak execution width of six μops per cycle, and we only enable the predictor when the actual issue rate is less than four.

While previous studies have proposed predictors that are similar in spirit, the specifics of how to build hardware to collect the required information had been omitted; in many cases (e.g., the QOldDep policy that marks children of the oldest instruction in the RS as critical [84]) are perhaps easy to add to a simulator, but non-trivial to build a practical

circuit for. We have provided a detailed explanation of the microarchitecture-level implementation of our load criticality predictor. The modifications are few (one extra bit per RAT entry without requiring any extra ports, the ROB counters which can be implemented as a separate table to avoid impacting ROB area and timing, the simple PC-indexed CLPT which can be easily accessed in the processor front-end off any critical timing paths, and a few bits in the RS or LDQ entries to record the actual criticality predictions), which leads us to conclude that the predictor is practical to implement.

6.3.4 Intuition for Predictor Effectiveness

Our predictor works on the assumption that when a load has a larger number of consumers, the probability that *at least* one of its consumers lies on a critical path increases. For every committed load, we tracked the number of consumers as well as the instruction types of each consumer. Figure 48 shows the breakdown/percentage of consumer instruction type for loads with varying numbers of consumers. We separate the instruction types into six categories: low latency (instructions like add, subtract, multiply and so on), cache hit (loads that hit in the DL1 cache), correct branch (correctly predicted branches), cache miss (loads that miss in the DL1 cache), mispredicted branch, and long latency (instructions like divide and other complex instructions). The first three categories can be regarded as instructions that will not experience very long latencies and are therefore unlikely to significantly impact processor performance. The last three categories are instructions with long latencies and are very likely to be critical.

At first glance, it does not appear that many consumers are likely to be critical (cache miss, branch misprediction or long latency). Even in the cases where a load has twelve consumers, only about 20% fall into one of the likely-to-be-critical categories. This twelve-child load, however, only needs *one* of its children to be critical for itself to be critical. The probability of each consumer to be *not* critical is 80%, and so the probability of *all* twelve of the load's consumers not being critical is only $0.8^{12} = 6.9\%$. In practice, this probability may be even lower since we only consider the load's immediate consumers, whereas some of the low-latency operations may in turn have critical consumers.

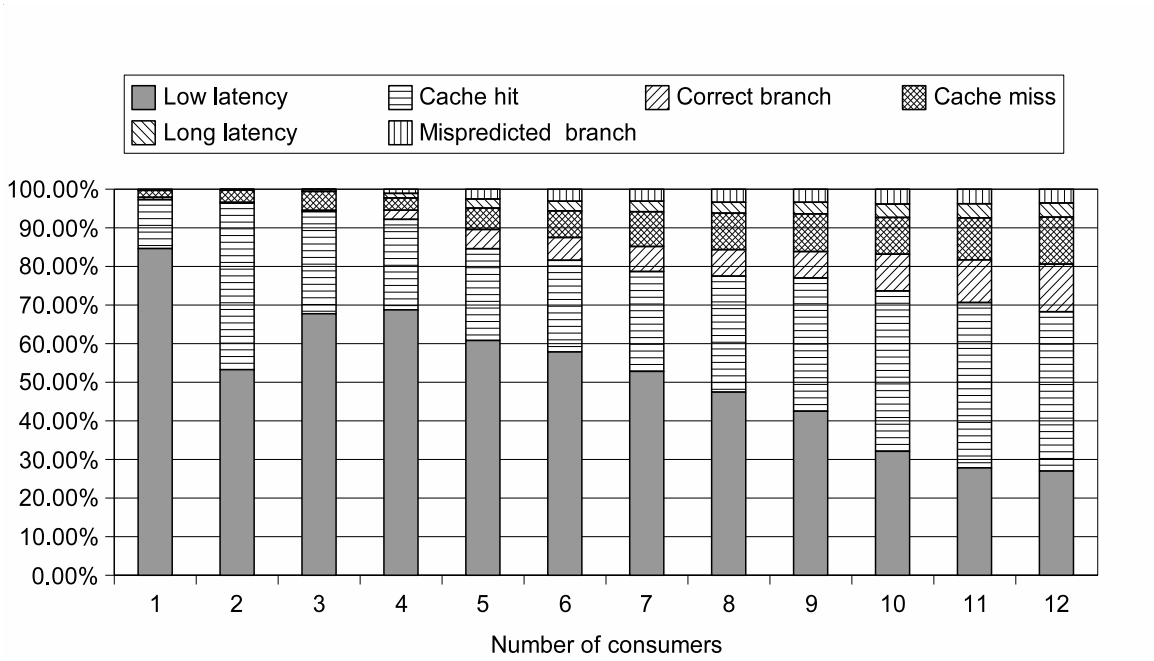


Figure 48: Distribution of consumers (for different consumer count values) according to instruction type.

The observed consumer instruction-type characteristics combined with this statistical argument provide a simple and intuitive explanation for why our predictor is able to perform very well. Srinivasan et al. proposed a predictor that explicitly used an instruction's type to determine criticality. Our results suggest that a load's consumer count alone can implicitly lead to the same final conclusion about whether a load is likely to be critical or not.

6.4 Simulation Methodology

Our simulation infrastructure uses a cycle-level model built on top of a pre-release version of SimpleScalar for the x86 ISA [3]. This simulator models many low-level details of a modern x86 microarchitecture including a detailed fetch engine, decomposition of x86 macro instructions to micro-op (μop) flows, micro-op fusion, execution port binding [72], and detailed cache architectures including all inter-level buses, MSHRs, etc. Our baseline configuration is loosely based on the Intel Core 2 [19], employing a 96-entry reorder buffer, 32-entry issue queue (RS), 32-entry load queue, 20-entry store queue, and a 14-stage minimum branch misprediction latency. The processor can fetch up to 16 bytes of instructions

per cycle (properly handling x86 instructions that span cachelines as two separate fetches) using a 5KB hybrid branch predictor [49], is 4-wide throughout its in-order pipelines (decode/dispatch/commit), and 6-wide at issue, with three integer ALUs, an integer multiplier, one load port, one store port, one FP adder and one FP complex unit (some of these units are bound to the same issue ports, which is why the number of functional units exceeds the issue width). We employ a load-wait table based memory dependence predictor [35]. The processor has 32KB, 3-cycle, 8-way L1 caches, a 4MB, 16-way, 9-cycle L2 cache, and DL1 and L2 hardware "IP-based" prefetchers [19]. Main memory has a simulated latency of 250 cycles.

We make use of programs from the following suites: SPECcpu2000 and SPECcpu2006, MediaBench [39, 25], BioBench [1] and BioPerf [5]. All simulations warm the caches and branch predictors for 500 million instructions and then perform cycle-level simulation for 100 million instructions. While 100 million instructions may seem rather short, we verified our results against runs of 500 million instructions and the results were nearly identical. We use the Sim-Point 3.2 toolset to choose representative samples [30]. Some applications do not yet run on SimpleScalar/x86 due to unsupported system calls and libraries.

6.5 Optimization Class 1: Faking the Performance of a Second Load Port

6.5.1 Problem Description

For modern out-of-order processors, a data cache with only a single read port may limit performance. Providing additional read ports is not simple because the area of SRAM arrays (such as the DL1 cache) increases quadratically with the port count. Besides the additional die area, the larger area increases the circuit path lengths which results in both higher latencies and higher power consumption. In an industrial simulator that we had access to, a second load data port increased the total active power of the processor by more than 13%. Furthermore, to feed two load ports, the main execution cluster may need an additional address generation unit (AGU), thereby forcing an increase in the issue width of the out-of-order execution core. As such, all out-of-order Intel x86 processors since the Pentium-Pro have only supported a single load port [72, 32, 27]. We would like to have the

performance benefits of a second load port, but we do not want to pay the area and power costs of physically adding another port to the cache array and increasing the out-of-order issue width to accommodate another AGU.

6.5.2 Implementation Details

Our first optimization uses load criticality to prioritize accesses to the single DL1 read port so that critical loads are given a higher priority. We use the consumer counter stored in the CLPT to determine how many immediate consumers are predicted to be waiting on a load. We also track whether a load was ready to issue but delayed due to contention for the data cache read port (i.e., there was another ready, older load that the select logic chose to issue instead). We augment each entry of the CLPT with the extra "ready-but-delayed" bit.

Our *non-critical load deferring* technique works as follows. If the CLPT value for the load is below the criticality threshold (five for our simulations), and the "ready-but-delayed" bit is 0, the load is determined to be not critical.

We then modify the select logic to give critical instructions higher priority. The select logic typically employs an oldest-first policy for choosing a ready instruction to issue. One way of implementing this is to have each instruction keep a timestamp, where the oldest instruction has the smallest timestamp [12], as shown in Figure 49(a). Instruction A is the oldest ready instruction (timestamp=000_2) and receives the grant to issue.[3] To modify the select logic to account for criticality, we simply extend the timestamp by a single bit: critical instructions will have this bit set to zero, while all other instructions will have this bit set to one. Figure 49(b) shows the same example instructions with the extended timestamps. Notice for example, that while load D comes later in program order than A (ignoring the extended bit of the timestamp, D has a timestamp of 011_2 while A has a timestamp of 000_2), the select logic effectively believes that D is the "older" instruction because its extended timestamp has a smaller value (0011_2 for D, and 1000_2 for A). Since we do not actually allow multiple loads to issue to the cache per cycle, this means that we do not require any additional tag broadcast buses for the wakeup/scheduling logic, nor do we need any

[3]Modifications to handle timestamp wrap-around are addressed in the original work by Buyuktosunoglu et al., and will not be further discussed here.

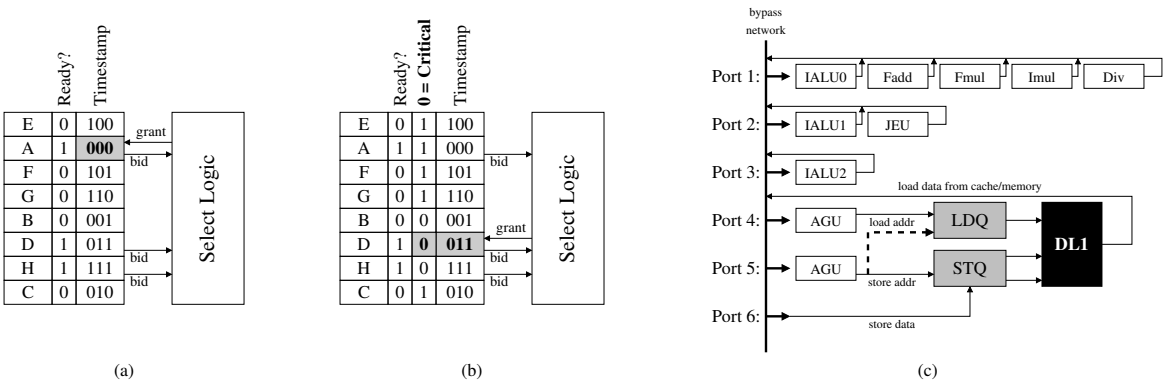


Figure 49: (a) Conventional timestamp-based oldest-first select logic, (b) augmenting the select logic to support criticality-prioritized select, (c) issue port arrangement with modification to use the store address AGU for a second load.

additional result bypass paths beyond the path already present for the original load port.

The second component that we consider is an additional load address port (address generation unit or AGU). Figure 49(c) shows the execution units and port assignments implemented in our simulator (based on the Intel Core 2 processor).[4] Note that load execution is effectively broken into two stages: first the load issues from the reservation stations to the AGU, which then deposits the address in the LDQ. The load may need to wait in the LDQ for some number of cycles until it is permitted to access the cache; this could occur due to predicted memory dependencies on earlier stores [50] or because the DL1 SRAM read port is busy. Note also that the execution units on issue ports four (load address), five (store address) and six (store data) do not connect directly back to the bypass network; the results of these three specific types of μops only get directly written to the load and store queues. We therefore propose to hijack the AGU on port five that currently only serves store address μops (denoted in Figure 49(c) as a bold, dashed arrow from AGU1 to the LDQ). The output of the AGU must now be routed to both the LDQ and the STQ, but this is far simpler than if we had to, for example, add another result bus to the bypass network. To prevent starvation of non-critical loads, we only allow a particular load to be deferred up to three times after which it gets priority to access the load port.

[4]We have taken the original 5-issue (two simple integer ALUs) Pentium Pro execution ports describe by Bob Colwell [72] and extended it for the 6-issue (three simple integer ALUs) of the Core 2.

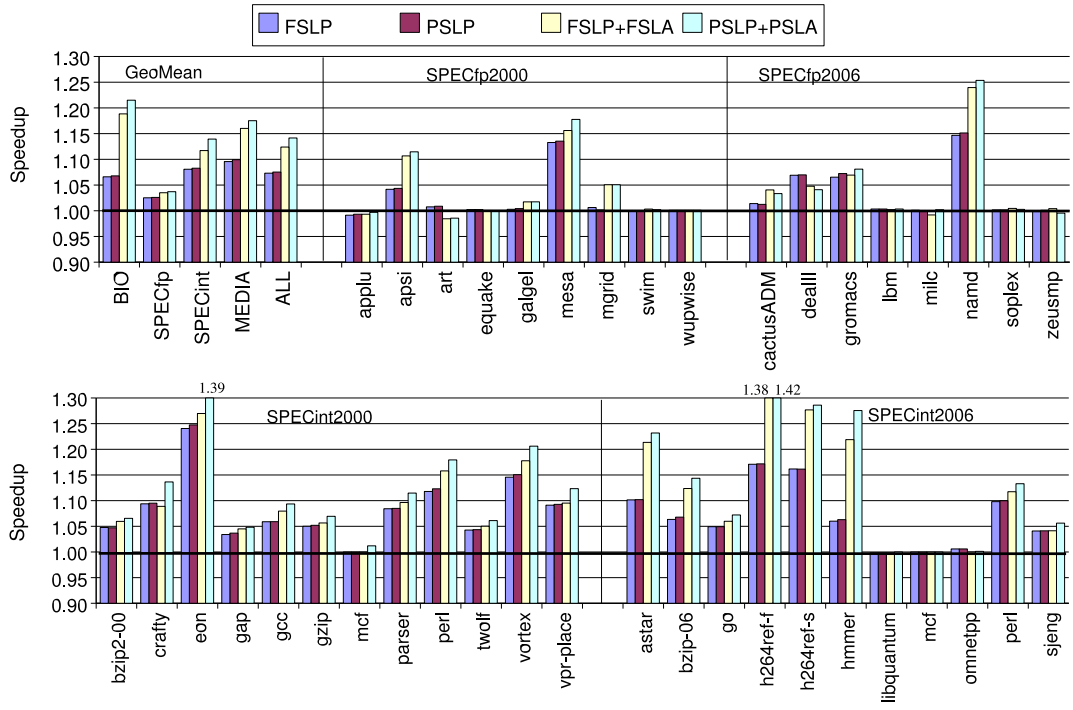


Figure 50: Deferring non-critical loads to achieve the performance of a second load data port.

6.5.3 Performance Evaluation

We present the performance of this application of load criticality in two scenarios. First we incorporate non-critical load deferring in a processor configuration with a single DL1 read port and a single load address port and compare this to a processor with a single load address port and two *true* DL1 read ports. Figure 50 shows the relative speedup when critical load prediction is used to prioritize load port usage. We present averages across all of the benchmark suites as well as present per benchmark results for the SPEC2000 and SPEC2006 suites. FSLP (Faking a Second Load Port) corresponds to deferring non-critical loads and PSLP (Pure Second Load Port) shows the performance if a complete additional DL1 read port is added. As we can see, for almost all of the simulated applications, FSLP is able to match the performance of the PSLP. For a few applications like *crafty*, *eon* and *bzip2-06*, the performance of FSLP is slightly lower than PSLP; in these applications we observed many loads being predicted as critical and hence could not be deferred. For *mgrid*, FSLP slightly outperforms PSLP. In this application, we observed extreme contention for the load ports in the PSLP configuration. In this case, it is better to issue a single critical load rather than simply the two oldest loads which may be non-critical. For our simulations, the PSLP results are slightly idealistic because we have not accounted for the latency increase that adding a second read port would cause; in practice, our FSLP's performance would be closer to, if not better than, a real dual-read-port DL1 cache.

In the second part of this experiment we simulated the baseline with two load address ports. One might wonder about the potential benefit of a second load AGU since the baseline configuration only has one load data port. Indeed, when we augment the baseline with an additional load AGU, the performance is almost equivalent to the baseline. We provide this comparison point to show that hijacking the store AGU to compute more load addresses in of itself is not of great benefit unless the data ports are better managed.

When the processor has either a second true DL1 read port or prioritizes accesses to a single read port as FSLP does, then the situation changes. In the case of the second DL1 read port, we simply need the additional AGU to keep the data ports busy. Providing two load AGUs *and* two DL1 read ports improves performance over the baseline by 14.8%.

In the FSLP technique, an extra AGU can expose additional ready loads, both critical and non critical, for the LDQ to send to the data cache (i.e., there are more opportunities to reschedule loads based on criticality). As explained earlier, rather than adding another functional unit for the second load AGU, we use the same functional unit that is used to compute store addresses. We refer to this strategy as FSLA (Faking Second Load AGU). The third bar in Figure 50 shows the performance of FSLP augmented with the FSLA technique (FSLP+FSLA), and the fourth bar shows the performance benefit of having two complete load data ports and two complete load AGUs (PSLP+PSLA). The PSLP+PSLA technique here is really an ideal truly dual-ported scenario. For some benchmarks like *hmmer*, this difference is close to 4%, where the dedicated second load AGU prevents loads from competing with stores for the AGU. Additionally, when there are more ready loads available, the PSLP technique is able to issue two loads per cycle. Even though some of these issued loads may be non-critical, they may expose some other independent work that can keep the processor busy. On average, however, the performance of FSLP+FSLA is within 1.7% of PSLP+PSLA, which is not much when one considers the low cost of our FSLP+FSLA compared to the overhead of implementing a true second load AGU and DL1 read port.

6.5.4 Why Does FSLP Work?

The previous sub-section demonstrated the performance benefit that load criticality prediction provides, when used to prioritize access to the data cache port. To understand why FSLP performs so well, we conducted an experiment that measures the average latency in cycles from when loads are ready to issue (address computation and memory disambiguation are complete) to when they actually issue. A delay in load issue in this experiment corresponds to the number of cycles that a load was forced to wait for the DL1 port to become available. We measure the average latency for both critical and non-critical loads for three configurations: a conventional DL1 read port, a DL1 read port augmented with FSLP and two DL1 read ports (PSLP). The aim of this experiment is to see whether the FSLP technique's performance benefit comes from being successful in reducing the average

Table 7: Average ready-to-issue delay in cycles for critical (CL) and non-critical loads (NCL).

Benchmark Suite	One Load Port		FSLP		PSLP	
	CL	NCL	CL	NCL	CL	NCL
BIO	14.8	16.4	3.4	14.4	3.3	6.6
SPECfp	12.1	16.2	4.4	11.5	4.4	8.5
SPECint	15.1	16.3	6.6	13.0	6.3	6.4
MEDIA	23.6	23.9	5.8	14.0	5.6	8.7

issue delay of critical load instructions. As far as non-critical loads are concerned, they do not generally impact the performance of the processor and hence can be delayed.

Table 7 shows, for all of the application suites, the issue delay in cycles for critical and non-critical loads for the three load port configurations. The first column of the table indicates the name of the suite, and the second and third column give the average latencies for a single DL1 port for critical and non-critical loads, respectively. The issue delay values might seem slightly high since the average value is biased by certain applications which experience very long delays as well as some effects of contention in the MSHRs. The fourth and fifth columns represent a single DL1 port augmented with FSLP while the last two columns correspond to the PSLP configuration. There are two important results that can be obtained from this experiment. First, in all four simulated benchmark suites, the issue delay of critical loads in the FSLP configuration is nearly equal to that of the PSLP configuration. When a load is critical, having a true second load port allows the critical load to issue much sooner. With FSLP, critical loads experience a very similar wait time. Second, since our FSLP approach still only has one real load port, the average ready-to-issue delay for non-critical loads is significantly higher than in the PSLP case. This has little impact on performance, however, since by definition these loads are not (likely to be) critical. These results explain why our FSLP approach performs nearly as well as the PSLP configuration.

6.6 Optimization Class 2: Data Forwarding and Memory Disambiguation

6.6.1 Data Forwarding

Every load normally performs an associative search of the store queue (STQ) to determine if it should receive a forwarded value from an older store to the same address. This CAM-based search comprises a significant portion of the store queue's power consumption [67]. To avoid performing these associative searches, one alternative is to force loads to wait until all earlier stores have committed and finished writing back to cache/memory. At this point, a load can safely issue to the data cache and be guaranteed to receive the most recent, architecturally correct value. Unfortunately, forcing all loads to wait for all earlier stores to commit and writeback can unnecessarily delay load execution. In our experiments, constraining loads this way caused an average performance degradation of 5% (but a 100% reduction in STQ searches). We observed, however, that while all loads associatively searched the STQ, on average only 13% of loads matched in the STQ and received forwarded data.

Using our load criticality predictor to predict, we only allow the critical loads to search the STQ. Only these loads can receive forwarded values from the STQ, while all other non-critical loads must wait until all older stores have written back to the data cache. The idea is that the power cost of repeatedly searching the STQ far outweighs the small potential performance benefit that can be achieved from earlier execution of loads that are not likely to be critical in the first place. Note that this optimization needs to be conservative since the prevention of even a few critical loads from receiving forwarded data through the STQ can significantly hurt performance, so we set a low criticality threshold of two. We also monitor the global issue rate when deciding if a load should be prevented from searching the store queue. If the issue rate of the processor is very low (less than three μops per cycle on average), we allow all loads, irrespective of criticality, to search the store queue.

Applying this optimization of critical-load prediction reduces over 93% of the store queue searches with less than 1% performance loss on average. Only one benchmark, *art* belonging to the SPECfp2000 suite saw a 7% slowdown due to missed forwarding opportunities. This further reinforces the fact that most of the STQ searches result in no matches, which has

been exploited by several other studies not directly related to load criticality [78, 68]. We extended our analysis to estimate the energy savings that this reduction in STQ searches would provide. We measured the activity numbers for the STQ in our simulator and used CACTI 4.1 to estimate the read, write and tag broadcast energy for our simulated STQ configuration [81]. Our results showed that reducing 93% of the store queue searches reduces the energy consumption of the STQ by 16%. Note that while this energy reduction does not account for the additional energy consumed by our predictor, we are not really explicitly proposing this STQ optimization to be implemented in *isolation*, but rather in conjunction with the other criticality-based techniques. Once one has already paid the cost of the predictor to provide the performance of a second load port, then this STQ optimization can effectively be had for no additional cost.

6.6.2 Memory Disambiguation and Memory Dependence Prediction

Our next application in this class involves speculative disambiguation of memory instructions. Before a load can issue, it needs to check if there are any older stores in the instruction window that have not resolved their addresses. If the load finds any older store with an unknown address, then the load cannot safely issue since it may have a true data dependency with this store. Most experimental data has shown that the majority of loads do not have true dependencies with any stores currently in the processor [50]. To prevent all loads from incurring this stall, some processors use memory dependence predictors that, based on past load behavior, predict whether a load will collide with any earlier unknown stores [19, 35]. If the prediction is incorrect, then all instructions after the load get flushed from the pipeline and need to be refetched.

Past research has shown that accurate memory dependence prediction has the potential to increase performance [17], but these memory dependence predictors need to be fairly large to achieve maximum load coverage and accuracy. In this optimization, we only allow critical loads to access the memory dependence predictor and speculatively execute. All non-critical loads must wait until all previous store addresses have been resolved. The non-critical loads can usually afford to wait and resolve all dependence ambiguities since they

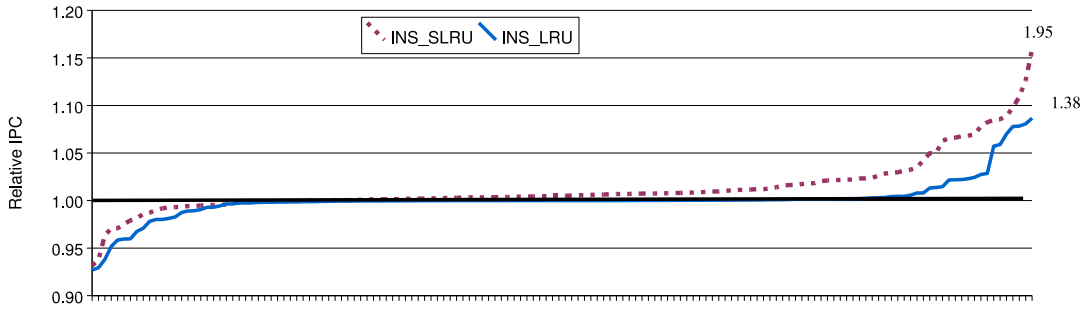


Figure 51: Relative speedup across the applications when all data is brought into the LRU position (INS_LRU) and when only non-critical data is brought into the LRU position (INS_SLRU).

will not have a significant impact on performance. There is no need to risk a misprediction and subsequent pipeline flush on a non-critical load where correctly speculating is unlikely to provide any significant performance benefits anyway. There is also less destructive interference in the memory dependence predictor since only critical loads update the predictor table. This allows a much smaller memory dependence predictor to be implemented while maintaining the same performance levels as the baseline predictor. In particular, using a memory dependence predictor modeled on a load-wait table predictor (like that used in the Alpha 21264 [35]), we were able to reduce the number of entries in the predictor table from 1024 to a meager 64 with no loss in performance.

6.7 Optimization Class 3: Data Cache

6.7.1 Insertion Policy for Non-Critical Loads

The first application in this class deals with the insertion policy used on a cache fill. Previous research has shown that inserting cache lines in the least-recently-used (LRU) position instead of the conventional most-recently-used (MRU) position can sometimes improve performance [57]. If the line is accessed a second time (cache hit), then it is promoted to the MRU position in the recency stack. The LRU insertion policy has the benefit of preventing lines with low reuse from residing in the cache for long periods of time. The drawback to this policy is that while it helps some applications that have very low reuse, it also hurts other applications with short or medium reuse distances. Adaptive selection of the insertion

policy has been propose to deal with this issue [57].

Rather than use an adaptive control scheme, we simply use our load-criticality predictions to guide the insertion policy. Data brought in by non-critical loads get inserted in the LRU position in the cache, where they are more vulnerable to eviction. All data read by critical loads follow the baseline MRU insertion policy, where several cache accesses must occur before the line is in danger of eviction. This organization uses a criticality threshold of eight.

Figure 51 shows the S-curves for all simulated benchmarks. Each point on a curve shows the relative speedup for one benchmark compared to a baseline processor that always inserts cache lines at the MRU position. Each curve is sorted from least to greatest speedup. The figure shows that inserting all data in the LRU position (INS_LRU = Insert all in LRU) causes performance slowdown in 30 of the 125 evaluated benchmarks, with a maximum speedup of 38%. Out of the 30 applications whose performance went down, five observe slowdowns of greater than 5%, and the worst degradation is for *lbm* (-6.8%). When load criticality prediction is used to control the insertion policy, the performance degradations are reduced as shown by the smaller tail at the left of the INS_SLRU (Insert Selectively at LRU) curve. The overall average speedup is 2.3%.

6.7.2 Cache Bypassing for Non-Critical Loads

Our second cache application deals with the placement of the data in the cache hierarchy. Some loads do not only have low reuse, but they sometimes have *no* reuse or extremely large reuse distances. In the first scenario, the data required by the applications are used only once and then sit idle in the cache until they are evicted. In the second case, the data is sure to get evicted from its set before it can be reused. In either case, these cache lines occupy space in the DL1 that could be better used by other data.

We use our load criticality information to simply prevent cache lines accessed by non-critical loads from being installed in the L1 data cache (i.e., they bypass the L1). This has similarities with McFarling's *Dynamic Exclusion*, where cache lines deemed to be likely to cause conflict misses are not allowed to be placed in a direct-mapped L1 cache [47].

Since this data is anyway only used by non-critical loads, we can afford the extra cycles required to access the L2 cache. Preventing non-critical loads from bringing data into the L1 reduces the cache pressure and can improve the hit rates for critical loads. As discussed in Section 6.1, while our non-critical load bypassing has similarities to other earlier proposals that prevent the data for non-critical loads from entering the DL1, our approach does not require building any additional caching structures such as the Non-Critical Load Buffer [24] of the Penalty Buffer [6]. We use a criticality threshold of four for this application.

6.7.3 Prefetching Policy for Non-Critical Loads

The third application deals with the DL1 hardware prefetcher. In our simulator, turning off data prefetching caused an average performance degradation of 2.9%. A prefetching algorithm is often difficult to tune and can end up polluting the cache by bringing in data that is never used. Prefetching can also cause additional contention for the off-chip bus, thereby causing further performance degradations. Unnecessary prefetches are even worse than the problems described for cache insertion and placement since, in those cases, the data that are brought in were used at least once. For this application we simply prevent non-critical loads from causing any prefetch requests. By restricting the prefetching algorithm to bring in data only for critical loads, cache pollution and bus utilization are both decreased. The criticality threshold for this technique is five.

6.7.4 Performance Evaluation

We now present results for the three applications described above. Figure 52 shows the performance speedup with respect to the baseline for the three optimizations. The first bar shows the performance due to inserting non-critical loads in the LRU position. While the overall performance improvement is 2.3%, some applications see large speedups. In particular, *mcf* sees a 95% improvement; *mcf* has low data reuse and experiences a 38% speedup even when the INS_LRU technique is applied. By limiting this strategy to only non-critical loads, the speedup is increased. Among the non-SPEC benchmarks suites, the media applications have nearly 5% performance improvement on average. Although not shown on this graph, *jpeg-encode* observes a 25% speedup. For this benchmark, the

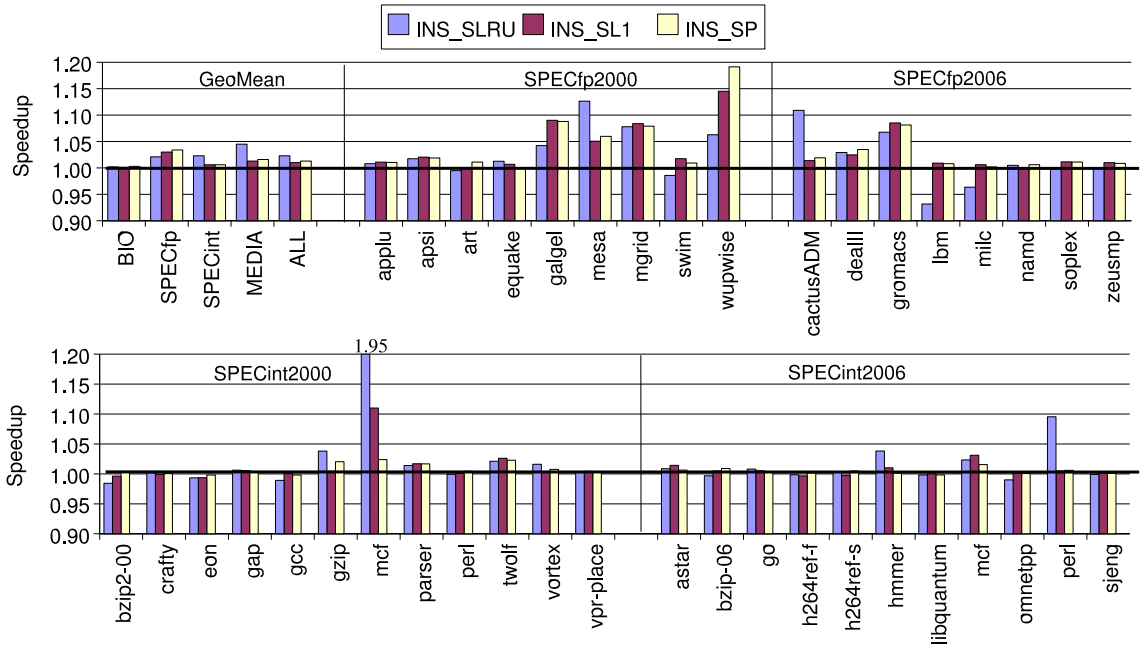


Figure 52: Speedup when critical load prediction is employed to optimize the L1 data cache.

INS_SLRU technique reduces the miss rate of critical loads in the L1 cache by 10%. When compared to the INS_LRU technique which inserts all loads in the LRU position, this strategy degrades the performance of fewer applications.

The second application in this class measures the performance benefit obtained by completely preventing non-critical data from entering the L1 data cache (INS_SL1 = Insert Selectively in L1). Figure 52 shows that this optimization achieves an average performance improvement of 3% for the SPECfp suite and 1% overall. While the average improvement is not very high, certain benchmarks do benefit.

Finally, the third bar in Figure 52 (INS_SP = Insert Prefetched Data Selectively) shows the impact of criticality-based filtering of prefetches. The results indicate that the floating point benchmarks seem to benefit the most by this technique. The average performance improvement for the SPECfp suite is nearly 4%, with *wupwise* showing speedup of nearly 20%. For *wupwise*, only 0.1% of prefetches in the baseline configuration turned out to be useful, so when load criticality prediction is used to filter out even some of the useless prefetches,

performance improves considerably. Even though each of these techniques only improves average performance by a few percent, we emphasize that these benefits are effectively free once the basic criticality predictor has been implemented.

6.8 Combining Our Criticality-Based Load Optimizations

In this section, we study the impact of simultaneously combining our optimizations. We present three sets of optimization configurations in Figure 53: faking a second load data port with hijacking the store AGU (FSLP+FSLA), combining all of the optimizations (ALL), and combining all except for hijacking the store AGU (ALL-FSLA). The ALL-FSLA configuration is provided to quantify the impact of all of the criticality-based optimizations without the impact of hijacking the store AGU. Note that for the ALL configurations, selective insertion policy takes precedence over the DL1 bypass optimization since it provided higher performance (insertion position is irrelevant if the DL1 is bypassed).

Figure 53(a) shows the average results for the individual benchmark suites. Most of the suites see considerable speedups even with the ALL-FSLA combination.The FSLP+FSLA technique provided a speedup of 12.3% on average, while the ALL combination achieves a speedup of 15.7%. Figure 53(b) shows the S-curves. When combining all optimizations (ALL), we see that the majority of benchmarks experience 10% or more improvement, a few benchmarks are performance neutral, and only a very small number exhibit some performance degradation. Recall that some of our optimizations target the reduction of power and the size of related hardware structures. For example, *art* (SPECfp2000) experienced a 7.0% performance degradation due to the fact that filtering non-critical loads from searching the STQ causes this benchmark to miss too many forwarding opportunities. When combining all of the optimizations, however, this degradation reduces to 3.5%. One can reduce this performance degradation even further by adjusting the thresholds to be more conservative about when to block loads from searching the STQ.

The FSLP+FSLA technique provides a substantial benefit for the relatively small investment of a less-than 1KB predictor and a few simple microarchitecture modifications. Once the predictor is in place for the purpose of achieving the performance of the second

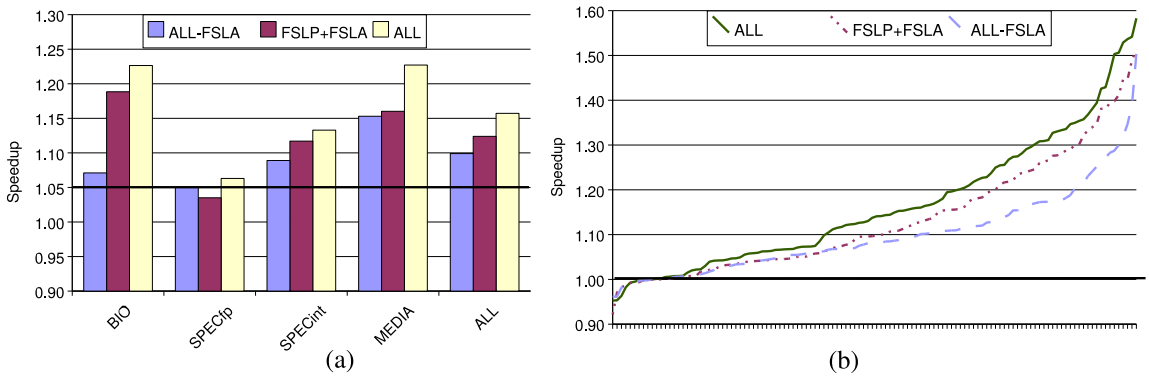


Figure 53: Speedup when critical load prediction is applied to all optimizations together: (a) per-suite averages and (b) S-curves for all benchmarks.

load data port, then the other optimizations which improve performance, reduce power and save area, are effectively free. Since we employ a single criticality predictor shared among all of the optimizations, it would not be difficult to make the different thresholds adjustable. Similar to how prefetch control parameters can be adjusted for different market segments [19], one could set different criticality thresholds to optimize for performance in the server markets, and for power in the mobile space.

While our criticality predictor is certainly not the first one to be proposed, we believe that it is simpler and more practical to implement than several of the earlier predictors. We simulated our criticality-based load optimizations (the simultaneous combination of all techniques on our academic simulator) using Tune et al.'s QCons heuristic that in each cycle finds the completed instruction with the largest number of children [84]. Note that in Tune et al.'s study, they used a 64K-entry predictor, which we model here, but we keep our own consumer-based predictor sized at only 1K entries. We also evaluated the Fisk and Bahar's predictor that tracks the number of dependent instructions added to a load's dependency list while it is servicing a miss [24], but found that the Tune et al. predictor was consistently better and therefore do not report the results for Fisk and Bahar's predictor.

In Figure 54, the first bar corresponds to the predictor that uses the QCons heuristic and the second bar corresponds to our predictor. The figure shows that the QCons heuristic is in fact a good indicator of load criticality as evidenced by the 12.3% average speedup

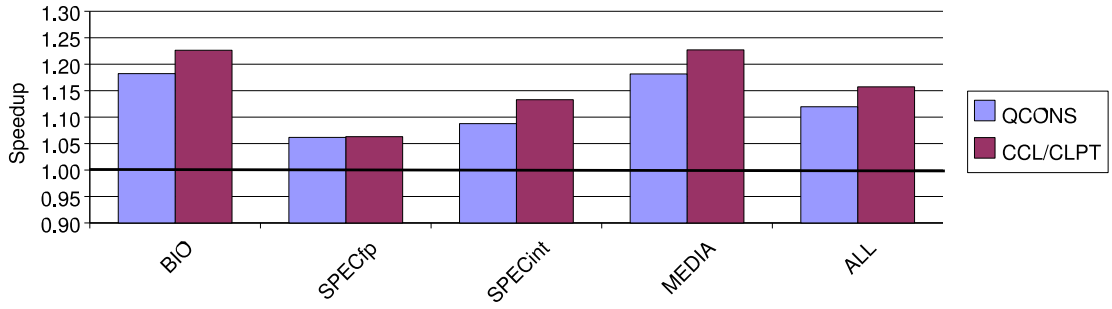


Figure 54: Speedup comparison of applying all proposed optimizations, but with different underlying criticality predictors.

that it provides. Despite the face that Tune et al.'s predictor uses 64× more entries than our predictor, we are still able to achieve higher performance. In particular, disabling our predictor during periods of low IPC helps to greatly reduce aliasing effects in the predictor, thereby allowing a small 1K-entry predictor to suffice.

6.9 Conclusions

In this chapter, we have demonstrated that predictability in load criticality can be exploited to improve the efficiency of modern processors. While load criticality has been extensively studied for optimizing value prediction and clustered microarchitectures, we have demonstrated that the idea is also useful for conventional, main-stream processor microarchitectures. We have been able to use *and reuse* a single, simple criticality predictor to optimize multiple facets of load execution. Our predictor, along with our optimizations that use the predictor, have all been designed with an eye toward minimizing microarchitectural impact. As such, we believe that these techniques are indeed practical to implement in near-term future microarchitectures.

Our load criticality predictor is simple and fairly non-invasive and its cost is more or less amortized over all the applications. The return on investment of this predictor is high considering the 15.7% speedup, 16% power reduction in the store queue and the area reduction in the memory dependence predictor. These performance, power and area benefits provided by our load criticality predictor and its associated applications help to improve

the overall efficiency of the processor.

CHAPTER VII

DEAD: DELAYED ENTRY AND AGGRESSIVE DEALLOCATION OF CRITICAL RESOURCES

7.1 Introduction

In this chapter, we reduce the commonly-occurring inefficiencies of conventional allocation and deallocation policies used in the load queue and the store queue so as to reduce their sizes and corresponding power consumption. Despite the power problem facing the processor industry, it appears that industry is still very interested in designing processors that achieve high performance. As a result there is a push toward more efficient microarchitectures that reduce power as much as possible (or atleast not increase it) while still providing high performance. Multi-core processors are one such example which can provide higher performance (if sufficient thread-level parallelism exists) than a traditional uniprocessor bound by the same power envelope. Conventional techniques to improve or even maintain single-threaded performance, however, still focus on implementing large structures that allow the out-of-order core to search more instructions from which to extract instruction-level parallelism. In our work, we take a different approach to the power-performance tradeoff problem. We attempt to relax the stringent conditions imposed on the allocation and deallocation of critical microarchitectural resources in the processor. By doing so, we are able to reduce the resource contention in these resources, thus allowing us to significantly reduce their sizes without losing any performance. Our designs for Deferred Entry and Aggressive Deallocation of the load queue (LQ), store queue (SQ) and the reservation stations (RS) allow us to build high-performing, relatively low-power queues that improve the efficiency of the processor.

In conventional processors, resource allocation and deallocation in the out-of-order core is generally done in program order. Forcing these stages to occur in program order may

introduce some inefficiencies. Some past work has explored delaying the allocation of physical registers beyond the regular allocation/dispatch pipeline stage [2, 18], and other work has considered mechanisms to allow instructions to commit out of program order [9, 18, 26]. In our research, we analyze the bottlenecks and opportunities imposed by in-order allocation and deallocation of microarchitecture resources. In particular, we find that the LQ, SQ and RS are inefficiently utilized. Based on our initial findings, we propose two novel techniques. The first relaxes the in-order constraints on the allocation of LQ, SQ and RS entries, effectively enabling a form out-of-order allocation. For the second technique, we make the observation that resource deallocation and instruction commit can, at least in part, be separated into two distinct tasks, which allows us to relax the in-order constraints on the deallocation of LQ entries. Our proposed techniques only require small changes to the microarchitecture, but they enable us to reduce the number of entries (and hence area) in the LQ, SQ and RS by 56%, 50% and 44%, respectively, with a negligible decrease in performance. This also has a direct impact on power consumption as the smaller structures consume less dynamic and leakage power.

In the next section, we will provide some background for our proposal by examining how much lost opportunity exists due to the in-order constraints on allocation and deallocation. We also describe related work to help establish the context for our contributions. Section 7.3 describes our scheme for supporting out-of-order allocation of the LQ, SQ and the RS, and Section 7.4 explains how to perform out-of-order deallocation of the LQ. Section 7.5 details our evaluation methodology. Section 7.6 evaluates the performance and power impact of our proposed techniques in the LQ, SQ and the RS. In Section 7.7, we present a quantitative comparison of our proposal against a technique for building scalable LQ/SQs, and Section 7.8 summarizes how our proposal improves the efficiency of the processor and presents our conclusions.

7.2 Background

In this section, we present some empirical case studies highlighting the inefficiencies that can arise from forcing the allocation and deallocation stages to operate in a strict in-order

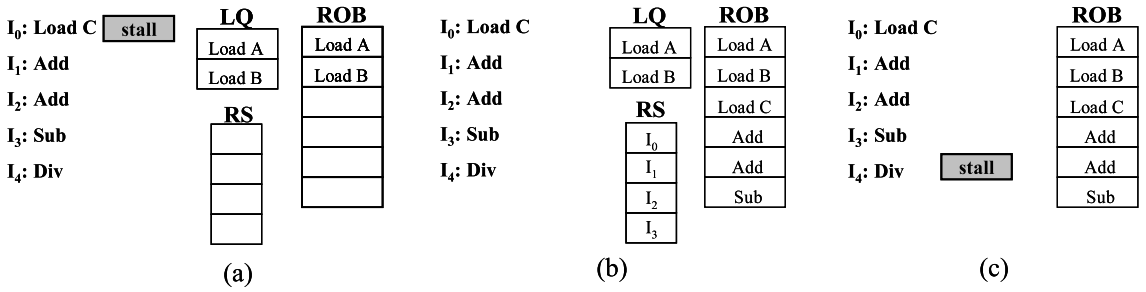


Figure 55: Potential of OOO Alloc: (a)load allocation stalls due to full LQ, (b)allocation continues (c)allocation only stalls due to full ROB

fashion. The full simulation infrastructure used in these studies is described in Section 7.5; the key parameters are a 96-entry ROB, 32-entry RS, 32-entry LQ and 20-entry SQ. We end the section with a discussion of related work.

7.2.1 Case Study: Stalls at Allocation

In the previous section we made the observation that an instruction that fails to allocate a ROB entry guarantees that all later instructions would not be able to allocate a ROB entry either, but the same guarantee does not hold for failed LQ and SQ allocations. We first consider the case of a load that fails to allocate a LQ entry, as shown in Figure 55(a). Note that the ROB and RS still have some entries available. If we could somehow ignore the fact that the load is missing its LQ entry and proceed with allocation of the following instructions, then the following three instructions would in fact be able to successfully allocate their resources (b). After this point, the divide instruction I_4 (as well as any instructions after that) stalls because no more ROB entries are available (c).

For some applications, this type of stalling can be common. For example, Figure 56(a) shows a histogram that measures for the benchmark *art* from SPECfp2000, for each load that stalled at alloc, how many subsequent instructions could have otherwise been allocated. For this application, allocation stalls due to a full LQ account for 68% of all stall cycles. This histogram shows that 27% of the time that a load is stalled at alloc, there is one additional instruction that could potentially be allocated because all of its other required resources are available. Another 24% of the time that a load is stalled at alloc, there are

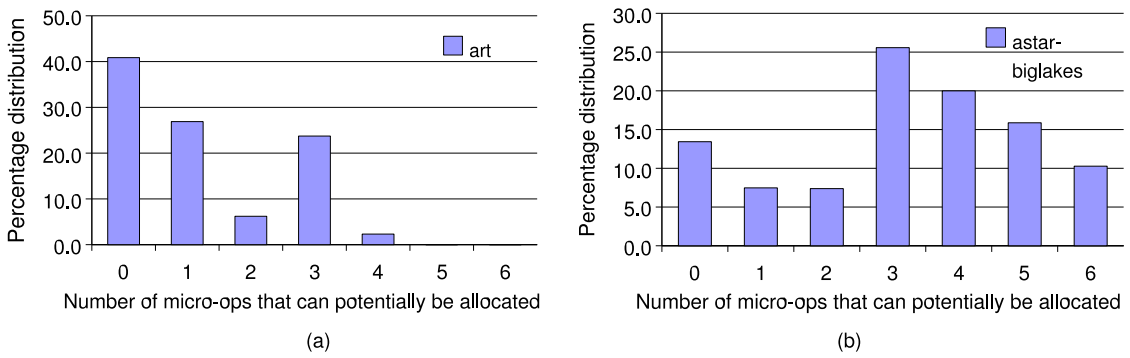


Figure 56: Percentage distribution showing number of instructions that are ready to allocate when a (a) load is stalled due to full LQ (b) store is stalled due to full SQ

three instructions after the load that have all of their resources needed to allocate, but they are stalled due to the earlier load. All of these instructions behind the alloc-stalled loads represent lost opportunities for exposing more instruction-level parallelism.

We collected the same type of data for SQ allocation-stalls. Figure 56(b) shows the number of unnecessarily stalled instructions that follow stalled stores in the benchmark *astar-biglakes* from SPECint2006. For this benchmark, 32.5% of all stall cycles were due to a full SQ. The histogram shows that 25% of the time that a store is stalled at alloc due to a full SQ, there are three instructions following the store that are ready and have all of the necessary resources to allocate. Even more telling is the fact that for this particular benchmark, there are a considerable number of missed opportunities where five or six non-store instructions that can be potentially allocated have been stalled due to a full SQ. In Section 7.3, we will describe an *out-of-order allocation* technique that exploits this behavior to avoid these unnecessary stalls.

While the previous histograms demonstrated the impact of stalls for specific programs, not all applications are dominated by LQ and SQ allocation stalls. Figure 57 shows the breakdown of stall cycles across all simulated benchmarks for a processor loosely based on the Intel Core 2 microarchitecture (full simulation details can be found in Section 7.5). This plot shows that across all simulated SPECcpu2000 and SPECcpu2006 applications, at alloc, stalls due to a full LQ account for 6.93% of all cycles and stalls due to a full SQ account for 9.62% of all cycles.

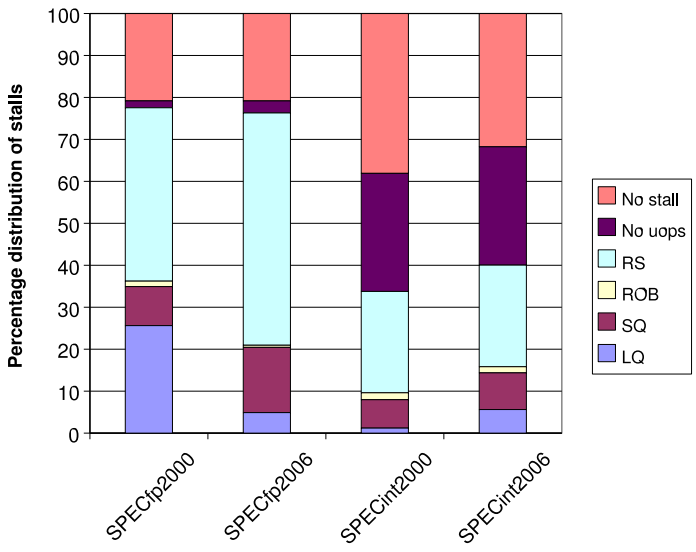


Figure 57: Distribution of stall cycles at alloc for SPECcpu2000 and SPECcpu2006 applications

7.2.2 Case Study: Unnecessary Delays for Deallocation

In this sub-section, we explain the inefficiency of in-order deallocation policies and describe the related stalls in the LQ that can occur. For an in-order deallocation policy, only when a load is the oldest instruction in processor is it allowed to commit and deallocate its LQ and ROB entries. As mentioned earlier, a LQ entry is responsible for holding the load address until the load can issue and access the DL1 cache (and search the SQ for a matching earlier store), and keeping the load address around so that earlier (in program order) stores that execute after the load (at a later cycle) can search the LQ to detect load-store order violations. If the load has already executed and it can be proven that this load cannot be guilty of a load-store ordering violation, then the load's LQ entry no longer serves a useful purpose. The time from when these two conditions first become true and when the load finally commits represents wasted use of the corresponding LQ entry. Note that all remaining functions related to commit can be handled by the load's ROB entry. The ROB entry includes the load's physical register, and therefore the final update to the ARF can (and does) happen directly from the ROB. Any faults associated with this load can be tracked by the ROB as well.

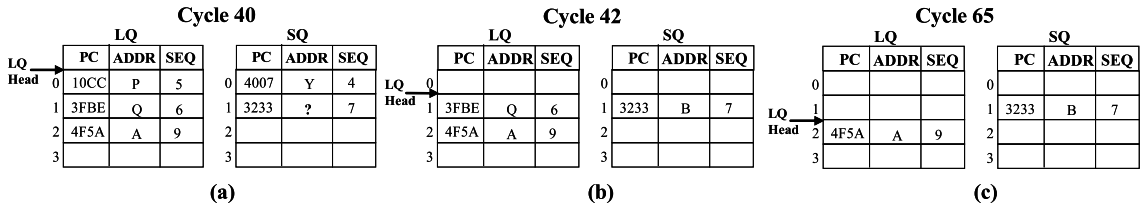


Figure 58: Inefficiency of in-order deallocation:(a)load executes speculatively on cycle 40, (b)last store address resolves on cycle 42, (c)load is forced to wait until it is oldest load to deallocate on cycle 65

To demonstrate this, consider the example in Figure 58. At cycle 40, the load (PC=4F5A) in LQ entry 2 has computed its effective address (A) and issued a request to the data cache to retrieve its data (a). The sequence id of this load is 9, indicating that the load is the youngest (most recently allocated) instruction. Note that this load execution is speculative since it is possible that the older store (PC=3233) SQ entry 1 with a sequence id of 7, which has not yet resolved its address, could potentially have a true memory dependence with this load. In cycle 42, Figure 58(b) shows that the oldest load and oldest store have committed and deallocated their LQ and SQ entries, respectively. During this cycle, the remaining store (SQ entry 1) has computed its address B which means that it does not have a store-to-load conflict with the last load (PC=4F5A). At this point in time, the load (PC=4F5A) is sure to have executed correctly, since no older stores to address A exist. Figure 58(c) shows that in cycle 65, this load becomes the oldest instruction in the reorder buffer, and it can finally deallocate its LQ entry (c). Note that in this example, starting from the end of cycle 42 all of the way until cycle 65, this load's LQ entry served no real functional purpose, but the conventional in-order deallocation approach does not allow this resource to be reused/reallocated during this entire interval.

As an example of this phenomenon, Figure 59 shows a histogram that measures for the *mgrid* benchmark from SPECfp2000, for each load that has completed and is sure not to misspeculate, how many cycles it is stalled before it deallocates its LQ entry. This histogram shows the percentage of loads that are forced to wait after they are guaranteed to have executed correctly for 1 through 9 cycles. One cycle is minimum possible since we assume that a load cannot complete execution and then commit in the same cycle. The

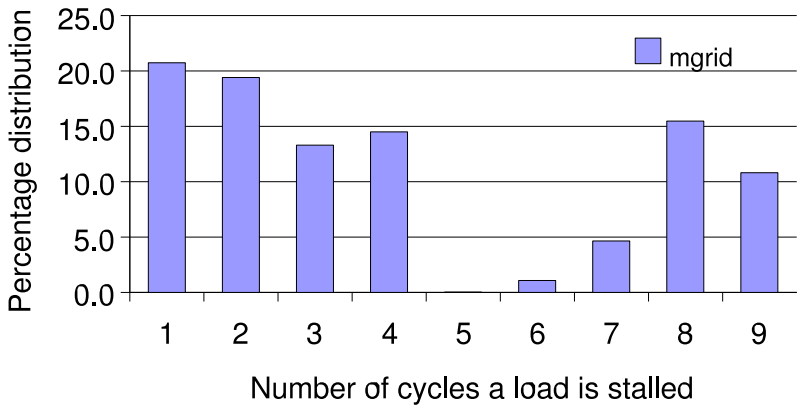


Figure 59: Percentage distribution of stalls cycles when a load is forced to wait to deallocate

histogram shows that, for this particular benchmark, 15% of all committed load instructions have waited for 8 cycles before they could commit and deallocate their respective LQ entries. These loads continue to occupy LQ entries, which can cause the LQ to become full, which in turn causes the allocation stage to stall as well. In Section 7.4, we will describe an *out-of-order deallocation* technique that can avoid these unnecessary stalls.

7.2.3 Related Work

Previous proposals for out-of-order allocation have mainly focused on the physical register file. Gonzales et al. describe tracking virtual free physical registers as a method of "virtually" allocating the physical register before it actually becomes available [2]. The virtual-physical register file technique is able to reduce the pressure on the register file since only a mapping to the actual physical register that will be allocated is required. The interesting observation that motivates this design is that a physical register is required only to hold a value produced by an instruction and is no longer required when all consuming instructions have read that value. Ephemeral registers is a similar technique that implements late register allocation by assigning a virtual tag at the time of actual allocation and making the actual physical register available only when the instruction issues [18]. Continual Flow Pipelines is another proposal that makes critical resources like the register file and the scheduler available to independent instructions that follow in the shadow of a cache miss [75]. In this technique, a load that misses drains out of the pipeline along with its

dependent instructions so that other independent instructions can use the resources, and the drained slice is processed once the miss returns. Most of these techniques focus on the physical register file and reservation stations to reactively deal with stalls once load misses have been uncovered, which requires large effective load and/or store queues [26]. Our proposed out-of-order allocation scheme primarily targets the inefficient use of the LQ and SQ and could potentially be applied to some of these schemes. More recently, unordered LSQs have been proposed which defer allocation of LQ/SQ entries until the load/store actually *issues* [**?**]. This form of deferred allocation is very aggressive and can help reduce the pressure on the load-store queue and allow for smaller-sized queues. This design, however assumes a unified LSQ as opposed to separate queues; hence every memory operation needs to access the entire queue. As a result, there is not much power benefit to using the smaller queues. There is however an area benefit since the size of the unified queue can go as low as 32 entries with no loss in performance.

Out-of-order commit and out-of-order deallocation have also been explored, but most proposals deal with checkpointing the ROB as a way to rectify any faults and preserve in-order execution semantics. Cherry is a technique that proposes the early release of registers and load/store queue entries by dividing the ROB into speculative and non-speculative regions [46]. The non-speculative entries in the ROB hold instructions that can be certain of not being misspeculated and thus can be released. Additionally, the processor maintains a checkpoint to handle precise exceptions. Most of these proposals are geared toward building large instruction windows and improving performance whereas our main goal is scaling down structures and reducing power while maintaining performance. As such, any checkpointing mechanism is likely to increase the area overhead as well as the power consumption. We do not require any virtual register pool, nor do we need to maintain any checkpoints. Bell et al. describe a methodology under which certain instructions can be proved to commit correctly even if committed out-of-order [9]. They define a set of necessary commit conditions (for example no occurrence of store-load or load-load replay traps) and if an instruction can satisfy these conditions it can commit out-of-order correctly. To our knowledge, no prior work has explicitly dealt with the out-of-order allocation and deallocation of LQ and SQ

resources.

7.2.4 Why Not Build Larger LQ/SQs?

In our work, we propose new techniques to reduce the size and power of several critical processor structures while maintaining very little performance impact. There are many related works for building *scalable* memory structures, such as CAM-RAM hybrid queues [8], Store-Queue Index Prediction [67], NoSQ [68], Fire and Forget [78], and several others. These scalable techniques effectively attempt to make normal-sized structures perform as well as much larger (but impractical) conventional structures.

At first thought, a technique that makes a regular-sized LQ or SQ perform like a larger queue should be directly applicable to make a small, low-power queue perform like a regular one. While this may be largely true for performance, applying these techniques to build smaller LQ and SQ's does not work from the perspective of power consumption. In particular, many of the techniques for scalable queues involve multiple predictors and commit-time load re-execution (and pipeline flushes) to ensure correctness. Compared to very large conventional queues, the power cost of the predictors, commit-time re-execution and pipeline flushes are relatively small and justifiable. When attempting to retro-fit these scalable queue techniques to implement smaller, lower-power structures, the power savings of the smaller queues is quickly annihilated by the overheads of the predictors and re-execution. While at first blush they may seem to be the same problem, building large scalable queues and building small efficient ones actually require very different approaches. We will revisit this issue in Section 7.7.

7.3 Out-of-Order Allocation through Deferred Entry

The previous section identified potential inefficiencies in the usage of both the load and store queues. In particular, we demonstrated how an allocation stall for an instruction due to the lack of a few resources can induce stalls in the allocation of other possibly unrelated instructions. In this section, we describe a design that can relax these in-order allocation constraints, specifically for loads and stores, thereby improving overall allocation efficiency.

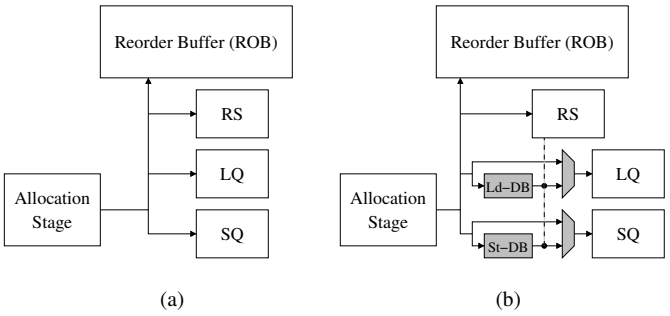


Figure 60: (a) baseline processor and (b) changes for deferred entry.

7.3.1 Description of the Technique

In this sub-section, we first describe our *Deferred Entry* technique in the context of load instructions and the load queue, and then briefly cover how this works for the store queue as well.

7.3.1.1 Load Queue

As described in Section 7.2, a load that cannot acquire a free LQ entry causes the entire allocation stage to stall in a conventional processor organization. Even though the LQ may be full, the ROB and RS may still contain available entries which subsequent non-loads should otherwise be able to use. When ROB/RS/LQ entries are available, the processor's alloc stage behaves the same as in a conventional processor. When the LQ becomes full and the alloc stage has a load instruction to deal with, our *Deferred Entry* (DE) technique allocates ROB and RS entries for the load, but defers entering the load into the LQ. Instead, the load instruction is shunted into an auxiliary structure called the *Load Deferral Buffer* (Load-DB). Figure 60(a) shows the allocation stage for a conventional pipeline, and Figure 60(b) shows the changes needed (shaded blocks) for implementing Deferred Entry. At some later point in time when the LQ eventually has a free entry, loads in the Load-DB can complete allocation by finally entering into the LQ. Our DE technique must prevent the load micro-op in the RS from executing because it does not yet have a valid LQ entry to write the computed address to. Note, however, that we can pre-determine the actual LQ index because at the time that the RS entry gets allocated (but the LQ allocation is deferred), we know that the next LQ entry to become available will be whichever entry

currently holds the oldest load (e.g., LQ_head). When the load completes allocation from the Load-DB, it sends a one-bit "go-ahead" signal (Figure 60(b) shows this as a dashed line) to its RS entry at which point the load micro-op can issue if its operands are ready.

After a load has been enqueued in the Load-DB, allocation can proceed for any instruction that follows the load. If the next instruction is not a load, then the alloc stage will simply acquire whatever resources are necessary for that instruction. If the alloc stage reaches another load, then that load can also be placed in the Load-DB. Note that if at any point the RS or the ROB becomes full, then allocation stalls for everyone. By forcing the loads to pre-allocate their ROB entries, the ROB continues to hold all instructions in the original program order, which therefore will not require any changes to its commit or misspeculation recovery mechanisms. Similarly, since we assume a unified ROB/PRF microarchitecture, allocation of a ROB entry implies allocation of a physical register, and so our DE mechanism does not require any changes to the register renaming logic.

7.3.1.2 Store Queue

The application of our Deferred Entry technique to the store queue is nearly identical to the load queue. When allocation attempts to acquire resources for a store instruction and finds that the SQ is full, the alloc stage first finds a free ROB and RS entry for the store, and then places the store in a *Store Deferral Buffer* (Store-DB). When the oldest store instruction deallocates its SQ entry, alloc can then move the deferred store from the Store-DB to the newly vacated SQ entry. Similar to deferring the entry of loads in the LQ, the corresponding store address (STA) and store data (STD) micro-ops in the RS cannot be allowed to issue until a SQ entry has been obtained. This seems like it might be a more complicated scenario, since the store allocating from the Store-DB must now send go-ahead signals to both STA and STD micro-ops which could potentially reside in different RS entries. Fortunately, modern x86 processors implement micro-op fusion where both STA and STD micro-ops get combined into a single RS entry, thereby avoiding the problem of routing twice as many go-ahead signals. Apart from this, the DE technique for the SQ is completely analogous to the LQ case. We now demonstrate the operation of DE with a

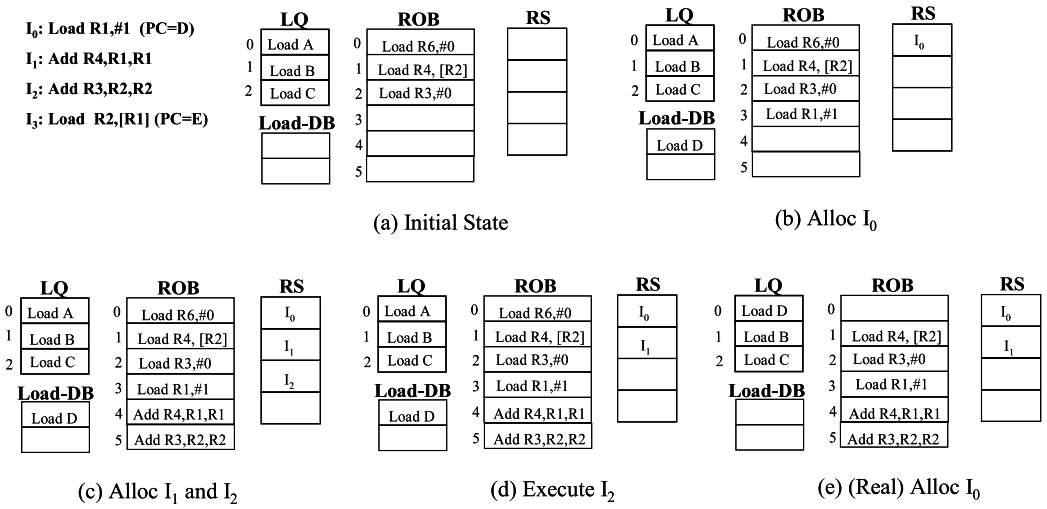


Figure 61: Step-by-step example of Deferred Entry.

detailed example.

7.3.2 Step-by-Step Operation

Since the basic methodology for Deferred Entry is the same for the LQ and SQ, our example only involves the LQ. In Figure 61, we show the allocation sequence for four instructions with DE. For the purpose of this example, we assume a 3-entry LQ, a 6-entry ROB, a 4-entry RS and a 2-entry Load-DB.

(a) Initial State: There are three loads that are already occupying the LQ entries; hence the LQ is full. The ROB still has some entries available and the RS currently has no entries allocated since the three loads have already issued.

(b) Alloc I_0: The first load has to be allocated. The ROB and RS have free entries, but since the LQ is full, this load would not be able to complete its allocation under a conventional allocation scheme (and would therefore stall the entire alloc stage). With DE, however, we have the additional Load-DB where we can place the load. The load allocates into the ROB and the RS in the usual manner.

(c) Alloc I_1 and Alloc I_2: The next two add instructions allocate. Since these adds have no knowledge that the load allocation was deferred, they will not stall. Note that we have effectively allocated these instructions out of program order.

(d) Execute I_2: The second add, which is independent of the deferred load, goes ahead and

executes. Note that the earlier load has not even completed allocation at this point in time. The first add has a data dependency with the deferred load; it simply waits in the RS until the load eventually allocates its LQ entry, executes, and broadcasts its tag.

(e) Completing Alloc for I_0: In this operation, the oldest load in the machine has deallocated its LQ entry, the deferred load acquires this entry and transitions from the Load-DB into the LQ proper. It also notifies the corresponding RS entry that it can go ahead and issue now.

Although not shown as part of this example, instruction I_3 will need to enter the Load-DB at alloc and wait there until another LQ entry becomes available. The operations for a store are identical, and the Store-DB operates exactly like the Load-DB.

7.3.3 Bonus: Extending DE to the Reservation Stations

In this section so far, we have explored how the technique of DE can be applied to the LQ and the SQ. This exploration was done with the assumption that allocation for loads/stores would be deferred *only* in the LQ/SQ respectively. Every instruction would still need to be allocated all other required resources such as the ROB and the RS. A deferred load/store, however will not execute until it is completely allocated and has been assigned an entry in the main LQ/SQ. Thus the deferred load/store does not really need an RS entry until it has been allocated a LQ/SQ entry. This RS entry can potentially be used by another independent instruction in order to increase throughput. Hence if we apply DE to the LQ and the SQ, we can also defer allocating an RS entry to the load/store in the Load-/Store-DB. Note that with this approach, we also avoid the need to implement separate "go-ahead" signals.

To avoid any kind of deadlock issues, we need to reserve one RS entry for a potentially deferred load, and one RS entry for a potentially deferred store (total two reserved entries). These reserved entries are not used by other instructions when the LQ or SQ are not empty. When attempting to defer the entry of a load or store, if the corresponding reserved entry is not available, then allocation must stall. This reservation ensures that when a load/store needs to move from the Load-/Store-DB to the main LQ/SQ, it can find an entry in the

RS as well. Since loads and stores are maintained in program order in the Load-DB and Store-DB, respectively, this reservation is all that is required for complete allocation. The RS entry can be populated from the load/store micro-op present in the DB.

7.4 Out-of-Order Resource Release through Aggressive Deallocation

The previous section described our implementation of Deferred Entry to relax the constraint on in-order allocation. In this section, we describe our design for aggressive resource deallocation for load instructions. As discussed in Section 7.2, committing an instruction involves making changes to the processor's architected state. This may involve writing to the architected register file, writing to memory, and dealing with interrupts and exceptions. These operations must occur in program order to maintain the in-order semantics specified by the instruction set architecture. Note that the above description says nothing about the deallocation of microarchitecture-level resources.

We propose *Aggressive Deallocation* (AD) to capitalize on this opportunity with a partial out-of-order release of LQ resources. This basically requires detecting three conditions. The first is that the load has completed execution, which is trivial to detect. The second is that the load cannot cause a load-store ordering violation. This is also trivial, because modern processor already effectively collect this information. For older processors without speculative memory disambiguation [50], as well as for processors that do support it, but the loads are predicted to have a dependence with an earlier store, the loads must wait until all previous stores have computed their addresses. We will consider the case where loads may speculatively issue in the presence of earlier unresolved store addresses, as this subsumes the case of no-load-speculation. If all earlier store addresses are known, then when the load issues, it will search the store queue. If all earlier store addresses are known, then the load will either match on one of these and receive a forwarded value, or it will not match on any and simply receive its value from the cache. If an earlier store address is not known, then the load will issue anyway (assuming it has not been predicted to have a dependence). At some later point in time when the earlier store address resolves, it will search the LQ to make sure there were no load-store ordering violations. If the load read

from the same address, then a violation would have occurred and the pipeline gets flushed, otherwise the load had correctly executed. Therefore, if a load has completed, all earlier store addresses are known (hardware already exists), and the load did not get squashed, then we are guaranteed that the load can no longer cause a load-store ordering violation. Our last criteria is that the load is the oldest load in the LQ, which does not necessarily imply that the load is the oldest *instruction* in the processor (ROB).

Once these three criteria have been fulfilled, our AD technique will relinquish the load's LQ entry which can in turn now be reallocated for newer loads stalled in the allocation stage. Our technique holds a load in the ROB even after it has been aggressively deallocated from the LQ, therefore precise exceptions can be maintained. While all loads can conceptually be deallocated as soon as they meet the conditions stated above, aggressive deallocation is useful only when the LQ is full or close to being full. Hence, we adapt our AD technique to deallocate a LQ entry only when the LQ has less than three available entries.

In a conventional LQ, memory consistency can be enforced by snooping the LQ on an external reference from another core. Since AD removes loads from the LQ, any attempts at snooping will miss the aggressively deallocated loads, possibly leading to violations in the memory consistency model. To enforce memory consistency, we effectively only need to track loads from when they deallocate from the LQ until they commit for real. For this, we employ a small, counting Bloom Filter that is indexed by the load address and is incremented when a load aggressively deallocates and decremented when the load actually commits. When an external reference is observed, the referenced address hashes into the Bloom Filter where a non-zero counter value indicates a potential consistency violation. In such situations, we conservatively flush the pipeline and re-execute the load. While this may seem like a very heavy-handed approach, this will only result in a spurious pipeline flush in very specific (and infrequent) situations. In particular, a load must have undergone aggressive deallocation, but not have yet committed, and the length of this period is typically not more than a dozen cycles. During this period, an external reference to a different address that still hashes into the same Bloom Filter entry must occur (an external reference to the same address would have caused a pipeline flush anyway, and so we do not introduce any

additional flushes in these situations). We only invoke AD when the LQ is almost full, which minimizes our exposure to spurious pipeline flushes due to false hits in the Bloom Filter. As a result, we simply make use of a very small 64-entry Bloom Filter, whose power overhead is included in our power analysis later in our results. The width of the counter in each Bloom Filter entry is small; for a 32-entry LQ, the counter only needs to be 5 bits wide to handle the worst possible (and highly unlikely) case where all 32 LQ entries are occupied, all are aggressively deallocated, and all entries hash into the same Bloom Filter entry.

Note that we do not apply AD to the SQ. The criteria for aggressive deallocation of stores is a bit more complex. In particular, a store must have completed and its value must have been read by *all* later dependent loads. The problem is that until this store has been completely overwritten by another later store to the same address, the processor cannot *guarantee* that it will not fetch another load that happens to read from this same address. Even if another store does completely overwrite this address, we must guarantee that that store will indeed complete (e.g., it cannot be in the shadow of a mispredicted branch). We believe that the hardware required to test all of these conditions is far to complex to implement in current processors, and so we do not pursue AD for the SQ any further in this work.

7.5 Evaluation Methodology

Our evaluation infrastructure uses a cycle-level simulator built on top of a pre-release version of SimpleScalar that supports execution of programs compiled for the x86 ISA [3]. This simulator models many low-level details of the microarchitecture including the cracking of complex x86 instructions to sequences of micro-operations, a fetch-engine that can speculate past multiple branches and even recover from mispredictions under the shadow of earlier but not yet resolved branch mispredictions, atomic x86 commit that stalls commit until all μops from the same instruction have completed and are free of faults. Our baseline processor is based on the Intel Core 2 [19], employing a 96-entry reorder buffer, 32-entry issue queue (RS), 32-entry load queue, 20-entry store queue, and a 14-stage minimum branch

misprediction latency. Since these parameters are taken from a real processor, we believe that our baseline model represents a well-balanced and fair starting point. The processor can fetch up to 16 bytes per cycle, decode and rename up to four x86 instructions per cycle with one complex and three simple decoders, issue up to six μops per cycle, and commit up to four μops per cycle.

In our simulator (and as generally implemented in hardware), we employ a senior store queue (SSQ) that holds stores that have committed, but not yet finished writing to cache. This SSQ is a virtual structure that is directly integrated into the conventional SQ. When a store commits, it writes its result to memory, but its SQ entry does not get deallocated until the write actually completes (which could be many cycles in the case of a cache miss). On commit, the store is *logically* moved from the regular SQ to the SSQ, although no physical movement of the SQ entry is required. An entry in the SSQ cannot be reallocated to a new instruction until the previous store has completed its writeback. As a result, both senior stores and regular stores compete for the SQ resources.

We used integer and floating point benchmarks from the SPECcpu2000 and SPECcpu2006 benchmark suites, although we were not able to include a few applications due to incomplete system call implementations in the current version of SimpleScalar/x86. We use the Sim-Point 3.2 toolset to choose representative samples [30]. We warm the caches and branch predictor for 500 million x86 macro instructions, and then perform detailed simulation for another 100 million instructions. All of our performance numbers are reported as speedups with respect to the baseline configuration. We use the geometric mean to report average numbers. The speedup numbers are reported in terms of the relative micro-ops committed per cycle (μPC).

For our power analysis, we incorporated a tool similar to Wattch in our simulator [10]. The simulator combines unit-level activity factors/access counts with per-block power models to estimate overall power consumption. Our power model is based on a 65 nm technology. Similar to Wattch, our tool collects activity numbers for all accessed structures in the various stages of the pipeline. Recent data suggests that leakage power accounts for 35% of total power in modern microprocessors, hence we use this estimate in our power model [54].

Table 8: Relative power contribution of different processor blocks. For the out-of-order and memory blocks, we include the individual power for some key units. The percentage contribution of these sub-units is already included in the macro-block's total.

Processor Block	Power	Processor Block	Power
Clock	25%	Memory Execution	27% (Total)
Front-End	8%	*LQ*	*4.4%*
Decode	10%	*SQ*	*3.6%*
Out-of-Order	24% (Total)	*DL1 Cache*	*11%*
RAT	*5.3%*	*L2 Cache*	*5%*
RS	*12%*	Execution Units	6%
ROB	*5%*		

Table 8 shows the per-block contributions to overall power.[1]

7.6 Evaluating the Application of Deferred Entry (DE) and Aggressive Deallocation (AD) in Critical Microarchitectural Resources

In this section, we evaluate the performance of our proposed techniques for deferred entry and aggressive deallocation when applied to the LQ, SQ and the RS. We evaluate three designs: applying DEAD individually to the LQ and the SQ, simultaneously applying DEAD to the LQ and the SQ, and simultaneously applying DEAD to the LQ, SQ and the RS.

Our primary motivation, as described earlier, is to reduce power consumption as much as possible while minimizing the performance impact. We achieve this by applying DEAD to the LQ, SQ and the RS which allows us to scale down their sizes with negligible performance impact. At various points, we measure the performance of the processor when augmented with our deferred entry and aggressive deallocation techniques and compare it to the performance of the original implementation without DEAD. We use non-uniform scaling factors while decreasing the size of the structures. We start with slightly larger scaling factors. At every simulated size, if there is a performance loss, we reduce the scaling factor until it reaches 1. After this we simulate all sizes until we observe the maximum allowable performance degradation. We set this threshold based on the performance of the optimized

[1]We had our per-block relative power consumption results validated by personal contacts at Intel (they would not release absolute numbers to us). Since we are reporting relative power reductions in this work, this model provides sufficient accuracy.

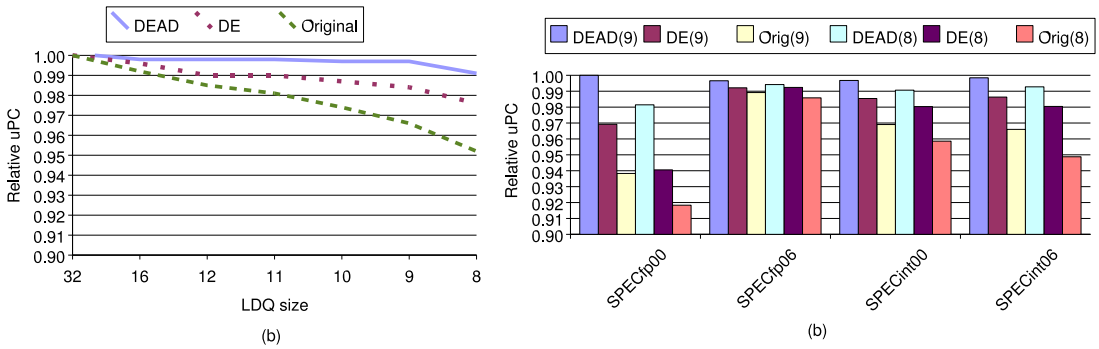


Figure 62: Performance impact of DEAD on LQ

scheme. When the optimized scheme reaches a configuration where the performance degradation is 1% with respect to the baseline, we stop scaling down the size further. The main idea in this study is that the performance of the DEAD technique should enable smaller sized structures remain competitive with the baseline processor. Additionally when the performance starts dropping off after a certain size, the DEAD technique should degrade more gracefully than the original unoptimized implementation. The DE component has the potential to hide stalls due to a limited size, since it allows the allocation of a certain number of instructions following an unallocated instruction. The AD component facilitates the DE technique for the LQ by removing loads as soon as they are completely ready to be deallocated.

7.6.1 Impact of DEAD on LQ

In this sub-section, we provide a detailed explanation of the performance impact of the DEAD technique when implemented for the load queue. Since our goal is maximum power reduction at a negligible performance cost, to evaluate the DEAD technique, we start with the baseline LQ size of 32 entries and gradually reduce the number of LQ entries. For the load queue, the performance benefit of allowing deferred allocation of loads depends on the number of independent instructions following the load that can be allocated. Since a deferred load cannot execute and retrieve data, instructions that depend on this load do not benefit from out-of-order allocation.

Figure 62(a) shows three performance curves (averaged over all benchmarks) corresponding to three configurations: using DE by itself on the LQ, using both DE and AD (DEAD) on the LQ, and the baseline that uses neither (Original). Starting with the base case of a 32-entry LQ, we reduce the LQ size and observe the overall performance impact. The x-axis represents different LQ sizes in decreasing order (note that the increments are non-linear). The y-axis shows the slowdown in terms of relative μPC for the decreasing LQ sizes compared to the baseline (32-LQ,20-SQ) configuration. The lowest curve on this plot (Original) shows that in a conventional allocation policy, initially scaling down the LQ does not hurt performance much. This result is also evident in Figure 57 which showed that at the baseline configuration, on average, the RS was more of a bottleneck. Scaling down beyond 16-entries, however, causes the performance to drop off rapidly. Beyond this size, the LQ becomes a critical bottleneck for the processor and the in-order allocation constraint drastically reduces the processor utilization. The DE curve shows that, on average, using deferred entry of loads provides higher performance than the conventional allocation policy even at smaller sizes. In fact, for an 11-entry LQ, allowing deferred allocation of loads loses just 1% performance with respect to the base case. Based on area estimates from CACTI [81], this provides a 30% reduction in LQ area. The performance numbers shown on this plot are for configurations employing a 4-entry Load-DB. We chose this size after performing a sensitivity study which showed that having more than four partially allocated loads does not provide much additional benefit. The Load-DB represents an additional power overhead not present in the baseline processor, but a four-entry FIFO buffer is not much compared to removing almost two-thirds of the LQ entries. Finally, the highest curve on this plot shows the performance when DE and AD are applied in conjunction to scale down the size of the LQ. The performance of this configuration, even at an 11-entry LQ, is almost equal to the baseline 32-entry LQ. A 9-entry LQ still manages to remain competitive with the baseline (0.5% performance loss), and an 8-entry LQ observes a 1% degradation. The 8-entry LQ provides over 50% reduction in area. The performance only starts really dropping off after the size is scaled beyond eight entries, which is only *one quarter* of the original capacity!

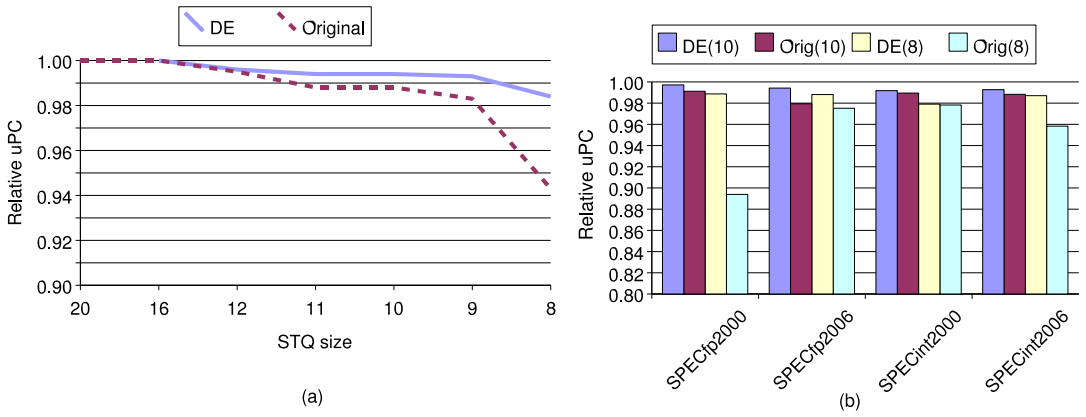


Figure 63: Performance impact of DE on SQ

After presenting the average performance for a spectrum of LQ sizes, we present per-suite averages for two specific configurations in Figure 62. We chose performance reduction cut-offs (with respect to the baseline configuration) of 0.5% and 1.0%, which corresponds to DEAD with a 9-entry LQ and an 8-entry LQ, respectively. The 9-entry LQ augmented with the DEAD technique performs nearly as well as the baseline 32-entry conventional load queue for all the benchmark suites. The performance of a LQ augmented with the DEAD technique also outperforms a conventional load queue of the same size across all the applications. Additionally, although not shown in the figure, the standard deviation in the conventional schemes is much greater than in DEAD. In particular, for the benchmark *galgel* from the SPECfp2000 suite, LQ allocation stall cycles account for 49% of all stall cycles in the conventional approach, due to which the performance falls by 20% with an 8-entry LQ. When DEAD is employed, however, the same configuration experiences only 1% stall cycles due to the LQ size constraint. In the conventional technique, scaling LQ sizes down to eight entries causes eight applications to suffer more than 5% performance loss.

7.6.2 Impact of DE on SQ

In this sub-section, we provide a detailed explanation of the performance impact of the DE technique when implemented for the store queue. As explained earlier, AD is not implemented for the SQ. Since our goal is maximum power reduction at a negligible performance

cost, we start with the baseline SQ size of 20 entries and gradually reduce the number of SQ entries. Unlike a load, a store will likely have fewer dependents since the only dependent a store could have is a load to the same address that needs the value of this store.

Figure 63(a) shows two performance curves which correspond to two potential allocation schemes for the SQ. The lower curve shown by the dotted line is the conventional in-order allocation approach (Original). The higher curve shows the performance when DE is applied to the SQ. The x-axis in this figure represents the different simulated SQ sizes in decreasing order and the y-axis represents the slowdown in relative μPC with respect to the baseline configuration. As can be seen in the figure, the performance of a 12-entry SQ using conventional allocation techniques as well as using DE is comparable to the baseline 20-entry SQ. Stores are fewer in number than loads and are generally not on the critical path; hence until the specific SQ size that impacts the allocation of other instructions is reached, the performance difference is marginal. After this point, however, the performance for the conventional allocation scheme drops off rapidly. Finally, at an 8-entry SQ size, the performance for the conventional allocation scheme drops by 5% with respect to the baseline. For the SQ configuration that employs deferred entry, even a small SQ of only eight entries loses just 1.2% in performance with respect to the baseline. The numbers shown on this plot represent a configuration with a 10-entry Store-DB. As explained earlier, even senior stores compete for the main SQ. Hence for small SQs, we need to maintain a 10-entry store FIFO to get maximum performance. This structure is a simple FIFO buffer that requires no CAM logic or indexing and consumes much less power than the original SQ.

In Figure 63(b), we present per-suite averages for a 10-entry SQ and an 8-entry SQ since these two points represent approximately 0.5% and 1% performance degradation, respectively. In this plot, we see that across all applications, a 10-entry SQ augmented with DE for stores outperforms a conventional SQ of the same size and is within 0.5% of the baseline 20-entry SQ performance. Using CACTI estimates, this provides nearly 20% area reduction for the SQ. For an 8-entry SQ without DE, the floating point applications see considerable degradation. One particular example is *equake*. In this application we observed more than 70% of the stalls were due to a full SQ.

7.6.3 Simultaneous Application of DEAD to LQ and SQ

In the previous sub-section, we analyzed the performance of the DEAD technique when applied separately to different structures. In this sub-section, we apply DEAD to both the LQ and SQ and optimize them together. Earlier, we were able to simply sweep the LQ or SQ size. Now that we are considering two variables at once, we employ a hill-climbing-like approach to find the best LQ/SQ sizes. We start at the baseline configuration of 32 LQ and 20 SQ entries, and then reduce the LQ by one entry, and then reduce the SQ by one entry. We choose whichever configuration results in the smallest average performance decrease, and then repeat the process starting from there.

7.6.3.1 Performance Results

Figure 64(a) shows the different (LQ,SQ) points that we obtained using our simulation approach on the x-axis. The y-axis shows the performance with respect to the baseline for the simulated configurations. In addition to comparison with the baseline, at every simulated point, we also compare with the unoptimized LQ/SQ configuration. That is, what would be the performance of a processor with the given LQ and SQ sizes and conventional in-order allocation/deallocation of the LQ and SQ entries. The dotted-curve corresponds to the conventional approach while the solid line represents the performance of an optimized LQ augmented with DEAD and an optimized SQ augmented with DE. As can be seen in the plot, for both curves, keeping the SQ size intact while reducing the LQ size provides the maximum performance until both are scaled down to 20 entries. Beyond this point, the LQ and SQ sizes impact each other which is why both the parameters are varied. Moreover, while the performance corresponding to the original technique drops off nearly linearly, the DEAD technique still remains competitive with respect to the baseline. In fact, even the configuration corresponding to a 13-entry LQ and a 10-entry SQ (13,10) only sees a 0.5% performance degradation. Finally, at a 9-entry LQ and 10-entry SQ configuration the performance of DEAD drops by 1% with respect to the baseline.

Similar to the earlier plots, we present per-suite averages at two configurations. We choose the two configurations which correspond to a 0.5% performance drop and a 1%

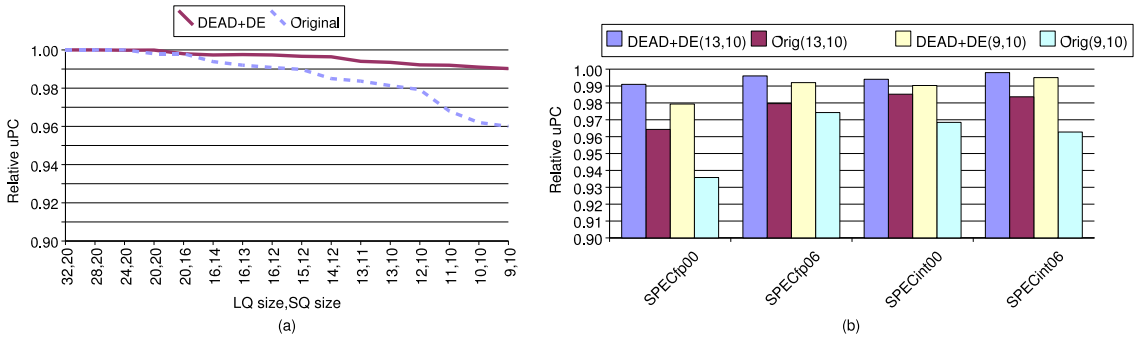


Figure 64: Performance impact of DEAD when simultaneously optimizing the LQ and SQ

performance drop to further analyze. Figure 64(b) shows average performance for the SPEC2000 and SPEC2006 benchmark suites for the (13,10) and the (9,10) configurations. The DEAD technique for the (13,10) configuration performs nearly as well as the baseline for nearly all the SPECcpu2006 applications.

7.6.3.2 Power Results

We now describe the potential power savings that can be achieved by simultaneously using the DEAD technique for the LQ and the DE technique for the SQ. Our power numbers indicate that the LQ and SQ together account to 8% of total processor power. When DEAD is applied for the LQ and SQ together we can reduce the sizes of the structures with minimal performance degradation. We measure the power savings at the 13-entry LQ and 10-entry SQ configuration since this configuration remained competitive with the baseline processor losing just 0.5% performance on average. By reducing the LQ by 19 entries and the SQ by 10 entries we can reduce the power of these structures by 50%. The load and store DBs that hold the deferred loads and stores are small buffers that require no broadcast and tag matching logic (we do account for the power overhead for these structures). The net result is a 4.9% reduction in power and a 4.8% improvement in the ED^2 metric. In the next sub-section, we will explain how we can further reduce the overall processor power.

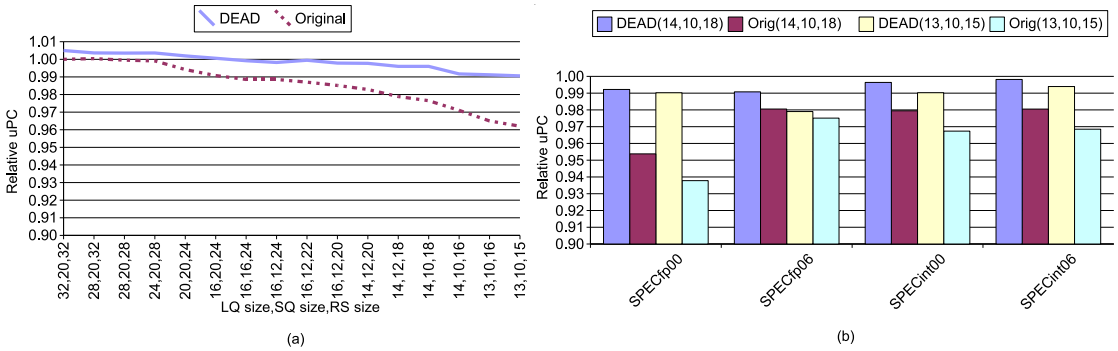


Figure 65: Performance impact of DEAD when simultaneously optimizing the LQ, SQ and RS

7.6.4 Applying DEAD to LQ, SQ and RS

In this sub-section we evaluate the impact of simultaneously applying DEAD to the LQ,SQ and the RS. Specifically we will use the DE technique to scale down the size of the RS. Reservation station entries are already deallocated aggressively (as soon as they execute). We use the same "least performance degradation approach" as described when applying DEAD for the LQ and SQ, except we now have three parameters to optimize. At every simulated point, we consider reducing the size of each of the LQ, SQ, and RS in turn, and select the choice that results in the smallest performance reduction, and then repeat this process. Through this scaling approach, we attempt to reach a balanced LQ-SQ-RS configuration that maximizes power reduction while minimizing the performance impact.

7.6.4.1 Performance Results

In Figure 65(a), the x-axis shows the different (LQ,SQ,RS) points we obtained using our simulation approach and the y-axis shows the the relative performance with respect to the baseline (32,20,32) configuration. The curves show a few interesting results. First, using DEAD on the LQ and DE on the SQ and RS (DEAD) for loads and stores, at the baseline processor configuration actually improves performance, albeit marginally. This indicates that even at the baseline (32,20,32) configuration, the RS is a frequently a bottleneck and by not allocating RS entries for deferred loads and stores, we can avoid unnecessary contention in the RS, which can improve performance. Even at (16,20,24), the DEAD

configuration matches the performance of the original baseline. After this point we can still continue to scale down the LQ, SQ and RS by applying DEAD while maintaining performance close to the baseline until we reach the (14,12,18) point, where we observe a 0.5% average performance degradation. Using DEAD, we can continue reducing the sizes of these three critical structures right down to a 13-entry LQ, 10-entry SQ and 15-entry RS with just 1% loss in performance. The conventional approach on the other hand performs poorly after the (16,20,24) point and experiences 4% performance degradation at the final simulated configuration.

The per-suite averages shown in Figure 65(b) indicate similar trends. As before, we picked the configurations that correspond to a 0.5% performance loss (14,10,18) and a 1% performance loss (13,10,15). Across all applications, scaling using DEAD outperforms a conventional allocation and deallocation scheme. In particular none of the applications observe slowdowns greater than 1.8%. The baseline implementation on the other hand observes considerable slowdown for a large number of applications at the 14-entry LQ, 10-entry SQ, 18-entry RS configuration. Several of the applications from the SPEC2000 suite observe slowdowns greater than 5%. Figure 66 shows the performance of each individual benchmark for our final configuration (14,10,18) for which DEAD experiences 0.5% performance loss on average. In the baseline case, the *galgel* benchmark shows a huge performance loss, which was earlier explained to be due to massive LQ allocation stalls. With DEAD, these stalls are largely avoided and the performance impact on this benchmark is greatly attenuated. Over all of the benchmarks, DEAD maintains performance quite close to the original baseline configuration (32-entry LQ, 20-entry SQ, 32-entry RS).

7.6.4.2 Power Results

After studying the performance impact of deferring the allocation of loads and stores in the three critical structures, the LQ, SQ and the RS, we measure the power benefits of being able to reduce the sizes of these structures. We measure the power savings at the 14 LQ,10 SQ,15 RS configuration since this configuration experienced only 0.5% slowdown with respect to the baseline 32 LQ, 20 SQ, 32 RS configuration. As mentioned earlier, the

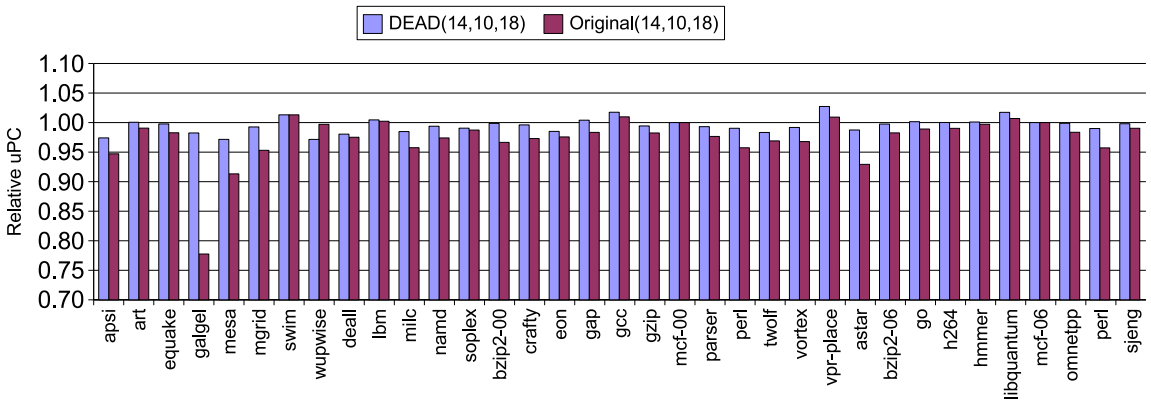


Figure 66: Per-benchmark performance for final configuration.

LQ and SQ together account for 8% of the processor power in the baseline. In addition to this the RS accounts for 12% of the total processor power. Reducing the number of RS entries by half, reduces the power of the RS by more than 54%. In addition to the savings provided by the scaled LQ and SQ sizes, applying DEAD at this configuration reduces overall processor power by 11%. Considering, that this configuration on average loses just 0.5% performance as compared to the baseline, the 11% power savings is quite significant. The equivalent improvement in the ED^2 metric is 10%. Note that our results are actually somewhat conservative. When these critical structures have their sizes reduced this much, their internal access latencies and circuit delays will now be much faster. For a fixed clock speed, this means that their circuit implementations can be relaxed (e.g., use CMOS instead of domino logic, or use slower/longer transistors to reduce leakage) while continuing to meet timing targets. This means that our 11% reported power savings really could be even greater if circuit-level re-optimization is employed.

7.7 Comparison with Scalable LQ/SQs

In Section 7.2, we qualitatively discussed why using scalable LQ/SQ designs to build small LQ/SQs does not result in a net power win. In this section, we provide additional experimental results to demonstrate this claim. We compare our DEAD approach to a generalization of a hybrid CAM+RAM approach proposed by Baugh and Zilles [8]. Their technique divided the SQ into two components: one capable of performing store-to-load forwarding,

and another simpler one that lacks such support. The highly-associative (CAM-based) forwarding SQ only needs to be large enough to support those stores that are (predicted to be) involved in store-to-load forwarding. For the remaining stores, it is relatively easy to build a large, scalable, non-associative (RAM-based) queue. The combination provides an effective solution for implementing a large effective store queue. A predictor determines whether a store gets allocated into the forwarding or non-forwarding store queue based on whether it was involved in store-to-load forwarding in the past. Since these predictions can be wrong, a store which should forward data may get allocated into the RAM-based queue, resulting in a later load not being able to get the correct value. The hybrid SQ makes use of Filtered Load Re-execution (FLR) to verify a load's data prior to commit [59], flushing the pipeline on an ordering violation.

We generalize Baugh and Zilles CAM+RAM SQ technique to the LQ as well. Any load that is predicted to receive data from an earlier store is allocated into a CAM-based forwarding LQ, and all other loads allocate into a non-forwarding LQ. All loads in the forwarding LQ can search the forwarding SQ for data. All stores in the forwarding SQ can search the forwarding LQ for ordering violations. Any ordering violations that involve loads or stores in the non-forwarding queues will be only be detected by the commit-time FLR. With high prediction rates regarding whether a load or store will be involved in forwarding, which has been shown to be relatively easily predicted [78, 67], this hybrid CAM+RAM approach can result in scalable LQ and SQ structures.

Figure 67 presents the performance of the Hybrid (Load and Store) Queues design in comparison to the baseline as well as the DEAD technique. For fair comparisons, the DEAD results in this section do not include reducing the RS size. The data points on the x-axis are derived by applying the same hill-climbing approach as used in the earlier studies; except that we scale the LQ and SQ size (the CAM components) according to the performance of the Hybrid Queues rather than DEAD. The RAM components are fixed at 30 entries for the LQ and 20 for the SQ, which were simply experimentally found to provide the best performance results.

The DEAD approach consistently outperforms the hybrid queue technique. The reason

the Hybrid Queues do not perform better is that we are starting with a configuration modeled on the Intel Core 2 processor, which is already balanced in terms of the sizes of the ROB, RS, LQ and SQ. In many of the scalable LQ/SQ works, the ROB and RS have been up-sized to make the LQ and SQ the primary bottlenecks in the processor. With a well-balanced machine where the LQ and SQ are much smaller than in the other scalability studies, there is not much performance to be gained. More so, when the sizes of the LQ and SQ are reduced, the overheads of commit-time re-execution and the occasional pipeline flush actually make the performance decrease.

We also analyzed the power consumed in a microarchitecture using Hybrid LQ/SQs. For our power analysis, we also include the overheads of filtered load re-execution with a 512-entry Store Sets Bloom Filter (SSBF) and 1024-entry Store PC Table (SPCT), and 512-entry predictors for determining whether a load or store should be allocated into the RAM-based or CAM-based queues. At the smallest configuration simulated (13,10), the Hybrid Queues design reduces the power consumption in the LQ and SQ together by 30% and overall processor power by 5%, *assuming that the forwarding predictor, commit-time re-execution and pipeline flushes costs no power at all.* When including the power consumption of the predictors, re-executions and pipeline flushes for the Hybrid Queues, the power savings is reduced to only 2.2%. Note that for the same sized queues, the DEAD approach also achieves a 5% power reduction, but since DEAD does not make use of predictions, it does not have to spend any additional power on auxilliary predictors and re-executions. These results illustrate our previous assertion that at smaller queue sizes, the power overheads of the auxilliary structures and re-execution for the scalable LQ/SQ techniques are no longer negligible, and that different techniques such as DEAD are needed to build low-power LQ and SQs.

7.8 Conclusions

The power consumption of microprocessors is a very serious challenge facing chip designers today. In this chapter, we eliminated the commonly-occurring inefficiencies that exist in allocation and deallocation policies for memory instructions. We present novel, yet simple

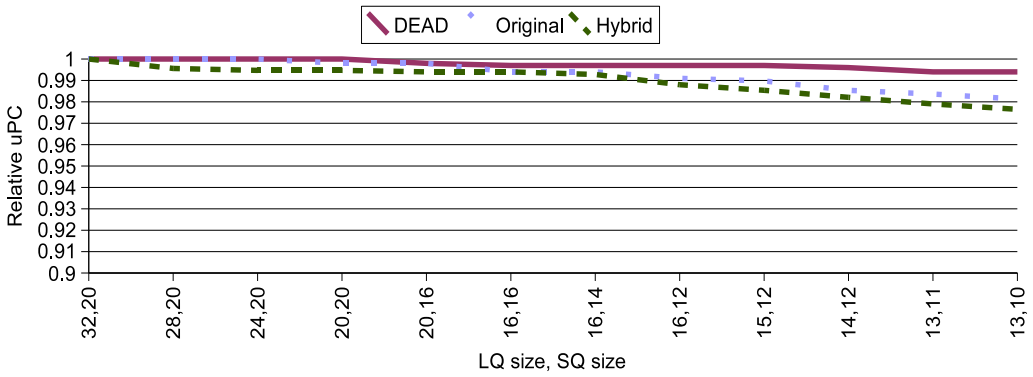


Figure 67: Performance of the Hybrid Load and Store Queues, DEAD techniques for the Load and Store Queues, and Original (Conventional) Load and Store Queues

techniques to reduce the power consumption of some of the most critical resources in a modern out-of-order microprocessor. Using Deferred Entry and Aggressive Deallocation, we reduce the sizes of these critical structures by more than half, thereby reducing the area required for them, too. We are able to reduce the power consumed by the LQ, SQ and RS significantly which provides a conservative estimate of a 11% total power savings.

The deferral buffers used in DE are fairly small, non-associative structures which are not very difficult to include in a processor. In comparison with to all past proposals that try and reduce the power consumption of the critical structures in a processor, DEAD provides a very simple design. The power and area savings achieved in the LQ, SQ and the RS without losing overall performance allow DEAD to improve the overall efficiency of the processor.

CHAPTER VIII

SUMMARY AND CONCLUSION

Processor efficiency can be improved by exploiting common-case and predictable behaviors of memory instructions. In this dissertation we have demonstrated that certain efficiency metrics, such as performance, power consumption, area overhead, scalability and implementation complexity can be optimized and improved by using the following memory behavioral patterns: predictability in memory dependences, predictability in data forwarding, predictability in instruction criticality and conservativeness in resource allocation and deallocation policies.

The main contributions of this dissertation are studying the impact of memory instruction processing on the processor efficiency and proposing novel designs and optimizations to improve conventional memory instruction processing. We proposed and evaluated five designs in this regard.

Our Store Vectors memory dependence predictor improves processor performance while incurring minimal design complexity by making use of predictable memory dependence patterns in loads and stores.

Our PEEP instruction fetch methodology methodology improves resource utilization, thus improving processor performance by applying accurate memory dependence prediction in the fetch engine of an SMT processor.

Our Fire-and-Forget data forwarding design uses common-case data forwarding patterns to reduce the power consumed by the critical memory processing structures, the load and store queues while providing a slight improvement in performance, thereby improving processor efficiency. Further additions to our base implementation also provide scalable processor designs that completely do away with the load and store queues while actually improving the performance of the processor.

Our Load Criticality Prediction technique provides an interesting classification of load

instructions based on their importance to processor performance and enables us to use the predictability in load instruction criticality in applications that improve the performance while reducing power consumption and area cost.

Finally, our DEAD designs significantly reduce the power consumption in and increase the performance potential of the load and store queues by reducing their sizes while relaxing stringent allocation and deallocation constraints in these queues.

Some potential future research avenues could explore the conjunction of these techniques to further improve processor efficiency. The application of the Store Vectors predictor for example in the PEEP fetch methodology could provide better prediction accuracy and increase processor throughput. The Load Criticality Predictor can potentially be used in the Fire-and-Forget design to only maintain prediction history for critical loads thereby reducing the hardware required for the data forwarding predictor. Fire-and-Forget can be used in conjunction with DEAD to reduce the power of the load and store queues by removing the associative control logic in them. Furthermore, due to the low implementation complexity and design scalability of all our techniques, they can be used across a range of processor sizes as well as in conjunction with other design proposals that eliminate inefficiencies in the processor.

This dissertation focuses primarily on single-threaded, single-core high-performance out-of-order processors (PEEP, is an exception which improves the efficiency of SMT processors). In this context, it is important to discuss alternative microarchitectural paradigms such as multi-core processors and embedded processors. The design of multi-core processors is in fact orthogonal to our proposed techniques. All our designs can be applied in the multi-core domain with minimal, if at all, design variation. Both Store Vectors and Fire-and-Forget can be used to improve the performance of each core in the processor while maintaining the constrained per-core power budgets. PEEP can also be useful in a multi-core design since some future multi-core designs are likely to also be multi-threaded [33]. The LCP and DEAD designs can be extended to improve the performance of each core while reducing the individual power and area overhead thereby making the entire processor more efficient. Embedded processors have slightly different design goals from high-performance processors.

Specifically, optimizations that focus more on reducing the power consumption may be more useful than, for example, the Store Vectors or the PEEP design. Modified versions of the DEAD technique might be useful in these types of processors. All these research avenues offer exciting scope for future work and might be very useful in improving processor efficiency. They are, however, out of the scope of this thesis dissertation.

In this dissertation we have proposed that processor efficiency can be improved by exploiting common-case and predictable behaviors of memory instructions, and have offered five novel design proposals (four of which have been published in peer-reviewed conferences and journals) which support our thesis.

CHAPTER IX

APPENDIX

The following tables provide additional results which present the performance of the Store Vectors memory dependence predictor for the large and extra-large configurations in comparison to other proposed predictors. We include results in Table 9, Table 10, Table 11 and Table 12 which provide a quick comparison in terms of performance of all the memory dependence predictors over the entire set of applications. While the base IPC refers to blind prediction, Perfect depicts the performance with an oracle predictor. The results indicate that Store Vectors nearly reaches the performance of a perfect predictor for most applications.

Benchmark Name			Base IPC	LWT	St. Sets	St. Vectors	Perfect
adpcm-dec	M		3.83	0.0%	0.0%	**0.0%**	0.0%
adpcm-enc	M		1.58	0.0%	0.0%	**0.0%**	0.0%
ammp	F	x	1.60	2.7%	2.9%	**3.9%**	3.9%
anagram	P	x	2.05	8.8%	11.4%	**12.6%**	12.7%
applu	F	x	0.90	4.9%	4.3%	**4.9%**	4.4%
apsi	F	x	2.23	2.4%	3.3%	**5.1%**	5.4%
art-1	F	x	1.68	-0.1%	0.1%	**3.3%**	3.3%
art-2	F	x	1.66	-0.1%	0.1%	**3.6%**	3.6%
bc-1	P	x	2.17	0.7%	5.1%	**8.4%**	8.2%
bc-2	P	x	1.11	0.0%	1.9%	**5.8%**	7.0%
bc-3	P	x	1.15	4.7%	4.7%	**5.4%**	5.5%
bzip2-1	I	x	1.57	0.2%	1.0%	**1.8%**	2.1%
bzip2-2	I	x	1.62	0.0%	0.4%	**2.1%**	2.3%
bzip2-3	I	x	1.72	0.0%	0.9%	**2.5%**	3.2%
clustalw	B		3.33	0.1%	0.2%	**0.3%**	0.3%
crafty	I	x	1.25	3.0%	6.0%	**7.4%**	7.5%
crc32	E		2.62	0.0%	0.0%	**0.0%**	0.0%
dijkstra	E		2.15	0.0%	0.0%	**0.3%**	0.4%
eon-1	I	x	1.26	-3.2%	10.4%	**18.7%**	19.2%
eon-2	I	x	0.92	0.4%	2.7%	**5.8%**	6.0%
eon-3	I	x	1.03	0.0%	4.4%	**10.7%**	10.3%
epic	M		1.86	0.0%	0.0%	**0.0%**	0.0%
equake	F	x	1.00	5.0%	5.0%	**7.3%**	7.3%
facerec	F	x	2.19	6.8%	7.1%	**7.9%**	8.0%
fft-fwd	E		2.24	0.0%	0.0%	**0.4%**	0.4%
fft-inv	E		2.24	0.0%	0.0%	**0.3%**	0.3%
fma3d	F	x	1.14	3.1%	4.2%	**9.5%**	11.3%
ft	P		2.15	0.1%	0.1%	**0.3%**	0.4%
g721decode	M	x	1.76	0.0%	1.4%	**1.7%**	1.8%
g721encode	M	x	1.64	0.5%	1.1%	**1.8%**	1.7%
galgel	F		3.34	0.0%	0.0%	**0.0%**	0.0%
gap	I	x	1.28	0.0%	0.3%	**3.4%**	5.4%
gcc-1	I	x	2.33	0.0%	0.2%	**17.6%**	17.7%
gcc-2	I	x	0.76	-7.0%	3.7%	**5.0%**	5.8%
gcc-3	I	x	0.82	0.0%	2.0%	**3.9%**	4.2%
gcc-4	I	x	0.96	1.0%	2.2%	**4.2%**	4.4%
ghostscript-1	E	x	1.39	8.3%	15.1%	**16.4%**	17.9%
ghostscript-2	M	x	1.40	7.6%	14.7%	**15.2%**	17.4%
ghostscript-3	E	x	1.63	12.7%	15.3%	**15.2%**	16.2%
glquake-1	G	x	1.04	1.4%	3.7%	**5.1%**	5.4%
glquake-2	G	x	1.17	1.8%	4.0%	**5.2%**	5.3%
gzip-1	I		1.79	0.0%	0.0%	**0.6%**	0.5%
gzip-2	I		1.37	-3.2%	0.2%	**0.3%**	0.2%
gzip-3	I	x	1.49	1.0%	1.1%	**1.5%**	1.7%
gzip-4	I		1.81	0.0%	0.0%	**0.5%**	0.5%
gzip-5	I		1.34	-0.1%	0.4%	**0.6%**	0.5%
jpegdecode	M	x	2.21	-4.9%	1.2%	**2.3%**	2.3%
jpegencode	M	x	2.11	4.0%	4.7%	**6.4%**	6.2%
ks-1	P		1.23	0.0%	0.0%	**0.0%**	0.0%
ks-2	P		0.78	0.0%	0.0%	**0.0%**	0.0%
lucas	F	x	1.08	9.2%	5.7%	**9.4%**	9.5%
mcf	I		0.75	0.0%	0.6%	**0.9%**	0.9%

Table 9: The performance of the memory dependence predictors simulated on the 'large' configuration for the entire set of applications. An 'x' signifies that the benchmark is dependency sensitive ($> 1\%$ performance change between blind prediction and perfect prediction).

Benchmark Name			Base IPC	LWT	St. Sets	St. Vectors	Perfect
mesa-1	M		1.49	-0.9%	-0.1%	**0.4%**	0.7%
mesa-2	F	x	1.58	17.1%	31.7%	**39.1%**	40.3%
mesa-3	M	x	1.06	-0.1%	0.8%	**1.5%**	1.6%
mgrid	F		1.41	-0.3%	0.0%	**-0.5%**	0.0%
mpeg2decode	M	x	2.65	5.2%	5.6%	**6.4%**	6.4%
mpeg2encode	M		3.13	0.0%	0.0%	**-0.6%**	0.8%
parser	I	x	1.26	2.2%	7.4%	**8.9%**	9.2%
patricia	E		1.09	0.0%	0.3%	**0.0%**	0.3%
perlbmk-1	I	x	0.95	5.9%	9.9%	**13.9%**	14.6%
perlbmk-2	I	x	1.22	-6.6%	-0.4%	**3.9%**	3.1%
perlbmk-3	I	x	2.36	1.5%	2.4%	**7.3%**	7.6%
phylip	B	x	1.42	0.0%	0.0%	**1.1%**	1.2%
povray-1	G	x	0.77	13.1%	19.0%	**22.0%**	22.5%
povray-2	G	x	1.01	3.3%	7.3%	**10.6%**	11.2%
povray-3	G	x	1.06	12.2%	18.8%	**22.2%**	23.4%
povray-4	G	x	1.04	6.3%	11.5%	**14.3%**	14.9%
povray-5	G	x	0.88	3.8%	8.0%	**10.3%**	10.5%
rsynth	E	x	2.19	-0.2%	0.9%	**1.7%**	1.7%
sha	E	x	2.85	9.2%	9.7%	**9.7%**	10.2%
sixtrack	F		1.30	0.0%	0.0%	**0.4%**	0.0%
susan-1	E	x	2.34	0.3%	1.0%	**5.1%**	5.1%
susan-2	E		1.85	-0.1%	0.0%	**0.0%**	0.0%
swim	F		0.86	0.2%	0.0%	**2.5%**	0.0%
tiff2bw	E		1.52	-0.1%	0.0%	**0.8%**	0.8%
tiff2rgba	E	x	1.53	-0.2%	0.1%	**30.6%**	30.3%
tiffdither	E	x	1.59	1.6%	3.8%	**4.3%**	7.3%
tiffmedian	E		2.47	-0.6%	0.0%	**0.5%**	0.4%
twolf	I	x	0.86	1.4%	1.9%	**3.2%**	3.3%
unepic	M		0.40	0.0%	0.0%	**0.0%**	0.0%
vortex-1	I	x	1.53	30.2%	30.3%	**36.9%**	36.9%
vortex-2	I	x	1.08	31.0%	33.4%	**45.0%**	46.3%
vortex-3	I	x	1.42	34.8%	36.6%	**47.9%**	55.2%
vpr-1	I	x	1.07	0.0%	5.0%	**6.3%**	6.1%
vpr-2	I	x	0.66	-0.8%	3.3%	**4.0%**	4.3%
wupwise	F	x	2.45	3.0%	3.1%	**3.9%**	3.9%
x11quake-1	G	x	1.58	2.6%	5.3%	**6.2%**	6.4%
x11quake-2	G		2.89	-0.4%	0.3%	**-2.5%**	-2.7%
x11quake-3	G	x	1.57	2.6%	5.4%	**6.4%**	6.6%
xanim-1	G		3.27	-0.82%	0.0%	**0.1%**	0.2%
xanim-2	G	x	2.86	0.9%	1.3%	**1.7%**	1.6%
xdoom	G	x	2.07	4.4%	5.3%	**6.4%**	10.7%
yacr2	P	x	2.14	0.1%	1.6%	**1.1%**	1.4%

Benchmark Group			Base IPC	LWT	St. Sets	St. Vectors	Perfect
ALL			1.51	3.2%	4.9%	**6.7%**	6.8%
SpecINT	I		1.21	2.2%	5.3%	**8.6%**	8.9%
SpecFP	F		1.51	3.1%	3.1%	**4.7%**	4.6%
MediaBench	M		1.68	2.7%	4.3%	**5.0%**	5.1%
MiBench	E		1.94	2.8%	3.6%	**4.2%**	4.5%
Graphics	G		1.47	4.6%	7.2%	**8.5%**	8.9%
PtrDist	P		1.49	1.9%	3.1%	**4.1%**	4.3 %
BioBench	B		2.17	0.1%	0.1%	**0.7%**	0.8%
Dep. Sensitive		x	1.32	4.5%	8.0%	**10.7%**	11.1%

Table 10: Continuation of the data from Table 9.

Benchmark Name			Base IPC	LWT	St. Sets	St. Vectors	Perfect
adpcm-dec	M		3.83	-0.02%	0.0%	**0.0%**	0.0%
adpcm-enc	M		1.58	-3.3%	0.0%	**0.0%**	0.0%
ammp	F	x	1.60	0.8%	1.34%	**2.3%**	4.4%
anagram	P	x	2.05	10.6%	12.1%	**12.7%**	19.5%
applu	F	x	0.92	-0.3%	0.0%	**10.11%**	12.5%
apsi	F	x	2.38	2.7%	1.1%	**1.1%**	1.5%
art-1	F	x	1.70	1.9%	0.0%	**3.9%**	3.8%
art-2	F	x	1.66	1.9%	0.0%	**4.2%**	4.1%
bc-1	P	x	2.18	2.2%	6.6%	**8.7%**	9.9%
bc-2	P	x	1.11	-2.2%	3.0%	**5.7%**	8.0%
bc-3	P		1.27	-1.9%	0.3%	**0.7%**	0.7%
bzip2-1	I	x	1.56	-2.3%	1.4%	**2.2%**	3.5%
bzip2-2	I	x	1.57	-3.0%	0.7%	**5.36%**	2.5%
bzip2-3	I	x	1.71	-3.1%	2.2%	**3.1%**	4.7%
clustalw	B		3.33	0.0%	0.2%	**0.4%**	0.3%
crafty	I	x	1.24	5.2%	7.0%	**8.1%**	7.5%
crc32	E		2.62	-11.0%	0.0%	**0.0%**	0.2%
dijkstra	E		2.15	-7.82%	0.0%	**0.5%**	0.4%
eon-1	I	x	1.26	13.9%	16.4%	**19.8%**	26.8%
eon-2	I	x	0.93	1.4%	4.19%	**5.8%**	29.0%
eon-3	I	x	1.02	5.2%	8.1%	**10.7%**	10.5%
epic	M		1.86	-1.3%	0.0%	**0.0%**	0.0%
equake	F	x	0.99	-0.4%	0.0%	**19.4%**	21.0%
facerec	F	x	2.13	9.0%	9.3%	**16.1%**	16.0%
fft-fwd	E		2.24	-2.5%	0.0%	**0.4%**	0.4%
fft-inv	E		2.24	-2.5%	0.0%	**0.7%**	0.4%
fma3d	F	x	1.09	5.3%	6.1%	**25.0%**	25.1%
ft	P	x	1.88	-1.9%	0.3%	**0.1%**	14.9%
g721decode	M	x	1.76	-0.3%	1.6%	**1.8%**	1.9%
g721encode	M	x	1.64	1.0%	1.1%	**1.9%**	1.9%
galgel	F		3.34	0.0%	0.0%	**0.0%**	0.0%
gap	I	x	1.27	-9.1%	0.5%	**3.4%**	4.2%
gcc-1	I	x	1.7	37.3%	35.8%	**61.5%**	61.7%
gcc-2	I	x	0.76	-9.2%	4.3%	**5.7%**	5.7%
gcc-3	I	x	0.82	-0.2%	2.7%	**4.3%**	4.1%
gcc-4	I	x	0.96	-0.2%	2.5%	**4.4%**	4.1%
ghostscript-1	E	x	1.40	17.5%	15.8%	**15.6%**	16.9%
ghostscript-2	M	x	1.66	18.0%	15.6%	**14.8%**	16.6%
ghostscript-3	E	x	1.40	12.2%	12.7%	**12.7%**	12.8%
glquake-1	G	x	1.04	2.6%	4.8%	**5.6%**	6.0%
glquake-2	G	x	1.17	2.8%	4.7%	**5.3%**	5.1%
gzip-1	I		1.79	-3.0%	0.0%	**0.8%**	0.6%
gzip-2	I		1.37	-11.1%	0.4%	**0.4%**	0.2%
gzip-3	I	x	1.49	-4.8%	1.1%	**1.5%**	1.6%
gzip-4	I		1.80	-3.7%	0.0%	**0.6%**	0.6%
gzip-5	I		1.34	-7.5%	0.3%	**0.5%**	0.8%
jpegdecode	M	x	2.21	-4.9%	1.2%	**2.3%**	2.3%
jpegencode	M	x	1.72	-0.6%	3.7%	**4.5%**	4.4%
ks-1	P		1.23	-9.9%	0.0%	**0.0%**	0.0%
ks-2	P		0.78	-10.5%	0.0%	**0.0%**	0.0%
lucas	F	x	1.05	-0.2%	0.0%	**22.7%**	22.4%
mcf	I	x	0.70	-3.8%	0.0%	**12.5%**	11.8%

Table 11: The performance of the memory dependence predictors simulated on the 'extra-large' configuration for the entire set of applications. An 'x' signifies that the benchmark is dependency sensitive (> 1% performance change between blind prediction and perfect prediction).

Benchmark Name			Base IPC	LWT	St. Sets	St. Vectors	Perfect
mesa-1	M		1.49	-0.9%	-0.1%	**0.4%**	0.7%
mesa-2	F	x	1.60	37.3%	37.5%	**40.5%**	40.3%
mesa-3	M	x	1.06	-0.3%	1.07%	**1.98%**	45.8%
mgrid	F		1.41	-0.3%	0.0%	**-0.5%**	0.0%
mpeg2decode	M		2.85	-0.1%	-0.4%	**-0.1%**	0.1%
mpeg2encode	M		3.17	-0.2%	0.0%	**-0.6%**	0.8%
parser	I	x	1.26	2.2%	7.4%	**9.1%**	8.9%
patricia	E		1.09	-0.8%	0.5%	**0.0%**	0.3%
perlbmk-1	I	x	0.95	5.9%	9.9%	**13.9%**	14.6%
perlbmk-2	I	x	1.22	-6.6%	-0.4%	**4.3%**	3.1%
perlbmk-3	I	x	2.36	-0.6%	2.2%	**7.6%**	9.1%
phylip	B	x	1.42	-1.0%	0.0%	**1.5%**	1.5%
povray-1	G	x	0.77	16.6%	20.7%	**22.0%**	22.5%
povray-2	G	x	1.02	7.3%	8.6%	**10.2%**	10.2%
povray-3	G	x	1.06	19.2%	22.8%	**24.2%**	24.62%
povray-4	G	x	1.04	11.2%	13.6%	**14.4%**	14.9%
povray-5	G	x	0.88	6.8%	9.2%	**10.2%**	10.1%
rsynth	E	x	2.59	-0.2%	0.9%	**1.7%**	1.7%
sha	E	x	2.85	12.3%	11.9%	**11.8%**	13.6%
sixtrack	F		1.38	0.0%	0.0%	**0.4%**	0.0%
susan-1	E	x	2.39	-0.5%	1.1%	**4.7%**	4.7%
susan-2	E		1.85	-0.1%	0.0%	**0.0%**	0.0%
swim	F		0.86	0.2%	0.0%	**2.5%**	0.0%
tiff2bw	E		1.52	-0.1%	0.0%	**0.8%**	0.8%
tiff2rgba	E	x	1.53	-0.2%	0.1%	**30.6%**	30.3%
tiffdither	E	x	1.59	1.6%	3.8%	**4.3%**	7.3%
tiffmedian	E		2.47	-0.6%	0.0%	**0.5%**	0.4%
twolf	I	x	0.86	-0.7%	1.4%	**3.1%**	1.3%
unepic	M		0.40	0.0%	0.0%	**1.27%**	0.0%
vortex-1	I	x	1.53	30.7%	30.3%	**42.9%**	46.3%
vortex-2	I	x	1.08	31.0%	33.4%	**45.0%**	46.3%
vortex-3	I	x	1.42	34.8%	36.6%	**47.9%**	55.2%
vpr-1	I	x	1.08	1.9%	5.4%	**6.3%**	6.1%
vpr-2	I	x	0.66	-0.8%	3.0%	**16.9%**	16.3%
wupwise	F	x	2.45	0.1%	0.4%	**11.3%**	12.1%
x11quake-1	G	x	1.58	2.6%	5.3%	**6.2%**	6.4%
x11quake-2	G		2.89	-0.4%	0.3%	**-2.5%**	-2.7%
x11quake-3	G	x	1.57	2.6%	5.4%	**6.4%**	6.6%
xanim-1	G		3.27	-0.82%	0.0%	**0.1%**	0.2%
xanim-2	G	x	2.86	0.9%	1.3%	**1.7%**	1.6%
xdoom	G	x	2.07	4.4%	5.3%	**6.4%**	10.7%
yacr2	P	x	2.14	0.1%	1.7%	**1.1%**	1.7%

Benchmark Group			Base IPC	LWT	St. Sets	St. Vectors	Perfect
ALL			1.51	2.2%	4.7%	**7.7%**	8.8%
SpecINT	I		1.22	2.8%	7.2%	**11.4%**	12.7%
SpecFP	F		1.51	1.5%	1.3%	**8.1%**	8.4%
MediaBench	M		1.72	1.7%	3.9%	**4.4%**	7.2%
MiBench	E		1.92	0.9%	3.0%	**5.3%**	5.7%
Graphics	G		1.48	5.5%	7.5%	**8.1%**	8.6%
PtrDist	P		1.49	-1.9%	2.9%	**5.31%**	6.6%
BioBench	B		2.17	-0.5%	0.2%	**0.9%**	0.9%
Dep. Sensitive		x	1.35	6.7%	9.2%	**11.8%**	13.0%

Table 12: Continuation of the data from Table 11.

REFERENCES

[1] Albayraktaroglu, K., Jaleel, A., Wu, X., Franklin, M., Jacob, B., Tseng, C.-W., and Yeung, D., "BioBench: A Benchmark Suite of Bioinformatics Applications," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, (Austin, TX, USA), pp. 2–9, March 2005.

[2] Antonio Gonzalez, J. G. and Valero, M., "Virtual-Physical Registers," in *Proceedings of the 4th International Symposium on High Performance Computer Architecture*, (Las Vegas, NV, USA), pp. 195–205, January 1998.

[3] Austin, T., Larson, E., and Ernst, D., "SimpleScalar: An Infrastructure for Computer System Modeling," *IEEE Micro Magazine*, pp. 59–67, February 2002.

[4] Austin, T. M., Breach, S. E., and Sohi, G. S., "Efficient Detection of All Pointer and Array Access Errors," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, (Orlando, FL, USA), pp. 290–301, June 1994.

[5] Bader, D. A., Li, Y., Li, T., and Sachdeva, V., "BioPerf: A Benchmark Suite to Evaluate High-Performance Computer Architecture of Bioinformatics Applications," in *Proceedings of the IEEE International Symposium on Workload Characterization*, (Austin, TX, USA), pp. 163–173, October 2005.

[6] Bahar, R. I., Albera, G., and Manne, S., "Power and Performance Tradeoffs Using Various Caching Strategies," in *Proceedings of the International Symposium on Low Power Electronics and Design*, (Monterey, CA, USA), pp. 64–69, August 1998.

[7] Baniasadi, A. and Moshovos, A., "Asymmetic-Frequency Clustering: A Power-Aware Back-end for High-Performance Processors," in *Proceedings of the International Symposium on Low Power Electronics and Design*, (Monterey, CA, USA), pp. 255–258, August 2002.

[8] Baugh, L. and Zilles, C., "Decomposing the Load-Store Queue by Function for Power Reduction and Scalability," in *Proceedings of the IBM P=AC^2 Conference*, (Yorktown Heights, NY, USA), October 2004.

[9] Bell, G. B. and Lipasti, M. H., "Deconstructing Commit," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, pp. 68–77, 2004.

[10] Brooks, D., Tiwari, V., and Martonosi, M., "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations," in *Proceedings of the 27th International Symposium on Computer Architecture*, (Vancouver, Canada), pp. 83–94, June 2000.

[11] Brown, M. D., Stark, J., and Patt, Y. N., "Select-Free Instruction Scheduling Logic," in *Proceedings of the 34th International Symposium on Microarchitecture*, (Austin, TX, USA), pp. 204–213, December 2001.

[12] Buyuktosunoglu, A., El-Moursy, A., and Albonesi, D., "An Oldest-First Selection Logic Implementation for Non-Compacting Issue Queues," in *Proceedings of the 15th International ASIC Conference*, (Rochester, NY, USA), pp. 31–35, September 2002.

[13] Cain, H. W. and Lipasti, M. H., "Memory Ordering: A Value-Based Approach," in *Proceedings of the 31st International Symposium on Computer Architecture*, (München, Germany), pp. 90–101, June 2004.

[14] Calder, B., Reinmann, G., and Tullsen, D., "Selective Value Prediction," in *Proceedings of the 26th International Symposium on Computer Architecture*, (Atlanta, GA, USA), pp. 64–74, June 1999.

[15] Castro, F., Pinuel, L., Chaver, D., Prieto, M., Huang, M., and Tirado, F., "DMDC: Delayed Memory Dependence Checking through Age-Based Filtering," in *Proceedings of the 39th International Symposium on Microarchitecture*, (Orlando, FL), pp. 297–306, December 2006.

[16] Cazorla, F. J., Ramierz, A., Valero, M., and Fernández, E., "DCache Warn: an I-Fetch Policy to Increase SMT Efficiency," in *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, (Santa Fe, NM), pp. 74–83, April 2004.

[17] Chrysos, G. Z. and Emer, J. S., "Memory Dependence Prediction Using Store Sets," in *Proceedings of the 25th International Symposium on Computer Architecture*, (Barcelona, Spain), pp. 142–153, June 1998.

[18] Cristal, A., Santana, O. J., Valero, M., and Martínez, J. F., "Toward Kilo-Instruction Processors," *ACM Transactions on Architecture and Code Optimization*, vol. 1, pp. 389–417, December 2004.

[19] Doweck, J., "Inside Intel Core Microarchitecture and Smart Memory Access," white paper, Intel Corporation, 2006. http://download.intel.com/technology/architecture/sma.pdf.

[20] El-Moursy, A. and Albonesi, D., "Front-End Policies for Improved Issue Efficiency in SMT Processors," in *Proceedings of the 9th International Symposium on High Performance Computer Architecture*, (Anaheim, CA, USA), pp. 185–196, February 2003.

[21] Eyerman, S. and Eeckhout, L., "A Memory-Level Parallelism Aware Fetch Policy for SMT Processors," in *Proceedings of the 13th International Symposium on High Performance Computer Architecture*, (Phoenix, AZ, USA), pp. 27–36, February 2007.

[22] Fields, B., Bodík, R., and Hill, M. D., "Slack: Maximizing Performance Under Technological Constraints," in *Proceedings of the 29th International Symposium on Computer Architecture*, (Anchorage, AK, USA), pp. 47–58, May 2002.

[23] Fields, B., Rubin, S., and Bodík, R., "Focusing Processor Policies via Critical-Path Prediction," in *Proceedings of the 28th International Symposium on Computer Architecture*, (Göteborg, Sweden), pp. 74–85, June 2001.

[24] Fisk, B. R. and Bahar, R. I., "The Non-Critical Buffer: Using Load Latency Tolerance to Improve data Cache Efficiency," in *Proceedings of the International Conference on Computer Design*, (Austin, TX, USA), pp. 538–545, October 1999.

[25] Fritts, J. E., Steiling, F. W., and Tucek, J. A., "MediaBench II Video: Expediting the Next Generation of Video Systems Research," *Embedded Processors for Multimedia and Communications II, Proceedings of the SPIE*, vol. 5683, pp. 79–93, March 2005.

[26] Gandhi, A., Akkary, H., Rajwar, R., Srinivasan, S. T., and Lai, K., "Scalable Load and Store Processing in Latency Tolerant Processors," in *Proceedings of the 32nd International Symposium on Computer Architecture*, (Madison, WI, USA), pp. 446–457, June 2005.

[27] Gochman, S., Ronen, R., Anati, I., Berkovitz, A., Kurts, T., Naveh, A., Saeed, A., Sperber, Z., and Valentine, R. C., "The Intel Pentium M Processor: Microarchitecture and Performance," *Intel Technology Journal*, vol. 7, May 2003.

[28] Goshima, M., Nishino, K., Nakashima, Y., ichiro Mori, S., Kitamura, T., and Tomita, S., "A High-Speed Dynamic Instruction Scheduling Scheme for Superscalar Processors," in *Proceedings of the 34th International Symposium on Microarchitecture*, (Austin, TX, USA), pp. 225–236, December 2001.

[29] Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B., "MiBench: A Free, Commerically Representative Embedded Benchmark Suite," in *Proceedings of the 4th Workshop on Workload Characterization*, (Austin, TX, USA), pp. 83–94, December 2001.

[30] Hamerly, G., Perelman, E., Lau, J., and Calder, B., "SimPoint 3.0: Faster and More Flexible Program Analysis," in *Proceedings of the Workshop on Modeling, Benchmarking and Simulation*, (Madison, WI, USA), June 2005.

[31] Hill, M. D. and Marty, M. R., "Amdahl's Law in the Multicore Era," *IEEE Computer*, 2008.

[32] Hinton, G., Sager, D., Upton, M., Boggs, D., Carmean, D., Kyler, A., and Roussel, P., "The Microarchitecture of the Pentium 4 Processor," *Intel Technology Journal*, Q1 2001.

[33] Intel, "First the Tick, Now the Tock: Next Generation Intel Microarchitecture(Nehalem)," white paper, Intel Corporation, 2008. http://www.intel.com/technology/architecture-silicon/next-gen/whitepaper.pdf.

[34] Jacobson, E., Rotenberg, E., and Smith, J. E., "Assigning Confidence to Conditional Branch Predictions," in *Proceedings of the 29th International Symposium on Microarchitecture*, (Paris, France), pp. 142–152, December 1996.

[35] Kessler, R. E., "The Alpha 21264 Microprocessor," *IEEE Micro Magazine*, vol. 19, pp. 24–36, March–April 1999.

[36] Kun Luo, Jayanth Gummaraju, M. F., "Balancing throughput and fairness in SMT processors," in *Proceedings of the 15th International Symposium on Performance Analysis of Systems and Software*, (Tucson, AZ, USA), pp. 9–16, November 2001.

[37] Kun Luo, Manoj Franklin, S. M. and Seznec, A., "Boosting SMT Performance by Speculation Control," in *Proceedings of the 15th International Parallel and Distributed Processing Symposium*, pp. 9–16, 2001.

[38] Larson, E., Chatterjee, S., and Austin, T., "MASE: A Novel Infrastructure for Detailed Microarchitectural Modeling," in *Proceedings of the 2001 International Symposium on Performance Analysis of Systems and Software*, (Tucson, AZ, USA), pp. 1–9, November 2001.

[39] Lee, C., Potkonjak, M., and Mangione-Smith, W. H., "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communication Systems," in *Proceedings of the 30th International Symposium on Microarchitecture*, (Research Triangle Park, NC, USA), pp. 330–335, December 1997.

[40] Lepak, K. M. and Lipasti, M. H., "Silent Stores for Free," in *Proceedings of the 33rd International Symposium on Microarchitecture*, (Monterey, CA, USA), pp. 22–31, December 2000.

[41] Limousin, C., Sebot, J., Vartanian, A., and Drach-Temam, N., "Improving 3D Geometry Transformations on a Simultaneous Multithreaded SIMD Processor," in *Proceedings of the International Conference on Supercomputing*, (Sorrento, Italy), pp. 236–245, June 2001.

[42] Lipasti, M. H., "Value Prediction: Are(n't) We Done Yet?," in *Proceedings of the 2nd Value Prediction Workshop*, (Boston, MA, USA), October 2004.

[43] Loh, G. H., Sami, R., and Friendly, D. H., "Memory Bypassing: Not Worth the Effort," in *Proceedings of the 1st Workshop on Duplicating, Deconstructing, and Debunking*, (Anchorage, AK, USA), pp. 71–80, May 2002.

[44] Marculescu, D., "Application Adaptive Energy Efficient Clustered Architectures," in *Proceedings of the International Symposium on Low Power Electronics and Design*, (Newport Beach, CA, USA), pp. 338–343, August 2004.

[45] Marr, D. T., Binns, F., Hill, D. L., Hinton, G., Koufaty, D. A., Miller, J. A., and Upton, M., "Hyper-Threading Technology and Microarchitecture," *Intel Technology Journal*, vol. 6, February 2002.

[46] Martinez, J. F., Renau, J., Huang, M. C., Prvulovic, M., and Torrellas, J., "Cherry: Checkpointed early resource recycling in out-of-order microprocessors," in *Proceedings of the 35th International Symposium on Microarchitecture*, (Istanbul, Turkey), pp. 27–36, November 2002.

[47] McFarling, S., "Cache Replacement with Dynamic Exclusion," in *Proceedings of the 18th International Symposium on Computer Architecture*, (Toronto, Canada), pp. 191–200, May 1991.

[48] McFarling, S., "Combining Branch Predictors," TN 36, Compaq Computer Corporation Western Research Laboratory, June 1993.

[49] McFarling, S., "Branch Predictor with Serially Connected Predictor Stages for Improving Branch Prediction Accuracy." USPTO Patent 6,374,349, April 2002.

[50] Moshovos, A., *Memory Dependence Prediction*. PhD thesis, University of Wisconsin, 1998.

[51] Moshovos, A., Breach, S. E., Vijaykumar, T. N., and Sohi, G. S., "Dynamic Speculation and Synchronization of Data Dependences," in *Proceedings of the 24th International Symposium on Computer Architecture*, (Boulder, CO, USA), pp. 181–193, June 1997.

[52] Moshovos, A. and Sohi, G. S., "Streamlining Inter-operation Memory Communication via Data Dependence Prediction," in *Proceedings of the 30th International Symposium on Microarchitecture*, (Research Triangle Park, NC, USA), pp. 235–245, December 1997.

[53] Moshovos, A. and Sohi, G. S., "Speculative Memory Cloaking and Bypassing," *International Journal of Parallel Programming*, October 1999.

[54] Naffziger, S., "High-Performance Processors in a Power-Limited World," *Symposium on VLSI Circuits Digest of Technical Papers*, Q3 2006.

[55] Palacharla, S., *Complexity-Effective Superscalar Processors*. PhD thesis, University of Wisconsin, 1998.

[56] Perelman, E., Hamerly, G., and Calder, B., "Picking Statistically Valid and Early Simulation Points," in *Proceedings of the 2003 International Conference on Parallel Architectures and Compilation Techniques*, (New Orleans, LA, USA), pp. 244–255, September 2004.

[57] Qureshi, M. K., Jaleel, A., Patt, Y. N., Jr., S. C. S., and Emer, J., "Adaptive Insertion Policies for High-Performance Caching," in *Proceedings of the 34th International Symposium on Computer Architecture*, (San Diego, CA, USA), June 2007.

[58] Roth, A., "A High Bandwidth Low Latency Load/Store Unit for Single- and Multi-Threaded Processors," MS-CIS 04-09, University of Pennsylvania, June 2004.

[59] Roth, A., "Store Vulnerability Window (SVW): Re-Execution Filtering for Enhanced Load Optimization," in *Proceedings of the 32nd International Symposium on Computer Architecture*, (Madison, WI, USA), pp. 458–468, June 2005.

[60] Salverda, P. and Zilles, C. B., "A Criticality Analysis of Clustering in Superscalar Processors," in *Proceedings of the 38th International Symposium on Microarchitecture*, (Barcelona, Spain), pp. 55–66, November 2005.

[61] Sankaralingam, K., Nagarajan, R., Liu, H., Kim, C., Huh, J., Burger, D., Keckler, S. W., and Moore, C. R., "Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture," in *Proceedings of the 30th International Symposium on Computer Architecture*, (San Diego, CA, USA), pp. 422–433, May 2003.

[62] Sassone, P. G., Rupley, J., Brekelbaum, E., Loh, G. H., and Black, B., "Matrix Scheduler Reloaded," in *Proceedings of the 34th International Symposium on Computer Architecture*, (San Diego, CA, USA), pp. 335–346, June 2007.

[63] Sato, T. and Arita, I., "Revisiting Direct Tag Seach Algorithm on Superscalar Processors," in *Proceedings of the Workshop on Complexity-Effective Design*, (Göteborg, Sweden), June 2001.

[64] Seng, J. S., Tune, E., and Tullsen, D. M., "Reducing Power with Dynamic Critical Path Information," in *Proceedings of the 34th International Symposium on Microarchitecture*, (Austin, TX, USA), pp. 114–123, December 2001.

[65] Sethumadhavan, S., Desikan, R., Burger, D., Moore, C. R., and Keckler, S. W., "Scalable Hardware Memory Disambiguation for High ILP Processors," in *Proceedings of the 36th International Symposium on Microarchitecture*, (San Diego, CA, USA), pp. 118–127, May 2003.

[66] Seznec, A., Felix, S., Krishnan, V., and Sazeides, Y., "Design Tradeoffs for the Alpha EV8 Conditional Branch Predictor," in *Proceedings of the 29th International Symposium on Computer Architecture*, (Anchorage, AK, USA), May 2002.

[67] Sha, T., Martin, M. M. K., and Roth, A., "Scalable Store-Load Forwarding via Store Queue Index Prediction," in *Proceedings of the 38th International Symposium on Microarchitecture*, (Barcelona, Spain), pp. 159–170, November 2005.

[68] Sha, T., Martin, M. M. K., and Roth, A., "NoSQ: Store-Load Forwarding without a Store Queue," in *Proceedings of the 39th International Symposium on Microarchitecture*, (Orlando, FL), pp. 285–296, December 2006.

[69] Sharkey, J., "M-Sim: A Flexible, Multithreaded Architectural Simualtion Environment," CS-TR 05-DP01, State Univeristy of New York at Binghamton, Department of Computer Science, 2005.

[70] Sharkey, J., Balkan, D., and Ponomarev, D., "Adaptive Reorder Buffers for SMT Processors," in *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, pp. 244–253, 2006.

[71] Sharkey, J. and Ponomarev, D., "Efficient Instruction Schedulers for SMT Processors," in *Proceedings of the 12th International Symposium on High Performance Computer Architecture*, (Austin, TX, USA), pp. 293–303, February 2006.

[72] Shen, J. P. and Lipasti, M. H., *Modern Processor Design: Fundamentals of Superscalar Processors*. McGraw Hill, 2005.

[73] Srikanth T. Srinivasan, Roy Dz-ching Ju, A. R. L. and Wilkerson, C., "Locality vs. Criticality," in *Proceedings of the 28th International Symposium on Computer Architecture*, (Göteborg, Sweden), pp. 47–58, June 2001.

[74] Srinivasan, S. T. and Lebeck, A. R., "Load Latency Tolerance in Dynamically Scheduler Processors," in *Proceedings of the 31st International Symposium on Microarchitecture*, (Dallas, TX, USA), pp. 148–159, November 1998.

[75] Srinivasan, S. T., Rajwar, R., Akkary, H., Gandhi, A., and Upton, M., "Continual Flow Pipelines," in *Proceedings of the 11th Symposium on Architectural Support for Programming Languages and Operating Systems*, (Boston, MA, USA), pp. 107–119, October 2004.

[76] Stone, S. S., Woley, K. M., and Frank, M. I., "Address-Indexed Memory Disambiguation and Store-to-Load Forwarding," in *Proceedings of the 37th International Symposium on Microarchitecture*, (Barcelona, Spain), pp. 171–182, November 2005.

[77] Subramaniam, S., Bracy, A., Wang, H., and Loh, G. H., "Criticality-Based Optimizations for Efficient Load Processing," in *Proceedings of the 15th International Symposium on High Performance Computer Architecture*, (Raleigh, NC, USA), pp. 244–253, February 2009.

[78] Subramaniam, S. and Loh, G. H., "Fire-and-Forget: Load/Store Scheduling with No Store Queue At All," in *Proceedings of the 39th International Symposium on Microarchitecture*, (Orlando, FL), pp. 273–284, December 2006.

[79] Subramaniam, S. and Loh, G. H., "Store Vectors for Scalable Memory Dependence Prediction and Scheduling," in *Proceedings of the 12th International Symposium on High Performance Computer Architecture*, (Austin, TX, USA), pp. 64–75, February 2006.

[80] Subramaniam, S., Prvulovic, M., and Loh, G. H., "PEEP: Exploiting Predictability of Memory Dependences in SMT Processors," in *Proceedings of the 14th International Symposium on High Performance Computer Architecture*, (Salt Lake City, UT, USA), pp. 64–75, February 2008.

[81] Tarjan, D., Thoziyoor, S., and Jouppi, N. P., "CACTI 4.0," Tech. Rep. HPL-2006-86, HP Laboratories Palo Alto, June 2006.

[82] Tullsen, D. M. and Brown, J., "Handling Long-Latency Loads in a Simultaneous Multithreaded Processor," in *Proceedings of the 34th International Symposium on Microarchitecture*, (Austin, TX, USA), pp. 318–327, December 2001.

[83] Tullsen, D. M., Eggers, S. J., Emer, J. S., Levy, H. M., Lo, J. L., and Stamm, R. L., "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor," in *Proceedings of the 23rd International Symposium on Computer Architecture*, (Philadelphia, PA, USA), pp. 191–202, May 1996.

[84] Tune, E., Liang, D., Tullsen, D. M., and Calder, B., "Dynamic Prediction of Critical Path Instructions," in *Proceedings of the 7th International Symposium on High Performance Computer Architecture*, (Monterrey, Mexico), pp. 185–196, January 2001.

[85] Tune, E. S., Tullsen, D. M., and Calder, B., "Quantifying Instruction Criticality," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pp. 104–113, September 2002.

[86] Tyson, G. S. and Austin, T. M., "Improving the Accuracy and Performance of Memory Communication Through Renaming," in *Proceedings of the 30th International Symposium on Microarchitecture*, (Research Triangle Park, NC, USA), pp. 218–227, December 1997.

[87] Yoaz, A., Erez, M., Ronen, R., and Jourdan, S., "Speculation Techniques for Improving Load Related Instruction Scheduling," in *Proceedings of the 26th International Symposium on Computer Architecture*, (Atlanta, GA, USA), pp. 42–53, June 1999.

[88] Zhang, Y., Yang, J., and Gupta, R., "Frequent Value Locality and Value-Centric Data Cache Design," in *Proceedings of the 9th Symposium on Architectural Support for Programming Languages and Operating Systems*, (Cambridge, MA, USA), pp. 150–159, November 2000.

[89] Zilles, C. B. and Sohi, G. S., "Understanding the Backward Slices of Performance Degrading Instructions," in *Proceedings of the 27th International Symposium on Computer Architecture*, (Vancouver, Canada), pp. 172–181, June 2000.

Samantika Subramaniam is a PhD candidate in the School of Computer Science, College of Computing, at the Georgia Institute of Technology and a member of the Superscalar Technology INnovation Group (STING). Her research focus is in the impact of memory instruction processing on the efficiency of current and future processors. Professor Gabriel H. Loh is her thesis adviser. Samantika is the recipient of the Intel Foundation Fellowship for 2008-2009 and has been awarded the Outstanding Graduate Research Assistant Award for 2008 by the College of Computing at the Georgia Institute of Technology. She was a Graduate Research Intern at Intel Corporation from June 2007- December 2007.

Conference Publications:

- Samantika Subramaniam, Anne Bracy, Hong Wang, Gabriel H. Loh, Criticality-Based Optimizations for Efficient Load Processing, to appear in the Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA), February 2009, NC, USA.
- Samantika Subramaniam, Milos Prvulovic, Gabriel H. Loh, PEEP: Exploiting Predictability of Memory Dependences in SMT Processors, in the Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA), February 2008, UT, USA.
- Samantika Subramaniam, Gabriel H. Loh, Fire-and-Forget: Load/Store Scheduling with No Store Queue at All, in the Proceedings of the International Symposium on Microarchitecture (MICRO), December 2006, FL, USA.
- Samantika Subramaniam, Gabriel H. Loh, Store Vectors for Scalable Memory Dependence Prediction and Scheduling, in the Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA), February 2006, TX, USA.

Journals:

- Samantika Subramaniam and Gabriel H. Loh, Design and Optimization of the Store Vectors Memory Dependence Predictor, to appear in the ACM Transactions on Architecture and Code Optimization (Accepted with revisions).

Workshops:

- Rahul V. Garde, Samantika Subramaniam, Gabriel H. Loh, Deconstructing the Inefficacy of Global Cache Replacement Policies, in the 7th Workshop on Duplicating, Deconstructing, and Debunking (WDDD) 2008 (held in conjunction with ISCA 2008), Beijing, China.