Understanding and Designing for Dependent Store/Load Pairs in High Performance Microprocessors

by

Ravindra Nath Bhargava, B.S.E.

Thesis

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTERS OF SCIENCE IN ENGINEERING

The University of Texas at Austin

August 2000

Understanding and Designing for Dependent Store/Load Pairs in High Performance Microprocessors

Approved by
Supervising Committee:

____

____

To Lindsay, my parents, and the rest of my family.

Your support gives me the strength to do so many things.

Understanding and Designing for Dependent Store/Load Pairs in High Performance Microprocessors

Ravindra Nath Bhargava, M.S.E.
The University of Texas at Austin, 2000

Supervisor: Lizy Kurian John

General-purpose microprocessor performance is sensitive to load-store ordering, memory bandwidth, and memory access latency. Memory access latency is one of the primary bottlenecks in modern superscalar microprocessors. Reducing the frequency at which an application accesses the memory hierarchy significantly improves performance (measured in instructions per cycle) and reduces power. This thesis presents a characterization of the load and store behavior of C, C++, and Java programs compiled for the SPARC architecture. This analysis corroborates and expands on load/store dependency trends seen in other studies and instruction set architectures. For many applications, 80-90% of stored values are transient, i.e. local to the application. Additionally, a store to one address is often closely followed by a load or store to the same address.

The rest of the thesis focuses on reducing the impact of these transient memory requests. A store buffer exists in many current processors to accomplish one or more of the following: store access ordering, latency hiding, and data forwarding. Various store buffer issues are analyzed and the effect on performance is established. A store entry removal policy along with store buffer pipeline placement have significant impact on the overall performance of a microprocessor. This thesis illustrates how a well-designed store buffer achieves comparable performance to a larger buffer.

An alternative to forwarding data through the store buffer is to create a separate forwarding queue called the *Virtual Store Queue (VSQ).* The basic VSQ requires no value or memory address speculation. Using limited resources, the VSQ provides a significant performance increase. High-level studies show that a small eight-entry FIFO VSQ has the potential to eliminate 20.7% of load traffic, while a 32-entry LRU VSQ can eliminate 33.3% of load accesses on average. Using detailed simulation, the average reduction in load traffic to the L1 data cache is found to be 27.1%, and the average reduction in port utilization is 16.9%. The VSQ is show to be a more effective forwarding mechanism than the store buffer.

Finally, this thesis investigates the potential of the VSQ in a future, 16 instruction wide microprocessor. Simulations indicate that the VSQ will provide an overall performance increase of 7.2% for the benchmark suite studied. This performance can be enhanced using *load tokens.* These tokens, which are stored in the trace cache, orchestrate speculative data acquisition from the VSQ. The ability to satisfy a load early in the pipeline can provide an additional overall performance boost of 3.2%, leading to an overall performance improvement of 10.4% over the base model.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

The difference between the frequency of microprocessor cores and the speed at which memory is accessed is increasing steadily [7, 21, 37, 44]. This memory gap leads to an increased latency for off-chip memory accesses. Even the access latency for on-chip, first-level, and second-level caches has become less tolerable. High clock frequencies and deeply pipelined machines result in multiple-cycle cache accesses [1, 31]. Therefore, memory access operations consume more time and resources than register to register operations, even when the requested piece of data is located in the level-one cache.

Techniques exist to hide portions of the load and store latencies. Executing instructions in an out-of-order manner allows useful work to occur in parallel with memory accesses. Address translations are often done simultaneously with other operations. Compilers assist the processor by performing load hoisting, placing a load instruction well before its first dependent instruction. Prefetching data and instructions into the caches is an approach implemented in both hardware and software to reduce the high-latency off-chip memory accesses [9, 19, 28, 46]. Even though these and other techniques exist, memory access latency is still considered one of the primary bottlenecks in processors today.

Memory latency is a performance issue only if the memory is accessed repeatedly. Ideally, a single-process program should load values from a memory address only once as needed during the life of the program. Similarly, a store to a memory address should only occur once, after the output of the program has been computed. Despite the increase of on-chip resources and techniques like register renaming, reorder buffers, heavy-duty compiler optimizations, and load forwarding, it is not unusual for the dynamic instruction streams of modern workloads to consist of 25% to 40% memory instructions.

Many of the load and store instructions are unnecessary. Reducing the number of memory operations can both improve performance and reduce power by eliminating off-chip memory accesses. The abundance of spurious memory operations is attributed to a lack of resources. Specifically, the small number of available architectural registers requires data to be temporarily stored in memory (register spills). Additionally, the compiler often does not have enough information at compile time to determine if two variables are truly unique due to ambiguous memory references. Stacks designated in the instruction set architecture also lead to spurious memory accesses. For example, Java and x86 programs refer to their architected stacks using memory references. If it is possible to capture these types of transient values without the high cost associated with load and store instructions, then program performance can be improved.

1.2 Solutions

Due to the memory access bottleneck, almost all modern processors allow dynamic ordering of load and store instructions. Some combination of non-blocking caches and buffering structures such as write buffers, store buffers, store queues, miss status handling registers, and load queues is typically employed. Usually, there are several pieces to the memory management puzzle. Although the performance of a processor can be sensitive to the policies and implementations of these dynamic memory access structures, the strategies for these structures have not been thoroughly investigated in the available literature.

This thesis studies three methods of reducing the negative impact of memory accesses for transient values. The first two methods are low-resource methods that fit into modern microprocessor architectures. The last method is targeted for future wide-issue microprocessors.

- The first method leverages existing store buffers to increase the load forwarding opportunities and reduce the latency of many memory reads. The design policies of the store buffer have not been analyzed thoroughly in the literature. This work covers a large portion of the store buffer design space, determines the best policies, and presents the design tradeoffs.

- Another design presented for modern processors is the *Virtual Store Queue (VSQ)*. The VSQ is a dedicated forwarding buffer for transient values. This new structure is not subject to the same constraints as a store buffer, and can be fine-tuned for this specific purpose. The buffer is indexed using virtual addresses, often eliminating the physical address translation step for load instructions.

- The final proposal applies to wide-issue microprocessors equipped with a trace cache. *Load tokens* are introduced to orchestrate speculative data acquisition from the VSQ. Each token is associated with a load instruction and stored in the trace cache between invocations of the instruction. When the speculation is correct, load instructions are completed even earlier in the memory pipeline.

1.3 Thesis Statement

Frequent load and store instructions are a performance bottleneck in modern dynamically scheduled microprocessors. Many of the data reads and writes to memory are not necessary for the correct functionality of a program. The store buffer can be redesigned to decrease the load traffic to the memory system while improving overall performance. A virtual store queue can be implemented for the exclusive purpose of forwarding data, reducing the burden on the store buffer and further improving performance. For future wide-issue microprocessors, more performance can be reclaimed if the trace cache manages load tokens that enable accurate, low-latency, speculative load forwarding.

1.4 Contributions

This thesis makes several contributions:

- The relationship between load and store instructions is characterized in detail. This work corroborates and expands on previous memory dependence studies. This analysis includes analysis of dynamic memory characteristics such as port utilization, load forwarding, and store buffer occupancy. An understanding of the behavior of same address memory operations is also developed.

- The mechanisms discussed in this thesis are evaluated using a modern workload that is more diverse than comparable studies in the this area. The self-developed simulation and experiment framework allows for detailed cycle-level performance studies of Java and C++ workloads as well as the popular SPEC integer C benchmarks.

- For the first time in published literature, store buffer design policies are identified, defined, and thoroughly analyzed. The analysis shows that a lazy store removal scheme can have a large impact on processor performance. Smaller, well-designed store buffers can achieve comparable performance to larger, basic store buffers.

- The Virtual Store Queue (VSQ) is described and evaluated. The VSQ provides marginal performance impact for a four-wide machine, but has an overall performance increase of 7.2% for a 16-wide microprocessor. In the four-wide configuration, the average reduction in load traffic to the L1 data cache is 27.1%, and the average reduction in port utilization is 16.9%.

- For future wide-issue microprocessors, this thesis presents a VSQ enhancement to reduce the impact of memory operations. Load tokens are stored in the trace cache, allowing accurate, speculative accesses to the VSQ. This speculation increases the overall performance gain with a VSQ to 10.4% for a 16-wide microprocessor configuration.

1.5 Organization

This thesis begins with the current chapter (Chapter 1) which is an introduction to the thesis problem, solutions, and contributions. The second chapter discusses some of the store buffer design points and the proposed microarchitecture mechanism, the VSQ. Chapter 3 describes the experimental framework, methodology, and benchmarks. Chapter 4 characterizes the memory operations in the studied benchmarks. The relationship between load and store instructions is explored in detail. The performance analysis of the store buffer policies and VSQ is presented in Chapter 5. Chapter 6 proposes load tokens, an enhancement to the VSQ for next-generation microprocessors. The performance of the VSQ and load tokens are analyzed in a wide-issue microprocessor environment. In Chapter 7, other works in memory dependency analysis, data speculation, and store buffers are discussed.

Chapter 8 concludes the thesis.

Chapter 2

Designs to Improve Load Forwarding

2.1 Store Buffer Design Policies

One under-analyzed structure within memory access management is the store buffer. The store buffer is the mechanism within the microprocessor core that handles the tasks of dynamically tracking stores, maintaining their order, and properly synchronizing them with other memory accesses. This synchronization includes handling disambiguation issues, releasing stores to the memory system, and performing load forwarding. To maintain all these properties, each store is required to have its own entry in the store buffer throughout the life of the store.

The manner in which stores are handled within the store buffer can impact the performance of processors. In the absence of adequate literature discussing this impact, this thesis examines several store buffer issues, including *size*, *store removal*, *store retirement point*, *store priority shifting*, and *virtual store buffers*. A thorough study of these policies shows a potential for increased load forwarding and decreased load latency with changes in policy [3]. The challenge is to achieve the memory related performance gains without decreasing overall performance (e.g. stalling the pipeline because of a saturated store buffer).

2.1.1 Store Retirement Point

The retirement point of a store refers to the point in the life of a store instruction when it *attempts* to write its data in memory. Most processors do not allow this to happen until all previous instructions are complete. This helps insure that the stores will be sent to memory in order and eases the exception handling process.

Once a store instruction has its address calculated, has its data, becomes non-speculative, and is situated in the store buffer, the store may retire. However, there is no need to retire a store as soon as it is ready. Instead, store instructions may accumulate in store buffer entries and retire when a certain level of active store entries[1] is reached. This is similar to the concept of a *highwater mark* in write buffer design. Delaying retirement increases the opportunity for forwarding. However, it can lead to more store buffer stalls. The interaction with loads and the effects on the L1 cache hit rates determine the effectiveness of this policy.

2.1.2 Store Removal

Once a store has retired, it may be removed from the buffer, but this is not always necessary. It might be beneficial to keep the store data active in the store buffer for the purpose of load forwarding. The act of removing the active store buffer entry from the store buffer will be referred to as *store removal.* Once removed, the store buffer entry can no longer be accessed by any loads, and one additional empty slot is available in the store buffer.

One positive effect of this policy is an increase in the average occupancy of the store buffer and therefore an increase in load forwarding. One problem with having such a "lazy" removal policy is the potential of filling the store buffer. It is necessary to indicate with tags which stores have retired but are still active in

[1] *active entry* refers to any entry in the store buffer that is in use and currently contains a store in any state

the store buffer. Another problem with building up active entries is the possible increase in disambiguation time.

2.1.3 Store Priority Switching

Cache contention arises when there are multiple available memory instructions and a finite amount of ports or gateways into the first level of memory. Typically among loads, the oldest load that can access memory (i.e. address sufficiently calculated) has the highest priority. Among stores, only the oldest store is permitted to access the memory.

In a semi-aggressive memory model, loads can access memory out-of-order with respect to other loads, loads can bypass older stores with some restrictions, and stores are processed in-order. Typically among loads, the oldest load that is ready to access memory (i.e. address sufficiently calculated,) has the highest priority. Among stores, only the oldest store is permitted to access the memory given that it is non-speculative, has its data, and the effective address is sufficiently calculated.

The policy for selecting among the stores and among the loads is clear, but what happens if both a load and a store are ready to access memory in some given clock cycle? There are three options: i) the load is given a higher priority, or *loads first*, ii) the store is given priority, or *stores first*, or iii) the oldest instruction is given priority, or *oldest first*.

Load instructions retrieve data for upcoming instructions. Therefore, choice one is more attractive than choice two. If load forwarding exists, the time at which stores submit their data to the cache is of less importance to the execution of subsequent instructions. The third choice is a safe choice that would increase fairness and decrease the chance of store buffer stalls (i.e. not being able to issue a store due to a full store buffer).

There is also the option to change the priority scheme dynamically as implemented in the UltraSPARC-IIi. For instance, the default priority could be *loads first*. Once the level is equal to or above some threshold level, the policy then switches to *stores first*. The desired effect is to reduce the number of store buffer stalls and therefore improve the performance of the processor.

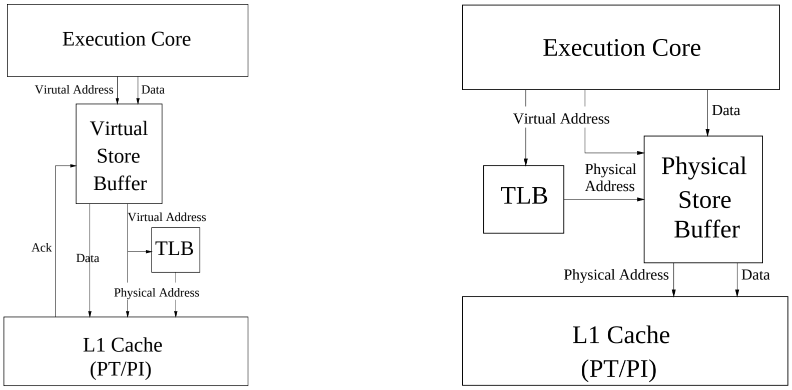
2.1.4 Virtual Store Buffer

Conceptually, store buffers may be accessed by loads before or after the address translation stage in processors. A *virtual store buffer* is accessed before translation using a virtual address. Therefore, to access a virtual store buffer no address translation is required. A load can receive data from the store buffer without having its address translated.

In a virtual store buffer, the data and virtual address of a store are placed in the previously allotted store buffer entry. A load that has its virtual address calculated may access the store buffer and receive the data from a store with the same address, if the data is available. A load can receive data from the store buffer without having its address translated, saving the address translation cycles. The address would have been translated later, either prior to the cache access or in parallel with the cache access. There are some other issues that must be addressed when using virtual addresses. These are discussed in Section 6.

In the case of a *physical store buffer*, both the load and the store must have their addresses translated before a load can properly access the data available in the store buffer. In this design, it is beneficial to have address translation take place as soon as possible to eliminate a translation bottleneck near the cache access point. The design complexity of using a physical store buffer is reduced because physical addresses are guaranteed to uniquely identify a piece of data.

The differences between the virtual and physical store buffer are illustrated in Figure 2.1. In this figure and throughout the simulations, a physically indexed



a) Virtual Store Buffer

b) Physical Store Buffer

Figure 2.1: Comparison of a Virtual Store Buffer and Physical Store Buffer
PT/PI = Physically tagged, physically indexed.

and physically tagged cache is assumed. For a discussion of how the terms virtual and physical apply to store buffers in the case of the virtually indexed, physically tagged cache, see Section 6. The essential difference is that an entry in the virtual store buffer is considered ready before address translation.

2.2 Virtual Store Queue

Instead of burdening the store buffer with data forwarding requirements as well as memory ordering duties, a separate structure can be tailored for the forwarding task. A *virtual store queue (VSQ)* is such a structures. The VSQ provides temporary storage for the data associated with store instructions.This allows store instructions to leave data in a fast, dedicated forwarding buffer for future load instructions. Load instructions may then obtain data from the VSQ early on in the pipeline, avoiding a large portion of the memory pipeline.

The sole purpose of the VSQ is to provide efficient forwarding of data. On the other hand, a store buffer is responsible for maintaining the relative order of inflight stores and providing temporary storage until the store request can be released to the data cache. More specifically, the virtual store queue differs conceptually from a typical store buffer in that it i) holds virtual addresses, ii) is not required to maintain program order of the stores, iii) may hold data even after the store has left the store buffer, iv) does not need to contain every store, and v) does not stall the processor when it is full.

The VSQ is an elegant and low-cost method for exploiting transient values and attacking the memory bottleneck. It does not require speculation like previously proposed memory dependence mechanisms. However, the VSQ still provides performance increases that rival these schemes in a modern microprocessor environment.

The VSQ adds and removes entries in a queue fashion, but is read and updated in a content addressable and associative manner, much like a reorder buffer. Each entry contains a tag, a virtual address, a valid bit, information about the operands, and the data. Data may be forwarded to load instructions that query the VSQ. A matching address results in a *VSQ hit*, allowing a load to access its data from the VSQ. This load instruction skips address disambiguation within the store buffer, address translation and data cache access.

Figure 2.2 depicts a portion of the instruction life for loads and stores. Loads and stores are the only instructions with direct interaction with the VSQ. All stores are placed in the VSQ after decode and then sent through the machine as usual, including effective address calculation, address translation, waiting in the store buffer, and dispatch to the memory system. A store will be left in the VSQ after it has been retired from the store buffer and/or reorder buffer (ROB).

Loads access the VSQ at two points as they flow through the machine. The first point is immediately after decode, an *early query*. Although the address of the load has not been calculated, it is possible to match the addresses successfully. The

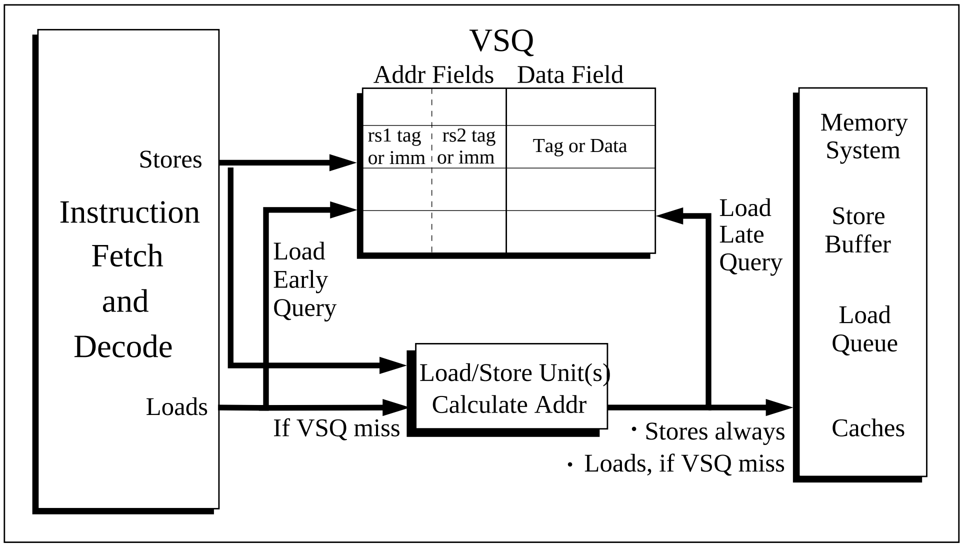


Figure 2.2: Virtual Store Queue (VSQ)

operands of a store instruction are stored in the VSQ in the form of an immediate, a value, or as a data tag. If these fields in the VSQ match the corresponding values and tags for the load instruction, then the load instruction has the same effective address as the store instruction. In this case, the data value or tag from the VSQ entry is forwarded to the load. The load no longer has to calculate its effective address or access the memory hierarchy.

If the load operation can not be completed early due to a lack of information, it will access the VSQ upon calculating its address. At this point, the load once again queries the VSQ. If a store with the correct relative age (oldest instruction younger than the load) has a matching address, then the load is basically complete at this point and is not sent to the memory controller and load queue. Another possible implementation is for the loads to continuously query the store buffer during every pipeline stage. This option is not pursued due to the hardware complexity.

The VSQ is similar to a store buffer, specifically the virtual store buffer presented earlier. The VSQ studied in this thesis is implemented as a FIFO queue.

However, this it not necessary. As shown in Chapter 5 the VSQ produces higher hit rates using the least-recently used replacement policy and is open to other schemes that may improve its efficiency. The store buffer traditionally maintains the ordering of store instructions and issues them correctly to memory. Therefore, it does not have the flexibility to be a high performance forwarding device. In addition, abundant store data in the store buffer leads to processor stalls instead of improved instruction throughput.

Another differentiating factor between a store buffer and VSQ is that store addresses and the corresponding values may be maintained as long as the replacement policy and resources dictate. If it is possible to identify multiple-load store addresses, these VSQ entries can remain in the VSQ longer than less-frequently accessed store data. Once again, the store buffer is restricted by the requirements to retire stores sequentially and to remain partially empty. Consistency also prevents the store buffer from killing stores with similar addresses as can be done in the VSQ.

2.3 Issues for the Store Buffer and VSQ

This section confronts general issues shared by both the store buffer and VSQ. Virtual address indexing, precise exceptions, misspeculation, and cache configuration are discussed. Additional issues, such as coherence, varying data sizes, and non-cacheable memory accesses must be handled as well. Snooping, tags, and detection/recovery, respectively, are possible solutions.

2.3.1 Accessing with Virtual Indexes

Aliasing is an important issue when using virtual addresses to tag and access the data. Aliasing results when two different virtual addresses map to the same physical address. This can lead to consistency problems. For example, if there is a store to virtual address `A` and a load from virtual address `B`, the virtual store buffer and VSQ would not forward the data of `A` to `B`. Unless aliasing exists, this is normally correct behavior. With aliasing, the load from `B` will receive incorrect data from elsewhere in the forwarding mechanism or from the memory system.

A different issue arises when two instances of the same virtual address map to different physical addresses. In this case, a virtually tagged and indexed structure may incorrectly forward data to a load. Sections of code with different address spaces or self-modifying code can cause this type of inconsistency.

These situations are infrequent, but they must be handled when using virtual addresses. Issues with virtual addresses have been dealt with in virtually indexed caches. Software and hardware solutions exist to guarantee correctness [11, 25, 59]. The UltraSPARC-IIi burdens the operating system with this responsibility, claiming that "software handles aliasing" with respect to its virtually indexed cache. Aliasing is common when two processes are sharing data.

Utilizing virtual addresses requires an easy method to distinguish unique processes. A typical solution is to locate a unique address space identifier or process ID. This ID can be concatenated to the virtual address to create unique tags. This is common in some of the 64-bit RISC processors, but quite a challenge in the x86 family of processors. Therefore, the Intel Pentium Pro MOB and the AMD K6 store buffer exclusively use physical addresses. Without the additional identification information, one option is to flush virtually indexed structures on context switches, for instances of self-modifying code, and on detection of multi-process data sharing. This is an expensive process and certainly deteriorates performance.

Instead, this situation can be handled as follows. When a store from one process is being placed in the VSQ or virtual store buffer, it checks the page offset bits of each address with different virtual tag bits and if any of those page offsets are equal to the page offset of the current store, then that entry is invalidated. Essentially, the store checks if stores from other address spaces could be mapped to the same page and potentially the same address. If this is discovered, the old store is eliminated. This will eliminate more stores than necessary, but such aliasing occurs extremely rarely and may not result in significant performance loss.

2.3.2 Precise Exceptions

Another issue of interest is exception handling. It is important to be able to recover from exceptions like misspeculations, interrupts, context switches, or traps. When an exception occurs, the processor wishes to be able to start over at the instruction that caused the exception without losing information. This requires that all preceding instructions complete and properly modify the processor state.

With the store buffer, this is especially critical. It is often the case that a store instruction in the store buffer is older than the oldest instruction in the ROB but has not been sent to memory. If an exception happens at this point, critical information could be lost. This is handled by flushing all the unreleased stores (older than the instruction that cause the exception) to memory. In the above scenario, when the exception returns, the store buffer is empty. A load that could have obtained a value from the store buffer will be forced to request the data from memory instead. The VSQ has an advantage dealing with this case. It does not need to flush its contents to memory. As long as the virtual indexing issues are properly handled, data can remain in the VSQ indefinitely.

2.3.3 Misspeculation

Speculative stores can not be released to memory. However, speculative stores can forward data from the store buffer. If a speculative store becomes non-speculative then nothing needs to change. If a speculative store needs to be squashed due to an incorrect speculation, such as a branch prediction, then that store buffer entry needs to be invalidated. Since stores are placed in a store buffer in a first-in, first-out manner, it is a trivial task to find the last non-speculative load and invalidate the remainder of the entries. It is possible that a speculative store could forward data to a load. This is also not a problem. Forwarding is only allowed between a store and a younger load. Therefore, if the store is along a misspeculated path, then the load will be also.

Speculative stores invalidated due to an inaccurate branch prediction should not remain in the VSQ or store buffer. The store buffer and a FIFO VSQ can selectively invalidate the misspeculated entries (similar to the ROB or store buffer). If the replacement policy of the VSQ allows for random ordering or store kills, then it is very complicated to recover properly from a branch. In this case, it is easiest to invalidate the entire VSQ.

2.3.4 Cache Configuration

The configuration of the cache plays a relevant role in the design of the store buffer. A machine with a virtually indexed and physically tagged cache (UltraSPARC-IIi, Alpha 21264) requires an address translation to take place in parallel with the cache access. This does not directly determine whether the store buffer is physical or virtual.

Each store has an entry in the store buffer. This entry can contain the physical address, the virtual address, either one or the other, or both. In general, it seems that most store buffers contain physical tags and only allow forwarding if both the load and related store have been translated. A load can look for data in the store buffer in parallel with accessing the cache or while waiting to be serviced by the cache. This is the case with a physically indexed data cache which always performs an address translation. Also, a virtually indexed data cache performs translations early and aggressively to reduce translation costs.

Chapter 3

Experiment Model

3.1 Simulation Tools Overview

The experiments in this thesis are performed using the SuperSim microarchitecture simulator. SuperSim is a self-made, execution-driven, cycle-level timing simulator that models the pipeline stages and resource contentions of a speculative out-of-order microprocessor core. SuperSim functionally executes any single-process SPARC executable. Therefore, the simulator uses the SPARC instruction set architecture [58] and handles the SPARC nuances in a proper fashion (e.g. register windows, conditional instructions, condition code registers, delay slots). All experiments are run on a Sun Solaris platform.

3.1.1 Functional Execution

The flexibility of SuperSim is due in large part to Shade, a simulation tool from Sun Microsystems [10]. Shade takes a SPARC executable as an input and dynamically executes the entire program including library calls. No special compile-time options or source code is necessary to create a Shade-readable executable. The output from Shade represents the retired instruction stream. Shade is a customizable tool and allows the user to specify the exact instruction information to collect. The information can be treated in any manner. Detailed information can be collected dynamically for every instruction and opcode. SuperSim extracts data such as the opcode fields, program counter, branch targets, memory addresses, and other relevant dynamic characteristics. Shade version 5.33A executes user and library code, but does not analyze kernel code. In addition, this version of Shade does not handle multi-threaded applications.

3.1.2 Resurrected Code

The use of software simulation to model modern high-performance microprocessors is becoming increasingly challenging as microprocessors grow in complexity. Accurate and meaningful performance analysis of an out-of-order, superscalar microprocessor is complicated by the fact that no component of the system is truly independent from the rest of the system. At the same time, each component of the system requires a fine level of simulation. Therefore, there exists a tradeoff between the accuracy of results and the amount of time necessary to create a simulation environment and perform the simulations.

SuperSim utilizes a structure that can decrease the simulation time of microprocessor software simulators and improve the accuracy of simulation. This is accomplished by exploiting static instructions that are executed many times dynamically. SuperSim recreates an approximate copy of the object code, called the resurrected code, using instructions from the dynamic instruction stream of the simulator [4]. The resurrected code decreases the time SuperSim spends decoding instructions, which is a significant amount of the simulation time.

Along with decreased simulation time, the resurrected code provides an improvement in accuracy for SuperSim, which does not have the entire program image available for use. Instructions are fetched from the resurrected code structure after a mispredicted branch and then introduced into the simulated processor. This allows for increased reality in the modeling of mispredicted path execution. In addition, the structure provides an elegant method for gathering statistical information regarding the use of specific static instructions. The resurrected code also becomes an easy means for quickly specifying internal simulator-specific hints and directions.

3.1.3 Simulated Microarchitecture

The base architecture model used in this thesis is based on a combination of the Sun UltraSPARC-II microarchitecture and features from the popular SimpleScalar *sim-outorder* default simulation model [6]. The model is a four-wide machine, i.e. four instructions can be decoded, dispatched, and retired each cycle. The basic architecture model is shown in Figure 3.1. The specific parameters may be found in Table 3.1.

The execution core of the base model contains two basic integer ALUs with one cycle latency and one cycle throughput (1-1), one integer multiply/divide unit (3-1 for multiply, 20-19 for divide), two basic floating point ALUs (2-1), one FP multiply-divide-square root unit (3-1 for multiplies, 12-12 for divides, 24-24 for square root), and two load-store units (1-1). Each of these units is supplied by a twelve entry, content addressable, queue of reservation station entries. The load-store unit calculates effective addresses and then dispatches the loads and stores to the proper location - the load queue or the store buffer. Branch target addresses are calculated prior to execution when possible. The simulator uses a separate 64-entry reorder buffer (ROB) and register file for floating point and integer instructions, as in the UltraSPARC. Stores are allocated entries in the ROB.

The branch predictor uses the gshare branch prediction scheme as described by McFarling [36]. The predictor contains 16k entries of two-bit counters and is direct-mapped. The table is indexed by the program counter hashed with 16 global

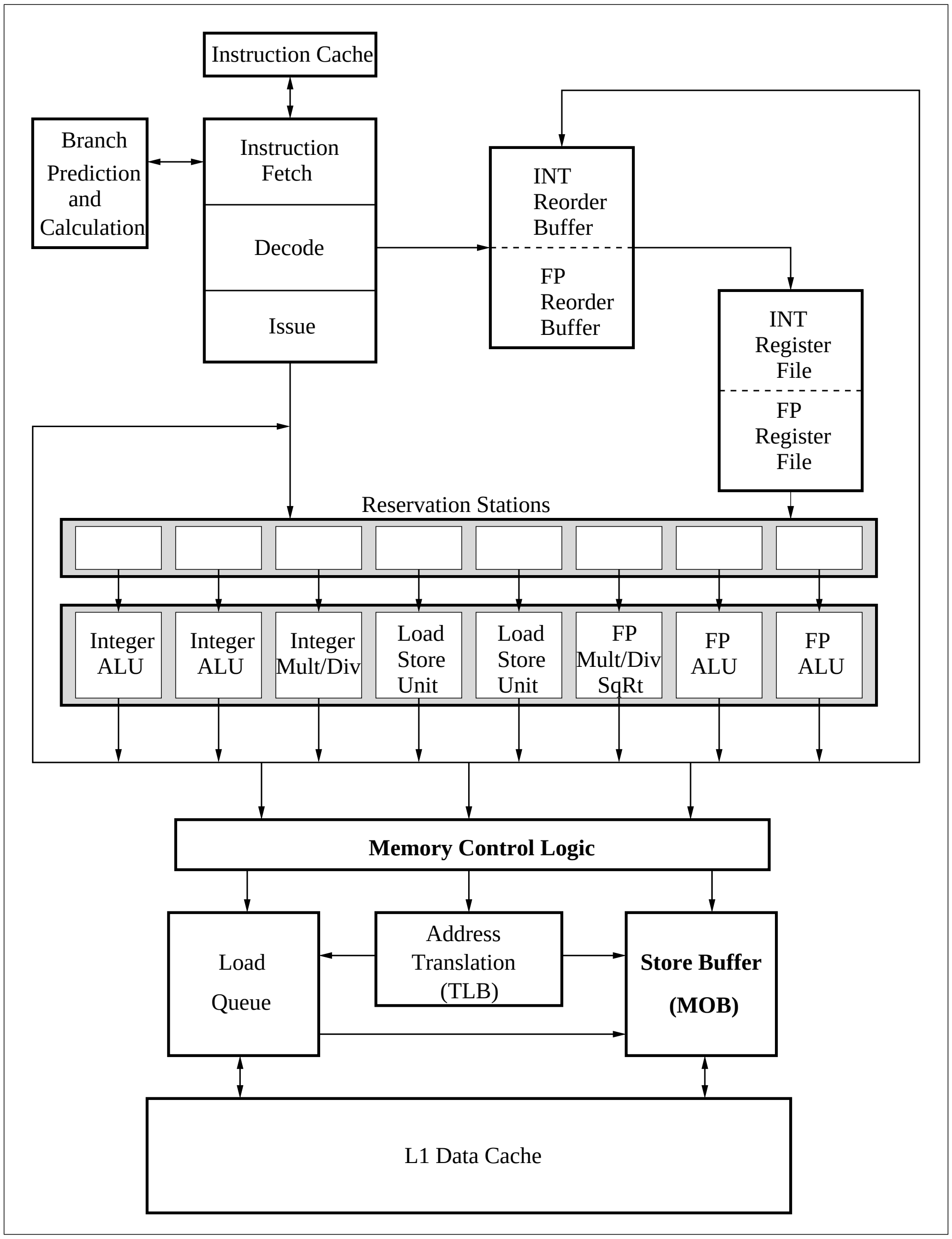


Figure 3.1: Overview of Simulated Base Microarchitecture

history bits. This predictor is accompanied by a 512-entry, direct-mapped branch target buffer (BTB) to predict target addresses for indirect branches.

Table 3.1: Simulated Architecture Parameters

Data memory			
· L1 Data Cache:	4-way, 64KB, 1-cycle hit		
· L2 Unified cache:	4-way, 1MB, 8-cycle hit		
· Non-blocking	8 MSHRs and 1 port		
· DTLB	128-entry, 4-way, 1-cycle hit, 30-cycle miss		
· Store buffer:	16-entry w/load forwarding loads access in 1-cycle		
· Load Queue:	16-entry		
· Main Memory	Infinite, +22 cycles		
Fetch Engine			
· L1 Instr cache:	4-way, 64KB, 1-cycle hit		
· Branch Predictor:	16k gshare predictor 3-cycle misprediction penalty		
· Indirect BTB	512-entry, direct-mapped		
· ITLB	128-entry, 1-cycle hit, 50-cycle miss		
Execution Core			
· Functional unit	#	exec. lat.	issue lat.
Load/store	2	1 cycle	1 cycle
Simple Integer	2	1	1
Int. Mul/Div	1	3/20	1/19
Simple FP	2	3	1
FP Mul/Div/Sqrt	1	3/12/24	1/12/24
· Separate 64-entry FP and INT reorder buffer			
· 12 reservation station entries/func. unit			
· Fetch width: 16 instructions			
· Decode width: 4 instructions			
· Issue width: 4 instructions			
· Execute width: 4 instructions			
· Retire width: 4 instructions			

The L1 instruction cache and the L1 data cache are 64 KB, four-way set-associative, write-through cache with a line size of 64 bytes, and a hit latency of one cycle. They use the LRU replacement algorithm. The L2 cache is a unified, 1 MB, four-way set-associative, write-back, write-allocate cache with a block size of 64 bytes, and a hit latency of eight cycles. It uses an LRU replacement scheme. Address translations are given a constant latency of one cycle for TLB hits and 30 cycles for TLB misses. The TLBs and caches are non-blocking.

Note that this thesis chooses to use an aggressive L1 data cache hit latency. This one-cycle time period includes the lookup and read from a large, set-associative cache. It is not uncommon for L1 cache accesses to take multiple cycles. As the memory latency increases, the importance of techniques like load forwarding and data speculation increases. Therefore, this aggressive configuration will lead to conservative gains from the proposed store buffer enhancements, virtual store queue, and the load tokens. Performance gains given a longer latency memory sub-system are presented briefly in Chapter 5.

The base store buffer model has 32-entries. Stores are retired once all previous instructions have completed. They are removed from the store buffer upon completion of their memory access (no lazy removal). Loads are always given priority over stores during memory interface contention. Loads may bypass stores while stores are always in-order. Loads are permitted to perform out-of-order with respect to each other. There is no VSQ in the base model.

Finally, the resurrected code is implemented as well. The resurrected code is created beforehand to prevent the slight disadvantage that results from dynamic creation. When the simulator is executing along the mispredicted path and encounters an instruction that is not in the resurrected code, no more instructions are fetched from the resurrected code and a branch redirection is initiated.

3.1.4 Differences Between Simulation Runs

All of the microarchitecture parameters listed above apply to the characterization experiments in Chapter 4 and the VSQ experiments in Chapter 5. The results from the store buffer experiments use slightly different values for several parameters. The indirect BTB is modeled as a perfect BTB in the store buffer simulations. Main memory latency is 30 cycles instead of 50 cycles. The number of MSHRs is two. There is one fewer fetch stage. The ITLB is modeled as a perfect buffer. The DTLB is direct-mapped instead of 4-way associative. Finally, and most importantly, the default store buffer is 32 entries instead of 16 entries.

The store buffer version of the model performs better than the VSQ version, but the VSQ version is a more realistic representation of processor hardware. The store buffer also tends to show larger performance changes because its idealistic fetch pipeline is less likely to become a bottleneck. This enables more memory bandwidth and latency optimizations to show up in the overall performance. Also, while the main memory latency difference is not a major factor in overall performance for these benchmarks, fewer MSHRs exacerbates the memory bottleneck. Therefore, reductions in memory traffic are welcome and serve to improve the situation.

3.2 Benchmarks

The SuperSim/Shade tools easily allow for simulation of benchmarks other than easy to compile benchmarks in C and Fortran, like the SPEC benchmarks. Object oriented programming has become a standard practice at all levels. C++ and Java are the primary vehicles for this paradigm and have shown characteristics that differ from the SPEC benchmarks including more memory references and indirect branches. Given the ability to simulate such workloads, this thesis includes a suite of C++ benchmarks and a suite of Java benchmarks in addition to the SPEC integer benchmarks. Descriptions of the benchmarks and the inputs used are in Table 3.2.

Simulation experiments are conducted on Sun UltraSPARC machines, using three sets of benchmarks to analyzed to evaluate the proposed microarchitecture schemes. The first group of benchmarks is the SPEC95 integer suite [55]. These commonly used C benchmarks are well-understood and a good point of reference. The second set of benchmarks is a C++ suite developed for the purpose of studying object-oriented workloads, specifically the effects of virtual functions on performance [8, 12]. These two suites of benchmarks are compiled with gcc 2.8.1. with full

Table 3.2: Benchmark Descriptions

Program	Description of Program	Length
SPEC CINT95: C programs		
compress95	Compresses large text files	38.8M
gcc	Compiles pre-processed source	267.1M
go	Plays the game Go against itself	500.0M
ijpeg	Performs jpeg image compression	500.0M
li	Lisp interpreter	170.6M
m88ksim	Simulates the Motorola 88100 processor	125.1M
perl	Performs text and numeric manipulations	41.7M
vortex	Builds, manipulates 3 interrelated databases	500.0M
Suite of C++ Programs		
deltablue	Incremental dataflow constraint solver	41.1M
eqn	Type-setting program for math. equations	48.2M
idl	SunSofts IDL compiler 1.3	85.5M
ixx	IDL parser generating C++ stubs	30.3M
richards	Operating system simulation benchmark	67.8M
SPEC JVM98: Java Programs		
compress	A popular LZW compression program	500.0M
db	IBM data management benchmarking software	81.7M
jack	Real parser-generator from Sun Microsystems	500.0M
javac	JDK Java compiler from Sun Microsystems	202.2M
jess	NASA's CLIPS rule-based expert systems	263.2M
mpegaudio	Core MPEG-3 audio decoding algorithm	500.0M
mtrt	Dual-threaded ray tracing program	500.0M

Long running benchmarks are stopped after 500 million dynamic instructions. The remaining benchmarks are run to completion.

optimizations (-O4) and are statically linked. The final group of benchmarks is the Java benchmarks from the SPECjvm98 suite [55]. The Java byte codes are executed by the Sun Java Virtual Machine (JVM) version 1.1.3. All thread management is handled by the JVM.

Chapter 4

Characterization of Store/Load Dependencies

4.1 Dynamic Memory Behavior

Characterizing and understanding the memory behavior of the workload is critical to understanding the reasons behind changes in microprocessor performance. The data presented in this section is collected using the base model configuration of SuperSim. Table 4.1 presents basic attributes of the store instructions in the three benchmark suites. Stores account for 8.92% of all instructions on average (`Pct of Instr`). In an out-of-order microprocessor with a memory hierarchy, the latency to access the cache can vary. In this thesis, the cache accesses vary from one cycle L1 hits to 58 cycles L2 misses. The L1 data cache hit rates (`Hit Rate`) for store instructions include only the one cycle accesses. Store instructions have good locality in these benchmarks. Only two programs show hit rates below 90%, and only `deltablue` is below 80%.

The average number of cycles to complete a store to memory (`Avg Latency`) is related to the hit rate, although they are not directly indicative of each other. For example, `db` and `idl` both have hit rates around 96%, but the average store latency for idl is significantly larger, 2.53 cycles versus 1.60 cycles. This can happen because of two events. One, a L1 cache miss can result in either a L2 cache hit or a L2 cache miss. Both factor into the L1 hit rate in the same way, but the L2 miss raises the average store latency. Two, a previous instruction may have already initiated the request for data that other instructions need. In this case, the second instruction may have missed the L1 cache but does not have to wait for the entire L2 cache latency because a prior instruction started a request to the same block of data.

Table 4.1: Characteristics of Store Instructions

Benchmark	Pct of Instr	Hit Rate	Avg Latency	SB Stall %	Avg SB Size
compress95	15.17	87.56	2.69	0.00	6.44
gcc	10.14	97.42	1.25	0.00	2.52
go	7.41	99.56	1.02	0.12	1.99
ijpeg	6.54	98.80	1.12	0.00	3.22
li	10.16	98.54	1.19	0.17	3.36
m88ksim	8.70	93.53	4.42	0.00	4.21
perl	10.90	97.16	1.23	0.73	3.48
vortex	11.05	93.55	3.70	0.00	3.96
deltablue	6.29	76.11	4.64	0.97	2.50
eqn	9.62	99.58	1.04	0.08	2.60
idl	2.74	96.08	2.53	0.63	1.08
ixx	7.91	95.51	3.08	2.45	2.50
richards	8.44	99.92	1.00	0.01	3.59
compress	10.11	99.27	1.23	0.00	2.92
db	7.85	96.36	1.60	0.81	2.64
jack	9.76	98.88	1.11	0.22	2.74
javac	8.08	97.13	1.36	0.49	2.87
jess	8.52	97.48	1.27	0.38	3.25
mpegaudio	9.47	99.44	1.08	0.13	2.26
mtrt	9.51	98.08	1.13	0.31	3.67

Pct of Instr: percentage of all dynamic instructions that are stores. *Hit Rate:* L1 data cache hit rate for store instructions. *Avg Latency:* Average number of clock cycles needed to complete a store to memory. *SB Stall:* Percentage of cycles where a store buffer stall is encountered. *Avg SB Size:* Average number of active store buffer entries each cycle.

The last two columns are characteristics of the store buffer. The SB Stall % represents the percentage of cycles in which a store buffer stall occurs. In this case, the issue stage is forced to stall on a store instruction because there are no available entries in the store buffer. This does not occur on a significant number of cycles for most benchmarks. Only ixx shows a strong tendency to back up the store buffer. The last column is the average store buffer occupancy or the average number of active entries each cycle (`Avg SB Size`). On average, 3.09 of the 16 store buffer entries are active each cycle. Therefore, on average, each load has three candidates from which it can forward its data.

The latency of load instructions is more critical to overall instruction throughput than that of store instructions. This is because load instructions often start data dependency chains in the dynamic instruction stream while store instructions do not. If the instructions that depend on the load are close to the load in the instruction stream, then providing a load instruction with its data as quickly as possible expedites instruction flow. Table 4.2 has data similar to the previous table, but for the load instructions.

Table 4.2: Characteristics of Load Instructions

Benchmark	Pct of Instr	Hit Rate	Avg Latency	Load Fwd Pct	Avg LQ Size
compress95	17.91	98.05	1.29	11.25	1.24
gcc	18.97	98.90	1.10	7.84	0.56
go	21.75	99.91	1.00	5.51	0.61
ijpeg	17.19	99.69	1.02	1.56	0.99
li	22.28	96.10	1.43	7.56	0.62
m88ksim	15.61	99.81	1.04	19.60	0.28
perl	20.91	99.87	1.01	11.11	0.88
vortex	21.50	99.22	1.10	5.71	0.92
deltablue	25.71	86.70	2.20	2.56	0.91
eqn	17.73	99.90	1.01	5.88	0.56
idl	22.55	98.19	1.13	3.79	0.63
ixx	15.50	98.32	1.34	8.70	0.46
richards	28.39	99.99	1.00	8.29	1.33
compress	31.69	99.16	1.19	14.29	0.92
db	20.92	98.53	1.29	8.15	0.74
jack	28.85	99.41	1.09	12.89	0.81
javac	22.53	98.34	1.28	8.35	0.85
jess	23.67	98.07	1.26	8.59	0.94
mpegaudio	30.98	99.81	1.03	10.63	0.76
mtrt	26.03	99.23	1.11	9.82	1.11

Pct of Instr: percentage of all dynamic instructions that are loads. *Hit Rate:* L1 data cache hit rate for load instructions. *Avg Latency:* Average number of clock cycles needed to complete a load from memory. *Load Fwd Pct:* Percentage of all loads that have data forwarded from the store buffer. *Avg LQ Size:* Average number of active load queue entries each cycle.

Load instructions represent 21.3% of the dynamic instruction stream on average, a significantly higher percentage than store instructions. The hit rates for load instructions are also better than store instructions. The higher hit rates have more to do with the frequency of the load instructions than the locality. Per address, there tend to be multiple loads for each store. Because the hit rates are higher, it is not surprising that the `Avg Latency` is lower than for store instructions. This is good since load instructions are critical to instruction flow. The benchmark `deltablue` stands out as a program with a poor L1 data cache hit rate.

The next column, `Load Fwd Pct`, represents the percentage of all load instructions that receive their data from a store in the store buffer. This varies greatly over the benchmark suites. The SPECjvm programs have a higher forwarding rate due to the stack-based nature of the Java byte codes. The C++ programs, on the other hand, have a much lower load forwarding percentage. The SPECint programs are program-specific, fluctuating from 1.56% to 19.60% forwarded loads. Benchmarks with a combination of a high percentage of loads, a high average load latency, and a lower load forwarding percentage are the best candidates to be aided by the techniques discussed in this thesis.

One component of the cache access latency is the arbitration for the memory ports. Memory port stalls and utilization are determined by the bandwidth out of the core as well as the tendency of the memory request to cluster in time. Table 4.3 has information related to the memory port of the base model.

A store instruction arbitrating for a memory port is denied sometimes because another memory operation is utilizing the port. The first column in the table represents the percentage of cycles where this is happening (`Store Port Stall %`). The second column represents a similar statistic for load instructions (`Load Port Stall %`). The memory port is two-way, so returning data does not block memory requests. Therefore, the memory port is only occupied on the initial cycle of a memory request. However, this results in roughly 50% port utilization for the benchmarks studied.

Table 4.3: Memory Port Contention

Benchmark	Store Port Stall%	Load Port Stall %	Port Utilized %
gcc	12.04	6.05	42.21
compress95	22.24	9.04	50.93
go	8.65	8.10	43.12
ijpeg	8.35	12.35	43.02
li	17.92	6.85	53.78
m88ksim	8.22	2.76	33.59
perl	17.47	7.87	52.80
vortex	16.62	9.59	54.07
deltablue	6.267	7.70	31.70
eqn	14.86	7.00	46.69
idl	4.80	12.50	41.22
ixx	11.17	4.99	40.20
richards	7.736	16.81	49.74
compress	13.01	13.91	56.81
db	11.24	9.18	43.00
jack	12.87	10.56	53.75
javac	11.54	10.89	44.80
jess	12.08	11.60	45.78
mpegaudio	12.82	14.28	58.48
mtrt	16.92	14.91	56.26

Store Port Stall %: is the percentage of cycles where a store could not proceed to memory due to port utilization. *Load Port Stall %:* is the percentage of cycles where a load could not proceed to memory due to port utilization. *Port Utilized %:* is the percentage of cycles where the L1 data cache memory port is utilized.

There are two important items to consider when viewing this table. First, remember that store instructions account for 8.92% of the instruction stream while load instructions account for 21.3%. Second, note that in the base model loads get priority over store instructions in the base model. Although load instructions are twice as frequent as store instructions, loads spend fewer cycles stalling on a utilized port due to the arbitration policy. This is one of the primary reasons that the store buffer occupancy from Table 4.1 is much higher than the load queue occupancy.

4.2 Same Address Memory Operations

To determine the potential effectiveness and implementation of the VSQ, it is important to understand how dependent loads and stores interact with each other. Similar analysis of memory access patterns has been done in the past. This thesis is reporting results of programs compiled and run on a Sun UltraSPARC-II. This experimental data is useful to motivate the thesis topic and verify that previous results apply to the SPARC platform.

Table 4.4 presents characteristics of the dynamics reference sequence. The fist column is the percentage of total dynamic instructions that are loads. The second column is the percentage of total dynamic instructions that are stores. Note that these values are different that in Tables 4.1 and 4.2. The benchmarks below are run for the entire length of the benchmark. Some of the benchmarks are not included to the memory intensive nature of the characterization.

Table 4.4: Memory Characterization of Benchmarks

Benchmark	% Loads	% Stores	% Addr No loads	% Addr One Load
compress	17.72	15.01	61.15	10.16
gcc	18.57	9.93	12.87	34.5
go	21.08	6.87	1.465	78.14
ijpeg	17.28	6.56	33.05	34.37
li	21.72	9.90	7.995	2.604
m88ksim	16.46	16.90	47.02	28.63
perl	13.74	9.46	12.76	57.00
vortex	20.66	6.79	3.338	83.38
deltablue	25.41	6.22	14.61	33.81
eqn	17.36	9.42	20.29	28.06
idl	21.84	2.65	21.45	36.97
ixx	15.16	7.74	16.81	49.4
richards	27.48	8.18	5.00	68.5

The Java benchmarks are not included due to their lengthy run times and large memory requirements.

Column three is the percentage of unique virtual addresses that are written to by a store instruction but are never read by a load. The final column is the percentage of unique virtual addresses that are written and only read once before the program ends or the store is overwritten. The remainder of the unique memory writes are read multiple times. Obviously, store addresses that have no subsequent reads will not provide any benefit to a load forwarding mechanism. Store addresses with one or more loads can potentially forward data to a load depending on how many cycles elapse between the two instructions.

In these benchmarks, `compress` does not look like a good candidate to benefit from the proposed mechanisms because over 60% of all stores have no dependent load and an additional 10% of the store instructions only have one dependent load. In contrast, a benchmark like `li` only 8% of the store instructions do not have a dependent load, and 89,4% of the writes have multiple reads. The programs `go`, `vortex`, and `richards` are unique in that they have a very large percentage of read once addresses. Events such as register spills and stack accesses can result in read-once writes. When this is the case, a large percentage of read-once writes leads to substantial load forwarding.

Another interesting statistic is the the distance between a store and a subsequent loading of the stored data. The distance is measured in the number of total stores to any address. This characterization has been presented by Moshovos and Sohi [40]. This metric gives an indication of the effectiveness of potential VSQ buffer sizes.

The percentages presented in Table 4.5 are the percentage of loads that have a dependent store within some store distance. For example, 39.2% of the load instructions in `m88ksim` could be satisfied by one of the previous eight store instructions. Figure 4.1 summarizes the store distance by benchmark suite. Across the benchmarks, approximately 50% of stores are read within 1024 stores. This translates to a potential 50% decrease in load traffic for a 1024-entry VSQ. These results correlate well to those presented by Moshovos and Sohi.

A similar metric is the store distance between writes to the same address.

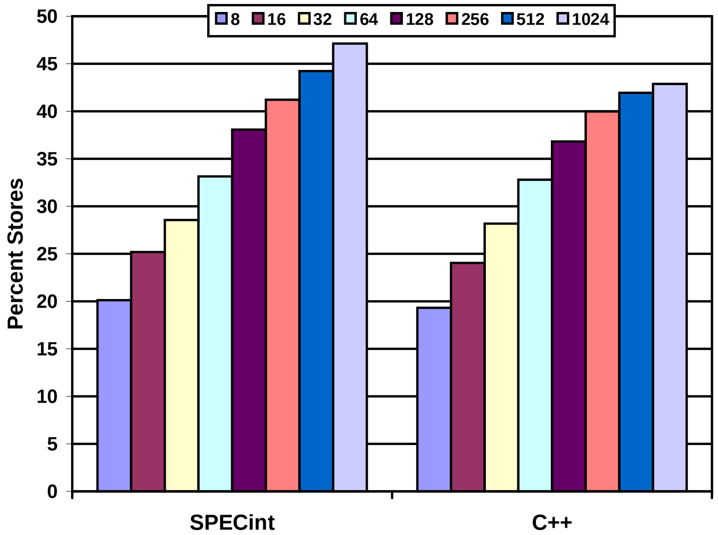


Figure 4.1: Stores Between a Dependent Store and Load

Table 4.5: Percentage of Dependent Loads Within a Store Distance

	Distance in Number of Stores								
Benchmark	4	8	16	32	64	128	256	512	1024
compress95	0	23.8	38.5	40.8	45.1	48.9	49.7	49.9	50.4
gcc	0	17.6	22.6	28	32.5	37.1	41	44.5	47.1
go	0	19.1	24.8	30.7	36.7	42.6	48.1	53.5	58.7
ijpeg	0	4.75	5.62	6.69	13.6	18.1	19.2	20.7	23.1
li	0	17	21.6	24.5	27.4	31.6	36.3	40.7	43.5
m88ksim	0	39.2	40.2	41.4	44.2	51.8	52.6	53.4	54
perl	0	19.4	23	27.8	32.5	36.4	41.6	46.9	53.1
vortex	0	13.2	15.5	18.4	23.2	29.4	35.3	40.1	48.8
deltablue	0	11.7	16.7	19.8	21.3	22.8	24.6	25.1	25.4
eqn	0	23.4	28.3	33.5	39	42	43.5	44.5	45.5
idl	0	14.3	16.4	19.4	24.4	32.1	38.7	43.8	45.7
ixx	0	19.4	23.7	27.4	32.6	34.9	36.1	37.3	38.5
richards	0	27.8	35.1	40.8	46.7	52.3	57	59	59.3

This table corresponds to Figure 4.1. The Java benchmarks are not included due to their lengthy run times and large memory requirements.

Table 4.6 approximates the life expectancy of store instructions. These numbers show that as many as 31% of stores can be rewritten before four more stores occur. Figure Table 4.5 showed that 0% of stores are read within four stores. Therefore, any store that is rewritten within four instructions is never loaded and probably unnecessary. Memory locations that are quickly rewritten, especially those rewritten with being read, pollute structures like the VSQ. These unused stores consume entries that could belong to a store with forwardable data.

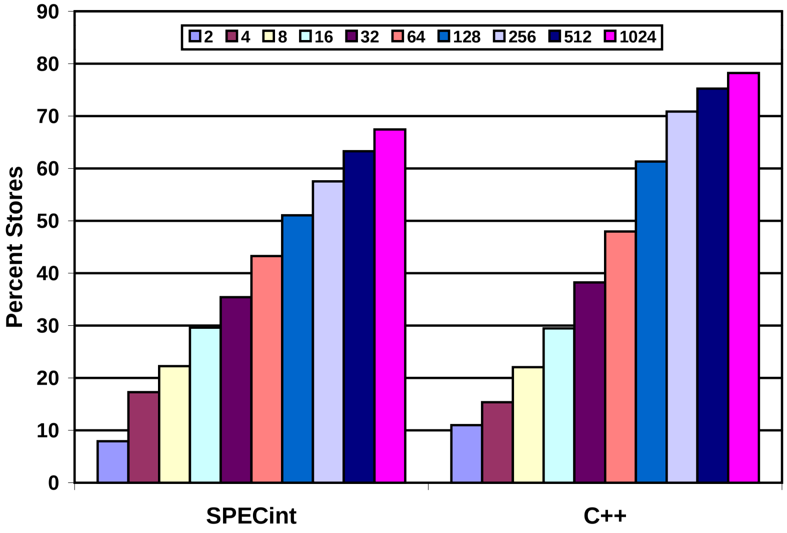


Figure 4.2: Percentage of Same Address Stores within a Store Distance

The two previous sets of data use the number of stores as the means for measuring distance. However, this does not provide enough insight into whether or not the dependent load will read from the store buffer during the course of execution in a real processor. Using the base configuration, the actual dynamic instruction distance between a dependent store/load pair is measured. The results from this study are presented in Figure 4.3.

Somewhat surprisingly, many store/load pairs are only a few instructions apart in the dynamic instruction stream. For the C++ benchmarks, about 50% of all

Table 4.6: Percentage of Same Address Stores Within a Store Distance

	Distance in Number of Stores									
Benchmark	2	4	8	16	32	64	128	256	512	1024
compress	5.49	19.9	21.5	42.2	45.5	48.3	49.7	54.6	54.6	54.6
gcc	4.06	6.67	13.9	22.5	32.5	38.7	44.2	49.7	71.5	83.9
go	8.64	16.2	22.8	29.7	38.8	48.8	58.2	68.1	76.9	83.6
ijpeg	4.27	4.81	6.97	8.41	9.86	18.6	27.9	38.5	40.2	42.2
li	16.2	20.5	28.0	36.5	44.7	50.7	55.8	58.0	59.1	60.4
m88ksim	14.8	48.1	51.8	52.7	53.3	56.2	73.2	74.6	75.5	76.1
perl	1.97	4.83	10.8	15.2	23.3	41.6	48.4	59.2	65.2	71.3
vortex	25.2	31.1	38.8	40.7	42.3	46.5	53.1	83.4	83.8	84.1
deltablue	3.21	5.14	6.5	16.6	23.1	24.7	35.1	52.2	55.9	56.2
eqn	3.53	9.99	12.4	15.8	25.7	44.1	67.1	72.3	75	77
idl	20.2	23.9	34.7	41.4	47.5	54	66.9	74.4	77.7	82.3
ixx	5.41	17.9	23.2	26.4	31	42.5	53.7	62.8	69.1	75.7
richards	11.6	19.9	33.5	47.1	63.9	74.5	83.8	92.6	98.5	99.9

This table corresponds to Figure 4.2. The Java benchmarks are not included due to their lengthy run times and large memory requirements.

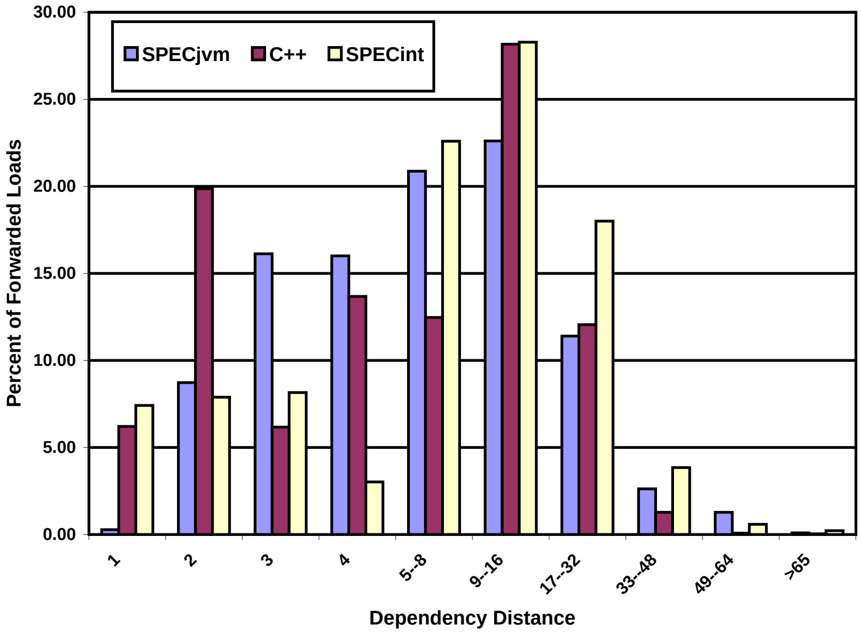


Figure 4.3: Dynamic Load Forwarding Dependency Instruction Distance
The average of the percentages of the individual benchmarks is represented in each bar.

forwarding stores are less than four or fewer instructions from a dependent load. The percentage for SPECint is approximately 41%, and 25% for SPECjvm. Many of these data dependencies are better communicated through registers than memory addresses. However, the compiler can not always identify these cases due to memory disambiguation uncertainties and unpredictable dynamic flows. Almost all of the dynamic load forwarding is done within an instruction distance of 32 instructions for all of the benchmarks. Since stores retire and are removed from the store buffer, as the instruction distance grows the store is less likely to be active in the store buffer. So large dynamic instruction distances are rare as a function of the realistic hardware setting. Note that wrong path instructions due to branch mispredictions count as part of the instruction distance and can lead to dependency distances greater than ROB size.

Chapter 5

Store Buffer and VSQ Result Analysis

5.1 Store Buffer Performance

A thorough set of experiments varying the individual store buffer policies for the base-line physical store buffer are presented in this section. Pipeline placement is found to have the greatest impact on the overall performance of the processor. The lazy store removal policy has the greatest impact on memory traffic. Priority switching is found to have negligible impact at almost all thresholds. Finally, varying store retirement policies provides some of the same benefits as a lazy store removal policy, but with a higher penalty.

The optimal 32-entry store buffer in the design space examined is a virtual store buffer with lazy store removal using a threshold of 24 active entries, no priority switching, and the original store retirement policy. The next section provides more details and analysis on how this configuration is determined. Table 5.1 reviews some of the basic characteristics of the benchmarks. The values in this table are essential to understanding the performance impact of the store buffer policies. Note that some values are different than those found in Tables 4.1 and TAB:LOADS in Chapter 4. This is because of the different configurations used for the experiments as noted in that Chapter.

Table 5.1: Performance of the Base Model

Benchmark	IPC	% Forw Lds	Ld Hit Ratio	St Hit Ratio	SB Stall %	Avg SB	Addr Trans
compress95	1.90	14.60	98.1	93.2	13.14	11.05	13.2M
gcc	2.28	11.87	98.7	97.9	0.43	5.81	77.6M
go	1.61	8.02	99.9	99.5	0.03	2.71	196.3M
ijpeg	2.06	2.37	99.6	98.9	1.89	5.78	124.1M
li	2.38	11.13	95.9	98.9	0.12	6.88	52.3M
m88ksim	2.54	32.51	99.7	97.0	4.57	6.25	29.2M
vortex	2.55	8.59	98.9	98.0	1.76	7.58	138.4M
deltablue	1.28	3.92	86.4	78.3	0.84	3.97	12.9M
eqn	2.48	9.65	99.9	99.7	0.05	6.33	13.2M
idl	2.45	4.79	98.1	97.5	0.73	2.12	20.3M
ixx	2.33	13.18	98.3	97.2	3.53	5.93	7.1M
richards	1.87	12.83	99.9	99.9	0.01	5.90	23.1M
db	2.30	14.22	98.6	96.9	0.75	6.65	25.0M
jack	2.58	24.97	99.3	98.9	4.37	16.94	191.4M
javac	2.20	14.48	98.4	97.5	0.42	6.61	61.2M
jess	2.13	13.51	98.1	97.7	0.26	6.71	83.8M
mpegaudio	2.75	24.62	99.7	99.5	1.41	13.11	201.0M
mtrt	2.39	15.24	99.2	98.2	0.15	8.66	173.3M

IPC is instructions per cycle. *% Forw Lds* is the percentage of all loads that are forwarded. *Ld Hit Ratio* is the L1 data cache hit ratio for load instructions. *St Hit Ratio* is the L1 data cache hit ratio for store instructions. *SB stall %* is the percentage of all processor cycles with a store buffer stall. *Avg SB* is the average number of active entries in the store buffer. *Addr Trans* is the total number of address translations.

5.1.1 Per Policy results

In Figure 5.1, the four policies and one combination are summarized based on their variation in IPC from the base model. A virtual store buffer policy (`V`) has the most significant impact on IPC followed by a lazy store removal policy (`LR24`) and a late store retirement policy (`RP16`). Priority switching (`P24`) had little effect in most cases. After examining different thresholds for these individual policies in prior simulations, these are the best. Combining the virtual store buffer with lazy store removal (`LR24.V`) provides the best overall performance and is represented in the figures.

Figure 5.2 summarizes how the configurations affect load traffic to the L1 data cache. The single parameter with the greatest impact on load traffic is lazy store removal. Late store retirement does not provide as much load traffic reduction. A virtual store buffer alone provides only a small improvement. When a lazy store removal policy is added to a virtual store buffer, it has a synergistic effect where the combined improvement in IPC is more than the sum of the individual improvements. A more in-depth analysis of these policies follow.

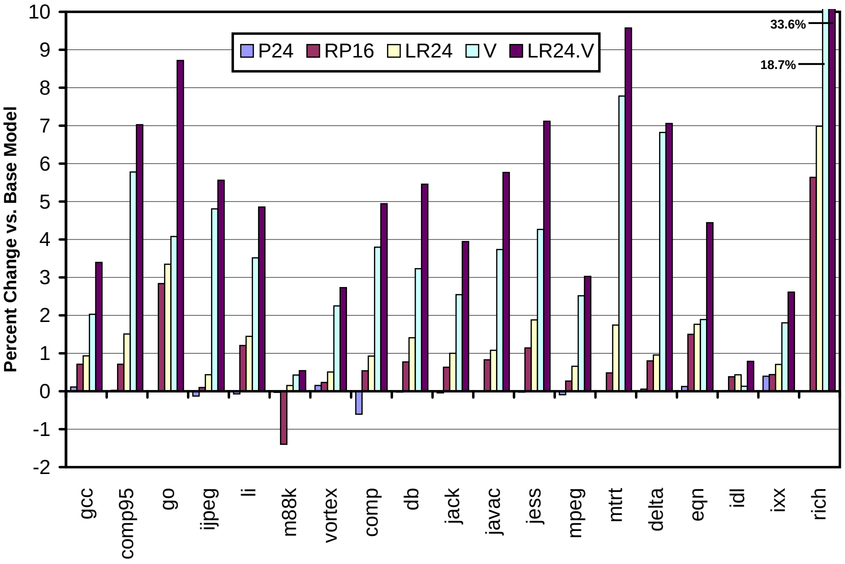


Figure 5.1: Effects of Store Buffer Policies on IPC

P24 switches high priority from loads to stores at 24 active entries. RP16 is a store buffer with store retirement point of 16 active entries. LR24 is a store buffer with lazy store removal at 24 active entries. V is a virtual store buffer. LR24.V is a virtual store buffer with lazy store removal at 24 active entries.

Impact of Lazy Store Removal. The policy of lazy store removal alone has an overall positive effect on the processor. Each benchmark analyzed has an increased IPC, ranging from a negligible performance increase in m88ksim to 3.3% IPC improvement in richards. The average store buffer occupancy rises to about 23 entries. Another effect of this policy is a substantial increase in the amount of load forwarding and therefore a reduction in load requests sent to memory. The amount of load traffic is reduced by an average of 12.5%, ranging anywhere from 3.3% to 28.6%.

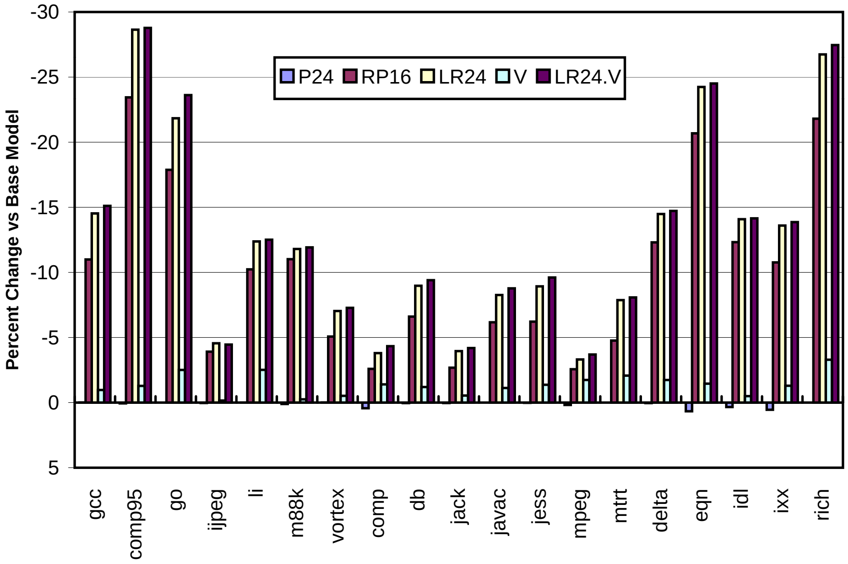


Figure 5.2: Effects of Store Buffer Policies on L1 Load Traffic

`P24` switches high priority from loads to stores at 24 active entries. `RP16` is a store buffer with store retirement point of 16 active entries. `LR24` is a store buffer with lazy store removal at 24 active entries. `V` is a virtual store buffer. `LR24.V` is a virtual store buffer with lazy store removal at 24 active entries.

Unfortunately, all of these saved loads do not translate directly into performance increase. Since the load operations must have the same address as a recent store to perform load forwarding, almost all of the loads being forwarded would have been hits in the level one cache (which has only one-cycle latency). Expensive loads, like L1 misses, are not usually caught by the store buffer. The simulated memory system also provides two MSHRs to further hide access latencies. In addition, a dynamically scheduled processor's ability to tolerate loads varies from load to load and program to program [54]. It is possible that the loads that are avoided due to increased load forwarding are ones that can tolerate latency.

It is interesting to note that the percentage of cycles with a store buffer stall varies very little, if at all. Although the number of average entries in the store buffer has increased, many of them (especially the older ones) are stores that have already been retired to memory. Therefore, they can be purged quickly if necessary.

Impact of Store Priority Switching. Switching priority from *loads first* to *stores first* at a threshold of 24 does not have a significant effect on performance in store buffers. In most cases, it reduces the percentage of cycles with a store buffer stall, but it is common for the IPC to actually decrease compared to the base model.

For example, the Java program `jack` had a large percentage of store buffer stall cycles in the base model, 4.37%. Using the priority switching model, this is reduced to 0.13%, but the IPC decreases by 0.05%. There are two probable reasons for the performance decrease. One is that the amount of load forwarding has been decreased, allowing more loads to access memory and incur a longer latency. The other reason is that the reduction in cycles with a store buffer stall is relatively small. Raising the threshold at which the store priority switches to 24 active entries also decreases performance, but by a lesser amount. The programs `compress95` and `idl` which also had significant stall cycles due to the store buffer did not see any improvement in performance from this policy.

Store Retirement Point. We can see that modifying the store retirement point results in a performance increase over the base model. The IPC is increased by an average of 0.93%, including one case of an IPC decrease. Our studies show that the average occupancy of the store buffer increases from an average of less than eight in the base model to an average of about 17. Therefore, increasing the store retirement threshold improves the potential of load forwarding, but also increases store buffer stalls. We find that the store buffer stalls degrade the performance gain from the extra load forwarding.

There is also an effect similar to that of a lazy store removal policy, a large increase in load forwarding. The difference is that the percentage of cycles with a store buffer stall now increases more substantially. A store can not be considered for removal until it has been retired. In the lazy removal case, stores are retired early and are fully prepared to be purged from the store buffer when the threshold is reached. In the late retirement scenario, stores are less likely to be prepared for removal as the store buffer fills with active entries.

These results show that allowing stores to contend for the memory interface resources as soon as possible does not hurt performance. If there are several outstanding stores, they can block the cache from performing important loads, but this does not appear to be a critical issue to performance. It is more important to utilize available L1 bus cycles.

Impact of Virtual Store Buffer. Implementing a virtual store buffer produces the best performance increase of any single policy studied. The IPC for the benchmarks increased by an average of 4.2%, ranging from a 0.13% increase to 18.77% increase in `richards`. Reducing the address translation step is the primary reason for the performance gain. The number of translations is reduced by an average of 5.82%, ranging from 0.96% to 10.89% since it is assumed that a process ID and a virtual address are sufficient to properly determine address dependencies.

For each load that is forwarded, the address translation latency is not incurred (the model allows one cycle for TLB hits which occur 99% of the time). Load forwarding increases slightly with a virtual store buffer versus the base model, because the load can access the store buffer earlier.

Adding Lazy Store Removal to a Virtual Store Buffer. Studying several combinations of the discussed policies, the best processor performance for a processor with a 32-entry store buffer is achieved by making it virtual and implementing the lazy store removal policy with a threshold of 24. Store retirement should remain at the point indicated in the base model, once all previous instructions are completed and the store is non-speculative. Switching store priority does not need to occur for performance reasons, although in the UltraSPARC-IIi it is used to avoid “lock-out conditions.”

Lazy store removal increases the number of forwarded loads, and virtual accessing reduces the number of address translations that can be saved. If the best case (33.6% increase for `richards`) and the worse case (0.54% increase for `m88ksim`) are ignored, we find that the performance increase available from combining a virtual store buffer with lazy removal ranges from 0.79% to 9.57%, and averages 5.11%. The decrease in L1 load traffic and address translations is substantial. The number of loads that access memory is reduced by an average of 12.97%, while the number of address translations is reduced by 12.63%. The address translations include translations for store instructions, so they do not reduce at exactly the same rate as the load traffic.

Load traffic reduction is a direct result of increased load forwarding. By implementing lazy store removal, the average number of active entries in the store buffer increases, improving the chances for load forwarding. Making the store buffer virtual accounts for the address translation reduction. For each forwarded load, there need not be an address translation to complete the load.

5.1.2 Policies versus. Size

This section investigates the effects of store buffer size with and without the improvement from the optimal policies. Figure 5.3 compares an array of configurations averaged over the three different benchmark suites. In addition to the original 32-entry size, sizes of four, eight, and sixteen are simulated. Each of the new sizes is optimized with the lazy store removal policy and then made virtual. The objective is to observe whether a smaller, smarter store buffer can outperform the larger, naive store buffer. This is important since it is often the case that access time due to disambiguation is less for smaller store buffers, making these policies easier to implement.

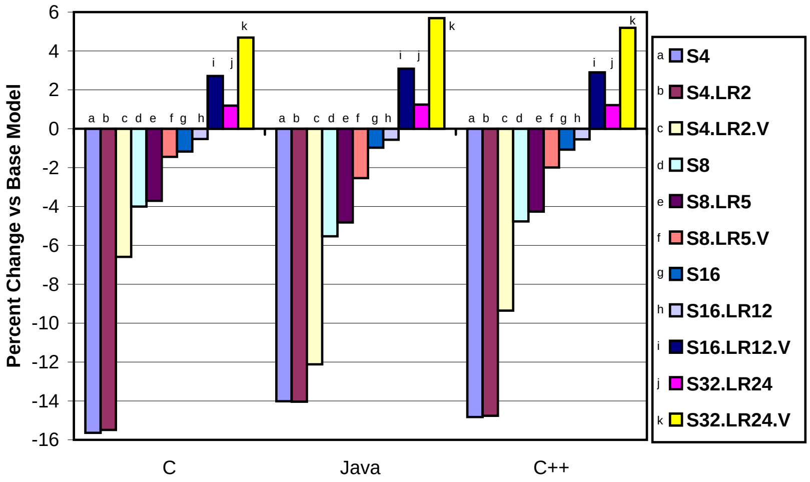


Figure 5.3: Effects of Store Buffer Size and Policy on IPC
`SX` indicates the size of the store buffer where X is the number of entries. `LRX` indicates the lazy store removal threshold where X is the the threshold. `V` indicates a virtual store buffer. These percentages are relative to the 32-entry, physical store buffer base model.

Figure 5.3 demonstrates that it is possible for smaller store buffers to approach and, in fact, surpass the performance of a larger store buffer. The first thing to note from this figure is that down-sizing to a 16-entry store buffer (`S16`) results in little performance degradation, about 1% overall and a maximum of 4%. The average store buffer size for the benchmarks is under eight active entries per cycle in the base runs. Therefore, it is not surprising that a store buffer of 16 is quite adept at handling the stores when no other policies are applied.

When lazy store removal is added (`S16.LR12`), the overall IPC is within 0.33% of the base model and six benchmarks actually improve over the base model as seen in Figure 5.4. If this store buffer is made virtual (`S16.LR12.V`), then all but one of the benchmarks (`m88ksim`) improves over the base model 32-entry store buffer.

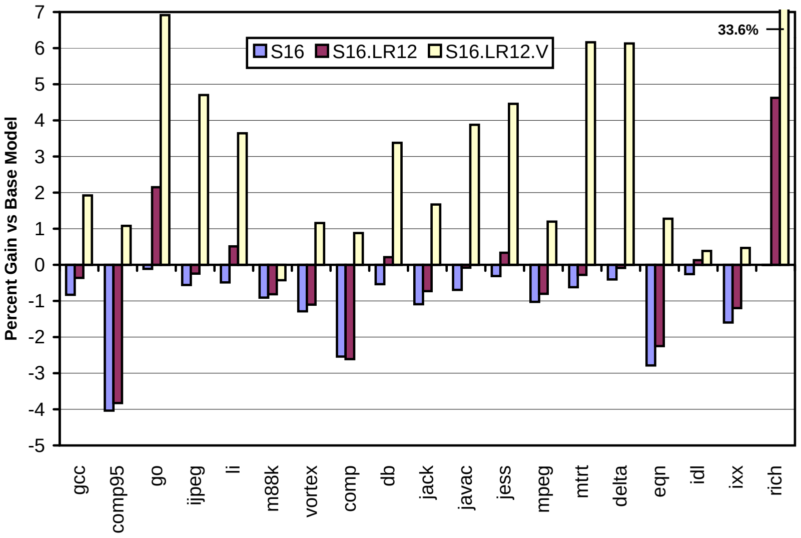


Figure 5.4: Change in IPC for a 16-entry Store Buffer

`S16` is a 16-entry store buffer. `S16.LR12` is a 16-entry store buffer with lazy store removal at a threshold of 12 entries. `S16.V` is a 16-entry virtual store buffer. `S16.LR12.V` is a virtual 16-entry store buffer with lazy store removal at 12 active entries. These percentages are relative to the 32-entry, physical base model.

Figure 5.3 also shows that the enhanced four-entry and eight-entry store buffers are not able to outperform larger store buffers on average. The four-entry store buffer shows a significant performance drop versus the base model. This is the result of the buffer being too small. A store buffer stall occurs in about one-third of all cycles in this case. The average store buffer occupancy per cycle is almost three entries, which explains the ineffectiveness of a lazy store removal threshold of two (`S4.LR2`). Making the four-entry store buffer virtual (`S4.LR2.V`) is a big improvement, especially for the C programs, but does not really approach the performance of a simple eight-entry buffer (`S8`).

The optimal eight entry buffer (`S8.LR5.V`), on the other hand, does approach the performance of a naive 16-entry store buffer (`S16`), despite the fact that its simple configuration (S8) is significantly worse that that of the 16-entry store buffer (S16).

Figure 5.4 illustrates that six benchmarks perform better on the optimal eight-entry buffer than the naive 16-entry buffer and four of those (`go, ijpeg, deltablue, richards`) perform better than the base model.

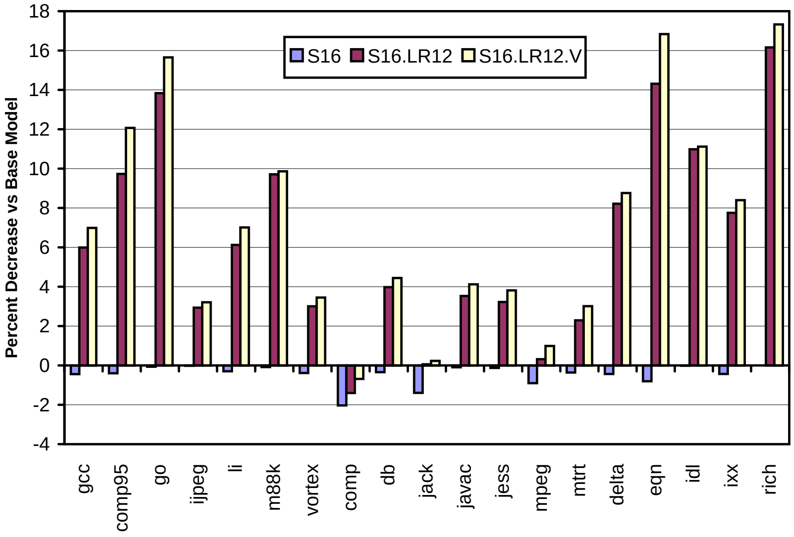


Figure 5.5: Change in Load traffic for a 16-entry Store Buffer
`S16` is a 16-entry store buffer. `S16.LR12` is a 16-entry store buffer with lazy store removal at a threshold of 12 entries. `S16.V` is a 16-entry virtual store buffer. `S16.LR12.V` is a virtual 16-entry store buffer with lazy store removal at 12 active entries. These percentages are relative to the 32-entry, physical base model.

The IPC numbers indicate that the virtual aspect of the 16-entry store buffer increases performance more than lazy store removal. In Figure 5.5, it is apparent that almost all of the improvement in load traffic is the result of the lazy store removal. So, the extra performance provided by virtual store buffers is strictly the result of a lower latency for acquiring load data. The details of the best configuration are presented in Table 5.2.

In the table, all percentages are relative to the 32-entry, physical base model. Column one shows the difference in IPC. In all but one case, this half-size store buffer outperforms the 32-entry base model. The next column shows the reduction in load traffic. Except with the Java `compress` program, all benchmarks show an improvement in load traffic to the data cache. The virtual aspect allows for the reduction in address translations (column three). The fact that, in column four, most of the benchmarks show an *increase* in average occupancy versus a store buffer of twice the size emphasizes the inefficiency of the straightforward base model. The last column reports the percent change in store buffer stall percentage. This simply shows that even though the percentage of stall cycles increases, performance may still be improved. The store buffer stall percentage is often quite small to begin with, so slight increases in the absolute value of store buffer stall cycles will show a large percentage increase.

Table 5.2: Virtual Store Buffer of Size 16 with Store Removal Threshold of 12

	Percent Change versus Base				
Benchmark	IPC	Loads to L1	Addr Trans	Avg SB	SB stall %
compress95	1.07	-12.06	-9.90	10.77	46.54
gcc	1.92	-6.98	-8.07	96.55	502.30
go	6.91	-15.65	-14.65	274.48	924.49
ijpeg	4.70	-3.20	-2.50	97.92	98.32
li	3.64	-7.01	-7.31	64.43	1059.31
m88ksim	-0.42	-9.86	-20.16	88.57	19.84
vortex	1.15	-3.44	-4.58	53.59	101.32
compress	0.88	0.68	-6.59	-6.91	405.45
db	3.37	-4.44	-6.39	73.85	245.27
deltablue	6.12	-8.75	-7.84	181.98	110.66
eqn	1.27	-16.83	-11.47	80.69	9692.08
idl	0.38	-11.11	-12.29	422.84	92.72
ixx	0.46	-8.39	-10.86	93.92	80.48
jack	1.67	-0.23	-7.10	-24.69	83.18
javac	3.87	-4.12	-6.13	73.27	349.14
jess	4.45	-3.81	-5.65	70.92	629.97
mpegaudio	1.19	-0.98	-9.73	-3.32	240.65
mtrt	6.16	-3.01	-5.37	40.12	2793.96
richards	29.59	-17.32	-19.39	88.60	160.15

All numbers are relative to the base model. *IPC* is the percent change in instructions per cycle. *Loads to L1* is the percent change in load instructions that access the L1 data cache. *Addr Trans* is the percent change in the number of address translations. *Avg SB* is the variation in the average number of active entries in the store buffer. *SB stall %* is the variation in the percentage of cycles with a store buffer stall.

The store buffer stall figure, Figure 5.6, shows the percent increase in store buffer stall percentage. The store buffer stall percentage is the number of cycles with a store buffer stall divided by the total number of cycles, $\frac{stallcycles}{allcycles}$. Most of the benchmarks expectedly increase their store buffer stall percentage when the store buffer policies are enhanced. Note that `eqn` sees a reduction in store buffer stalls when increasing its store buffer occupancy.

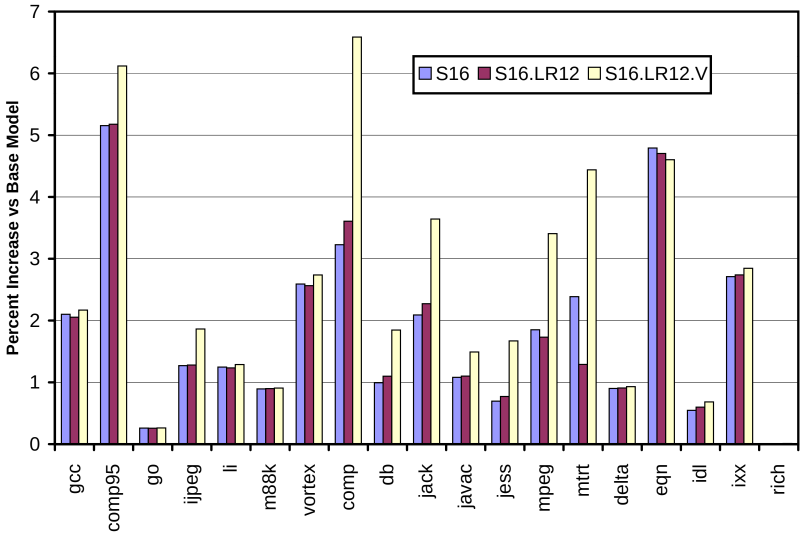


Figure 5.6: Change in Store Buffer Stalls for a 16-entry Store Buffer

`S16` is a 16-entry store buffer. `S16.LR12` is a 16-entry store buffer with lazy store removal at a threshold of 12 entries. `S16.V` is a 16-entry virtual store buffer. `S16.LR12.V` is a virtual 16-entry store buffer with lazy store removal at 12 active entries. These percentages are relative to the 32-entry, physical base model.

5.1.3 Best and Worst Behavior

This section briefly discusses the benchmarks that exhibit the best and worst performance increase in the optimal store buffer scheme. The C++ program `richards` achieves a 33.6% IPC increase with the lazy removal virtual store buffer versus the base model. One attribute that distinguishes `richards` from the other benchmarks is that it has one of the highest percentage of load instructions, 28.20% . Therefore, when the load traffic to L1 is reduced by 27.5%, a larger percentage of the total instructions in the program are being improved. Lazy store removal increases the average number of active entries in the store buffer and the potential for a store buffer stall. If the memory accesses are ordered in such a way that store buffer stalls are rare, it only helps these policies. A low percentage of store buffer stall cycles (almost 0%) indicates that this may be the case with `richards` (Table 5.1).

The C program `m88ksim` is least affected by the store buffer improvements. The best IPC increase obtained is 0.54%. Tables 4.1 and 4.2 show that `m88ksim` has the lowest percentage of load instructions and a high percentage of store buffer stall cycles in the base model. This is the opposite of `richards`. In addition, while `richards` enjoys a 27% decrease in data cache load traffic, `m88ksim` decreases this traffic by 11.9%.

5.2 Virtual Store Queue Analysis

5.2.1 Results from an Isolated VSQ

First, the VSQ is analyzed as a stand-alone component for initial studies. These experiments are run independent of the entire microprocessor core driven by memory references alone. Stores are placed in the VSQ using one of three methods: least recently used (LRU), random, or first-in first-out (FIFO). Load instructions examine the entire contents of the VSQ. If a virtual address of a store in the VSQ matches the virtual address of the load, the load is considered to hit in the VSQ. The LRU and random replacement methods are only tested with a 32-entry buffer. For the FIFO model, VSQ sizes of eight, 16, and 32 are considered.

Table 5.3 shows the percentage of loads that hit in the VSQ for the various configurations. A small eight-entry FIFO virtual store queue can reduce the load traffic to the memory hierarchy by 20.7% on average. A store buffer would have to maintain an average occupancy of eight active entries to match this forwarding rate. The improvement in hit rate due to increased buffer size varies from application to application (e.g. `compress` versus `m88ksim`).

Table 5.3: Percentage of Loads Removed by VSQ

	FIFO			LRU	RND
Benchmark	8-entry	16-entry	32-entry	32-entry	32-entry
compress95	24.22	35.44	43.63	51.07	41.59
gcc	17.02	24.02	31.24	34.89	28.44
go	18.95	26.13	34.48	36.17	31.43
ijpeg	4.96	5.69	7.10	7.26	8.27
li	16.44	21.84	25.61	27.07	24.30
m88ksim	38.93	40.27	41.52	41.69	42.14
perl	18.67	23.10	28.06	29.49	26.35
vortex	11.87	15.67	21.68	38.48	21.05
deltablue	11.77	16.15	19.22	20.99	17.90
eqn	22.20	27.85	32.63	37.68	32.31
idl	35.31	36.30	37.31	37.64	35.64
ixx	19.41	24.98	29.29	30.01	26.96
richards	27.92	37.65	46.90	57.43	46.36
compress	27.46	29.77	32.55	34.57	32.41
db	18.55	22.57	26.81	30.11	25.87
jack	26.27	28.15	33.93	34.66	33.20
javac	18.62	22.41	26.66	28.37	25.95
jess	17.12	21.66	25.98	28.23	24.98
mpegaudio	26.78	30.64	32.49	34.36	32.13
mtrt	15.48	20.92	24.89	26.32	24.49

Except in the case of `mpegaudio`, the LRU replacement scheme provides improvement over FIFO. This improvement varies from less than 1% for `ixx` to 17% for `vortex`. A random replacement scheme provides worse performance than both FIFO and LRU replacement schemes. The 32-entry LRU configuration provides the highest hit rate across the benchmarks, eliminating 33.3% of the load instructions on average. In some cases, this mechanism has the potential to reduce load traffic to the L1 data cache by one-half.

5.2.2 Results for a Full Processor with a VSQ

The 32-entry VSQ is examined in the context of a full microprocessor to evaluate its impact on instruction throughput and memory bandwidth. The VSQ is implemented as a content-addressable FIFO queue.

The reduction in memory bandwidth and port utilization is shown in Figure 5.7. Like the store buffer policies, the VSQ is able to significantly reduce the pressure on the memory hierarchy, allowing for a higher-performing more power-efficient architecture. The average reduction in load traffic to the L1 data cache is 27.1% and the average reduction in port utilization is 16.9%. It is fair to say that the VSQ is a more effective forwarding mechanism than the store buffer. The lazy store removal reduced load traffic by less than half that percentage.

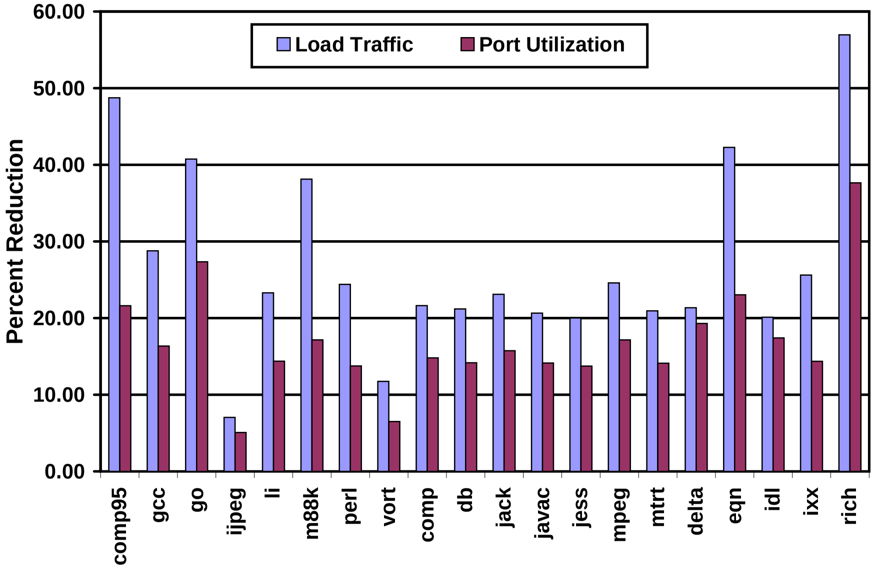


Figure 5.7: Percent Reduction in Load Traffic and Port Utilization

The percent increase in IPC is presented in Figure 5.8. Although the VSQ seems to be performing better than a virtual store buffer with lazy removal, the same performance gains are not seen. The average IPC improvement is 1.46%. This is primarily due to the more restrictive modeling discussed in Chapter 3.

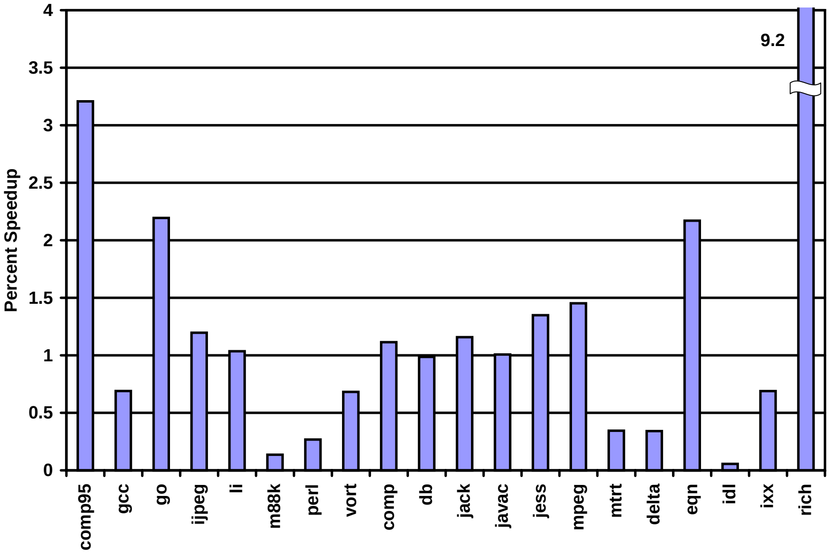


Figure 5.8: Percent Increase in IPC Using a 32-entry VSQ

Chapter 6

Solution for Wide-Issue Processors

6.1 Load Tokens

The instruction trace cache has been shown to increase the instruction bandwidth, allowing issue rates previously not attainable [43, 45, 49]. As instruction fetch techniques improve, aggressive issue and execution techniques can be employed to exploit the available instruction stream. Data speculation has been presented as one method to increase processor throughput. Value prediction is a way to relax some dataflow restrictions [20, 32, 33, 50]. Through confidence mechanisms, history tables and value buffers, many of these techniques are able to increase the performance of out-of-order processors. Similar to some forms of value prediction, memory renaming has also been shown to improve performance [30, 40, 57]. However, to achieve significant performance, large, complex data structures and quick recovery from misspeculation are required.

As an additional improvement to the VSQ, loads can access the VSQ speculatively. Unlike other speculation-based memory techniques, the decisions are governed by dynamically created load tokens supplied by the trace cache instead of a series of tables and buffers. These relatively small load tokens determine the confidence levels per static instruction and pinpoint the location of useful data within the VSQ.

6.1.1 Design of VSQ with Load Token Support

Each load is a candidate for a token. A load instruction whose effective address matches an entry in the virtual store queue is given a token. The status of the token could affect subsequent executions of the instruction when it is accessed from the trace cache. In this study, the token is a two-bit, saturating confidence counter and a six-bit offset. The counter is incremented each time a load instruction finds its data at the same VSQ offset as the previous access. When the counter is zero, it is always incremented on a VSQ hit. The counter is decremented when no hit occurs. A load from the trace cache may speculatively access the VSQ at the offset when the counter is in its maximum state.

The offset is the distance from the tail of the VSQ to the VSQ entry that contains the matching virtual address. An offset allows for a quick, direct mapped access to the VSQ when the load requests this value. A correct speculation allows future instructions that are dependent on the load to proceed when they otherwise would not be able. Although this method of speculation does not predict all types of load-store dependencies, it comes at a low hardware cost. The only enhancements needed to add speculation to the VSQ are additional fill unit logic and load tokens.

Tokens may be stored in a token table, separate from the trace cache, that allows N tokens per trace cache line. This method also requires one bit per instruction in the trace cache line to indicate which instructions have tokens. A second option is to provide each instruction slot in each trace cache entry with token storage. The first option uses less area and number of transistors. On the other hand, the second option does not restrict the number of loads that have tokens and allows all instructions to receive tokens.

6.1.2 Role of the Fill Unit

Figure 6.1 illustrates how the VSQ may fit into a trace cache environment. When a static instruction is encountered for the first time, it must be fetched from the instruction cache. As the instructions are issued, the fill unit collects and formats the instructions into trace cache lines. A trace cache line is held in the fill unit until all of the load instructions in the line compute their addresses. Each load informs the fill unit if it hit in the VSQ. If a hit occurs, the VSQ offset is communicated as well. At this point, the fill unit can place the formatted line into the trace cache (often times replacing the previous version of the line).

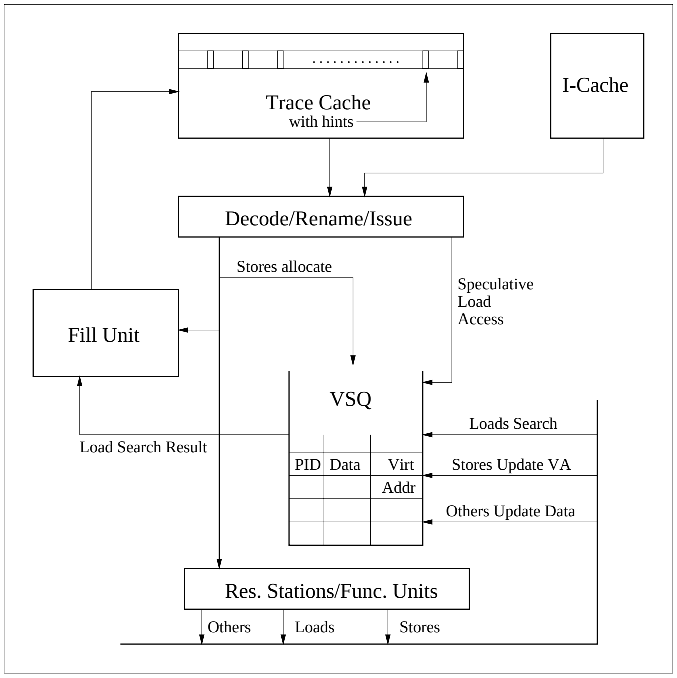


Figure 6.1: A trace cache processor augmented with the virtual store queue

When a load instruction is fetched via the trace cache, the related token information is passed to the fill unit. The fill unit can later combine the newest information from the load instruction with the previous state to create a revised token with a new confidence level and possibly a new offset. The fill unit may then write the trace cache entry with the revised token into the trace cache. All loads, including loads satisfied speculatively by the VSQ, eventually calculate their addresses and do a VSQ lookup.

The fill unit may begin to overflow if a load takes an extended number of cycles to calculate its address, because in the meantime instructions are continuously being issued. The fill unit has the option to place the line in the trace cache with no tokens, or to simply discard the line. This decision can be triggered by a "full" fill-unit or by a threshold on the time period. For simulation purposes, the fill unit is permitted to hold a line indefinitely. Later, the requirements of such a fill unit are analyzed.

6.1.3 Handling Incorrect VSQ Accesses

It is possible that a load speculatively acquires the wrong value from the virtual store queue. Determining this requires checking the calculated address against the address of the store entry that provided the data. When a load instruction speculatively loads from the VSQ, the virtual address of the corresponding VSQ entry is carried along with the load to a field in the reservation station entry of the load/store units. If this address matches the actual address then speculation was successful, otherwise the processor must recover.

There are two common strategies to recover from value misspeculations. In the most straightforward approach, the processor can *squash* all instructions that follow the offending load and then re-fetch and re-execute these instructions. This is the same strategy used for branch mispredictions. Additional checkpoints must be added for each load. Other than that, little additional hardware or data paths are necessary.

Selective re-execution is another option. This requires a more complicated hardware commitment but reduces the recovery time. In this scenario, only instructions that are in the dependency chain of the misspeculated load should be re-executed. Tyson and Austin found that only one-third of all instructions following a misspeculated load depend on that value. The tradeoffs and implementation of these two methods are further discussed in [32, 57].

6.2 Analyzing the Load Tokens

6.2.1 Wide-Issue Microprocessor Model

The trace cache implementation is based largely on the discussions in [43]. A fill unit collects instructions at issue time as in [43, 49]. Upon completing a trace cache line (also called a trace cache entry), the fill unit can then write the formatted trace cache line into the trace cache. Several additions to this basic trace cache architecture are required: the virtual store queue, load tokens, additional fill unit logic, and load speculation recovery.

Many trace cache studies work with unconstrained hardware resources, or optimistic resources such as 16 all-purpose functional units [15, 17, 43, 49]. Twenty specialized functional units are used. Their distribution is biased towards the integer-dominated benchmarks.

The fetch engine is implemented in a very similar manner to [15]. The level one instruction cache requires one more cycle to access than the trace cache. Fill unit optimizations, branch promotion, trace packing, next-trace prediction, and inactive issue are not implemented [14, 15, 24, 42, 43, 49]. The branch predictor is a hybrid predictor consisting of a gshare predictor with 15-bit history and a two-bit bimodal predictor [36]. All memory operations are allowed one cycle for memory disambiguation and/or address translation. The physically addressed store buffer allows load forwarding upon disambiguation and address translation.

Table 6.1: Simulated wide-issue architecture parameters

Data memory	
· L1 Data Cache:	4-way, 64KB, 1-cycle access
· L2 Unified cache:	4-way, 4MB, +9 cycles
· Non-blocking	4 MSHRs and 4 ports
· D-TLB	512-entry, 1-cycle hit, 30-cycle miss
· Store buffer:	64-entry w/load forwarding loads access in 1-cycle
· Main Memory	Infinite, +20 cycles

Fetch Engine	
· Trace cache:	4-way, 2K entry, 1-cycle hit partial matching, no path assoc.
· L1 Instr cache:	4-way, 4KB, 2-cycle hit one basic block per access
· Branch Predictor:	16k gshare/bimodal hybrid predictor 3-cycle misprediction penalty
· Branch target buffer	Perfect

Execution Core

· Functional unit	#	exec. lat.	issue lat.
Load/store	4	1 cycle	1 cycle
Simple Integer	8	1	1
Int. Mul/Div	3	3/20	1/19
Simple FP	3	3	1
FP Mul/Div/Sqrt	2	3/12/24	1/12/24

· 256-entry reorder buffer
· 12 reservation station entries/func. unit
· Fetch width: 16
· Decode width: 16
· Issue width: 16
· Execute width: 16
· Retire width: 16

6.3 Load Token Results

6.3.1 Impact of Virtual Store Queue

The maximum potential performance of the 64-entry VSQ is presented in Figure 6.2. This is called the ideal VSQ (Ideal64). This VSQ is able to identify all loads that will hit in the VSQ and performs early load forwarding. This is equivalent to full coverage on all VSQ hits and 100% speculation success. The overall potential performance improvement of the 64-entry ideal VSQ is 15.2%, and is benchmark specific, ranging from 0.2% to as much as 162.0%.

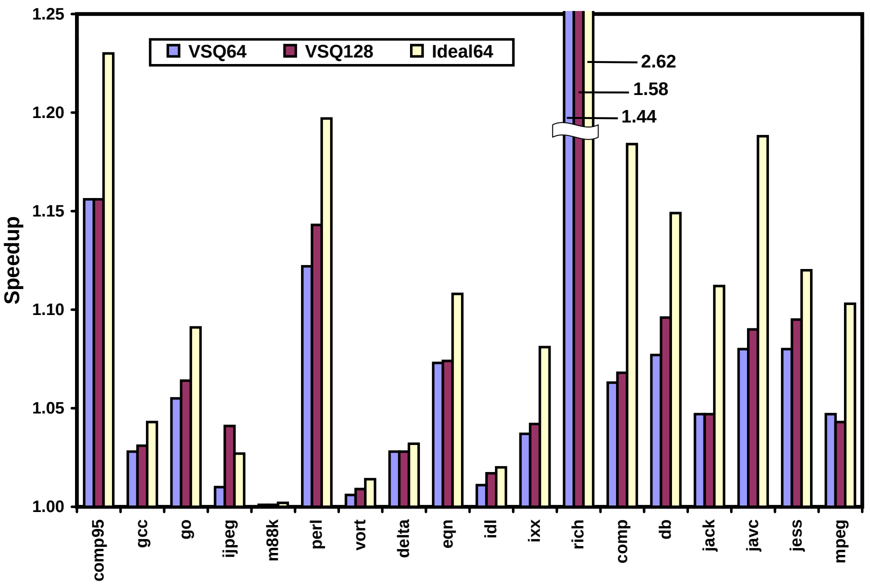


Figure 6.2: Impact of virtual store queue
VSQ64 is the base model with a 64-entry VSQ. VSQ128 has a 128-entry VSQ. Ideal64 has a VSQ with full coverage and perfect speculation for all VSQ hits.

The first non-ideal VSQ configuration examined in Figure 6.2 is the base model with a 64-entry virtual store queue but no speculative token-based loading (VSQ64). In this case, a VSQ hit saves the time required for address disambiguation in the store buffer and data cache access, but requires one cycle to access the VSQ. The addition of a 64-entry VSQ to the base model with no load tokens yields a 7.2% overall performance gain. The performance gains stem from the ability to bypass the memory system more often than in the base model. However, the exact amount of improvement cannot be directly tied to any one characteristic, but to a combination of several factors.

The percentage of load instructions in a program is the first important factor from Table 6.2 (`% Loads`). Programs with a large percentage of loads will be more sensitive to the effects of the VSQ. For example, the two programs with the lowest percentage of loads, `m88ksim` and `ixx` show very little performance gain, if any.

Table 6.2: Runtime characteristics of memory operations

	Base Model			Tokens, Rex3		
Benchmark	% Loads	% Stores	% Load forward	VSQ Hit %	VSQ Coverage %	VSQ Pred %
compress95	17.7	15.0	25.9	47.78	17.25	90.52
gcc	18.6	9.9	16.4	34.40	11.48	92.86
go	21.2	7.2	12.1	42.86	7.46	90.75
ijpeg	17.0	6.5	3.7	15.00	5.88	97.76
m88ksim	15.2	8.5	35.4	43.29	36.12	99.40
perl	20.5	10.7	17.6	32.99	15.50	96.14
vortex	18.7	9.5	11.1	19.78	10.19	98.21
deltablue	25.4	6.2	6.5	21.08	17.65	99.58
eqn	17.3	9.4	23.1	39.28	20.12	94.97
idl	21.8	2.7	8.5	24.07	12.17	98.95
ixx	15.2	7.7	19.0	32.99	17.53	98.48
richards	27.5	8.2	11.5	44.09	21.08	98.01
compress	31.7	10.0	23.4	30.40	12.07	95.37
db	21.6	7.6	13.9	26.28	12.61	98.22
jack	28.8	9.7	22.9	30.11	15.23	98.75
javac	22.7	7.9	14.2	26.48	13.51	97.10
jess	23.8	8.3	13.5	26.19	11.21	96.52
mpegaudio	31.0	9.4	23.6	29.84	13.02	97.90

`% Loads` is the percentage of all instructions that are loads. `% Stores` is the percentage of all instructions that are stores. `% Loads forward` is the percentage of loads that have their values forwarded from the store buffer. `VSQ Hit %` is the percentage of all decoded loads that hit in the VSQ upon calculating their address. `VSQ Coverage %` is the percentage of all loads that load speculatively from the VSQ. `VSQ Pred %` is the percentage of predicted loads that are predicted correctly.

Another factor is the amount of load forwarding present in the base model (`% Load forward` in Table 6.2). Load forwarding from the store buffer leads to many of the same benefits as VSQ hits. If the load forwarding percentage is comparable to the percentage of loads that hit in the VSQ (`VSQ Hit %`), then the VSQ has not served any additional purpose. For example, the program with the biggest difference, `richards`, benefits most from the VSQ and the program with the smallest difference, `m88ksim`, benefits the least.

Past research showed that load operations can be optimally satisfied at varying rates, i.e. some are more critical than others [54]. The VSQ may be eliminating a large percentage of loads, but if these loads happen to be non-critical then the impact will not be reflected in the performance. This is probably the case in a benchmark like `idl` which contains 21.8% loads and improves significantly on the load forwarding rate, yet does not respond well to the VSQ.

When the VSQ is doubled in size to 128 (`VSQ128`) entries, the additional overall performance gain is less than 1.5%. Only a small percentage of additional load instructions find their data in the VSQ.

6.3.2 Impact of Load Tokens

Although there is significant performance improvement using the 64-entry VSQ, only half of the ideal performance gain is reached. The remaining available performance can be achieved through speculation. This thesis examines two configurations which allow for load token based speculative loads. In the first case, squash recovery is used (`Tokens, Sq8`) and in the second case, selective re-execution is used (`Tokens, Rex3`). The load token misspeculation penalty when using squash recovery is eight cycles. The misspeculation penalty using selective re-execution is three cycles. The performance impact using these policies is illustrated in Figure 6.3.

Adding speculation with squash recovery does not increase the performance significantly. In some of the benchmarks there is performance degradation versus the non-speculative VSQ (`VSQ64` in Figure 6.2) and even versus the base model in some cases. Not surprisingly, speculation using the quicker selective recovery provides more improvement over a non-speculative VSQ implementation. With this method of recovery, overall performance increases by 10.4% over the base model.

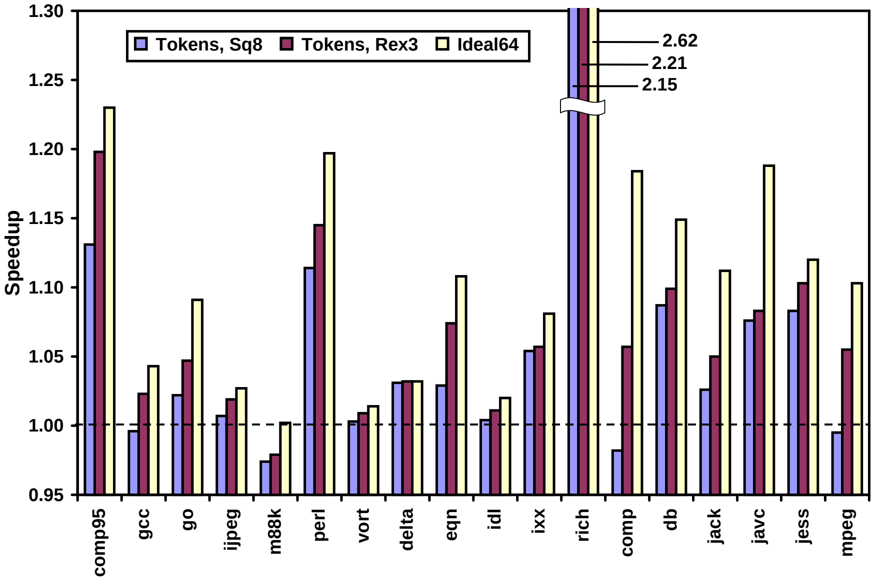


Figure 6.3: Impact of load tokens

Tokens, Sq8 has a VSQ, load tokens, and an 8-cycle squash recovery. Tokens, Rex3 has a VSQ, load tokens, and 3-cycle selective re-execution. Ideal64 has a VSQ with full coverage and perfect speculation for all VSQ hits.

This is an improvement over the non-speculative VSQ implementation, but does not reach the overall ideal performance of 15.2%. For speculation to be worthwhile, the cumulative rewards of correct speculation have to be far greater than the penalties of misspeculation. Although it has been found that many correctly speculated values may not directly enhance performance, as fetch bandwidth increases the rewards for proper speculations should increase [17]. Furthermore, in a processor with multiple stages between decode and execute, the reward for proper speculation may be higher than with this short pipeline.

Table 6.2 and Figure 6.4 provide a more in-depth look at the VSQ and token-based speculation. VSQ Hit % is the percentage of all decoded loads that find a matching store in the virtual store queue. On average, 31.3% of all load traffic can be eliminated using the VSQ. The next two columns concern speculation. The *VSQ Coverage %* is the percentage of all loads that have enough confidence to perform a speculative VSQ access. The final column is the prediction rate of those loads which do a speculative VSQ access.

The percentage of loads covered is approximately the percentage of loads that hit in the VSQ. If a load does not hit in the VSQ at some point, it will never speculatively load from the VSQ. Overall load coverage ranges from only 7.2% to 36.2%. This means most loads in the programs are left unspeculated. However, of the loads that are speculated, a high percentage are speculated properly, over 90% in all cases and an average of 96.64%. Although the tokens do not allow for many loads to be satisfied speculatively from the VSQ, they cause relatively few mispredictions.

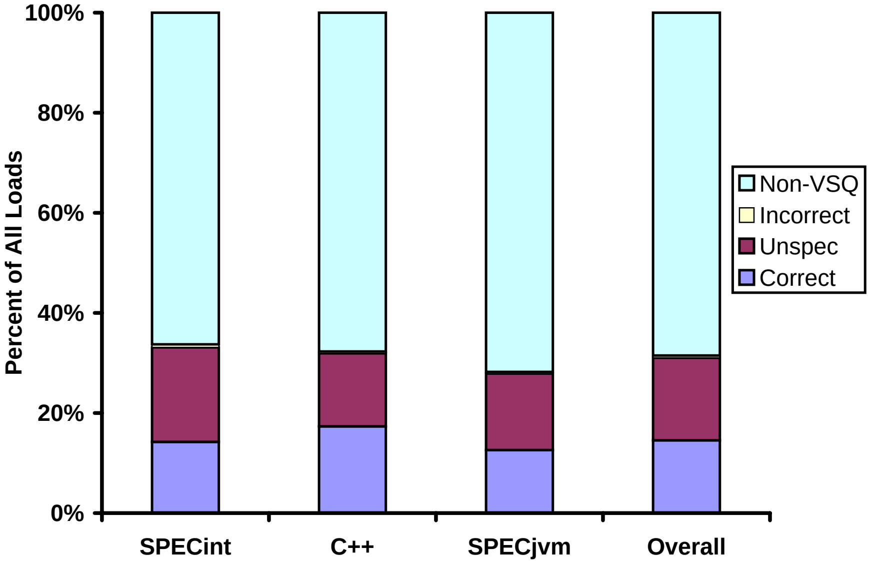


Figure 6.4: Breakdown of loads when using load tokens

`Non-VSQ` are loads not affected by the VSQ. `Unspeculated` are VSQ hits that didn't speculate. `Correct` are VSQ hits that properly speculated. `Incorrect` are loads that misspeculated.

6.3.3 Impact of Memory Latency on VSQ Performance

Multiple-cycle L1 cache accesses are becoming common. The potential benefit provided by the virtual store queue when the access time for a hit is increased from one cycle to three cycles is presented in Figure 6.5. In this model, L1 misses take an additional two cycles and L2 misses take an additional 10 cycles compared to the base model. To offset some of the latency penalty, the size of the L1 data cache is also increased from 64KB to 128KB.

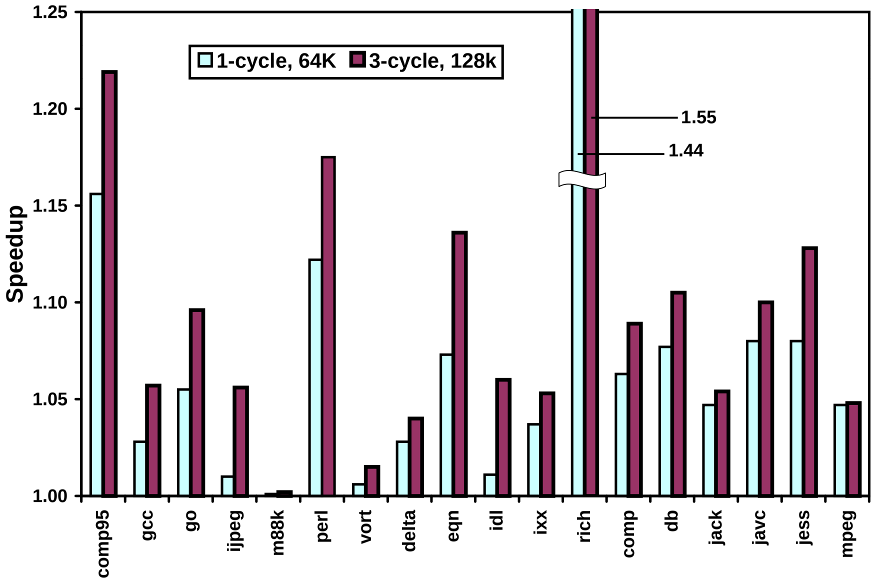


Figure 6.5: Impact of memory latency on 64-entry VSQ performance
The VSQ speedups for the two cache models are relative to two different base models. `1-cycle, 64K` uses the normal 1-cycle, 64KB L1 data cache for both the base model and VSQ64 model. `3-cycle, 128K` uses a 3-cycle, 128KB L1 data cache and longer latency L2 for the base model and VSQ64 model.

Using the new data cache parameters, the effects of the VSQ are more significant. The percent improvement over a base model with the same 3-cycle, 128 KB data cache is 10.5% while the same size VSQ provides a 7.2% improvement in the original model. These results indicate that the VSQ might be an appropriate mechanism for processors with multiple-cycle data caches.

6.4 Other Design Issues

Table 6.3 provides some indication of the efficiency and feasibility of the VSQ, load tokens, and supporting logic. The first column shows the percentage of VSQ entries that never satisfy a load during their lifetime in the VSQ. These can be thought of as unused VSQ entries. On average, 44.87% of VSQ entries go unused. A more judicious manner of adding stores to the queue could increase the performance with the same amount of resources.

Table 6.3: Design issue analysis

Benchmark	Unused VSQ	Load Addr. Delay	Delay Traffic	Lines w/tokens	Tokens/Line
compress95	58.1	5.51	36.74	30.3	1.56
gcc	59.5	8.60	45.61	23.1	1.63
go	40.0	8.66	46.98	23.3	1.56
ijpeg	69.7	24.65	70.19	11.0	1.58
m88ksim	50.3	13.62	77.23	24.1	2.70
perl	53.3	9.04	55.40	30.7	1.37
vortex	80.5	6.94	46.76	19.0	1.35
deltablue	39.8	99.68	99.37	24.3	1.53
eqn	57.4	6.67	39.92	23.5	1.89
idl	46.5	10.37	69.20	20.1	1.31
ixx	53.5	10.04	54.74	19.3	1.40
richards	29.9	24.78	125.41	44.4	1.24
compress	17.9	23.74	100.87	49.1	1.33
db	40.0	17.97	77.67	27.6	1.35
jack	21.4	26.65	121.44	46.9	1.21
javac	37.6	21.82	85.30	31.3	1.32
jess	38.7	22.48	85.02	29.9	1.31
mpegaudio	13.5	16.55	88.14	39.4	1.62

`Unused VSQ` is the percentage of VSQ entries that never forward data to a load. `Load Addr. Delay` is the average time in cycles that a load requires to calculate its address. `Lines w/tokens` is the percentage of trace cache lines gathered that contain at least one token. `Tokens/Line` is the average number of tokens per trace cache line for lines that have at least one token.

The next metric is the average number of cycles that a load requires to calculate its address, or *load address delay*. This indicates the average number of cycles the fill unit requires to buffer instructions before placing them in the trace cache. The *delay traffic* is the average number of instructions that are issued during the load address delay. The fill unit requires storage to buffer instructions while it packages them for trace cache placement. The delay traffic shows the buffer size magnitude that may be needed. Further studies are required to find the best heuristics and policies that balance performance and realistic fill unit expectations.

Lines w/tokens is the percentage of trace cache entries collected in the fill unit that contain at least one token. For lines with tokens, the average number of tokens per line is reflected in the last column of the table. The percentage of lines with tokens is important. Whenever a new token is written into a trace cache line, it is best to place that line back into the trace cache no matter what. Normally, many trace lines do not make it from the fill unit to the trace cache because a similar line is already present. Therefore, this metric shows the possibility for extra trace cache write traffic.

For the VSQ architecture, the last column is useful in the token storage implementation decision. Since there are on average only 1.75 tokens per line, why keep storage space for 16 tokens per line? The complimentary structure discussed earlier, with $N = 4$, would suffice.

Chapter 7

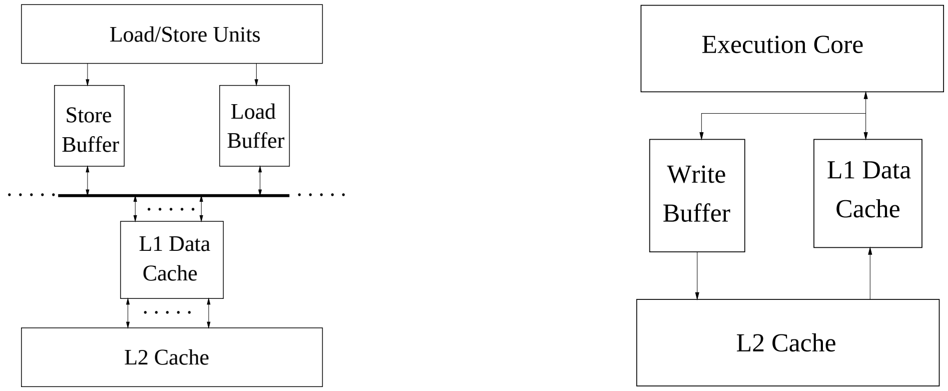
Related Work

Previously, there has been extensive research targeting the same basic load-store relationship. These works are similar in that they all take advantage of the relationship between dependent memory access instructions located close together in the dynamic instruction stream.

7.1 Basic Memory Techniques

Johnson provides an in-depth discussion of a store buffer which holds stores that conflict with a load for a data cache interface [26]. The store buffer maintains the ordering of the stores and allows stores to be performed only after all previous instructions (including loads) have been completed. Loads are allowed to bypass stores and data forwarding is performed when appropriate. Data forwarding allows a load instruction to "load" its data from a store instruction located in the store buffer. The alternative is to allow the store to complete and then load the value from memory. Finally, loads are performed in order with respect to other loads, for simplicity. Johnson's store buffer is our base model. Additional work in memory ordering has been performed by McKee et al. [38, 39].

A load instruction receiving data from a uncommitted store instruction is called *load forwarding* or *data forwarding*. Studies have found that due to conditions, such as register spilling, stores are often closely followed by a load to the same address as the store. Some research uses speculative methods to try to get maximum use out of this relationship [30, 40, 57]. Other studies use a buffer, much like a store buffer, to forward the store data[25, 34]. We will attempt to allow the store buffer to exploit this fact while maintaining its memory ordering and latency hiding functionality.



a. Store Buffer from Thesis b. Write Buffer from Literature

Figure 7.1: Write buffer versus Store buffer

One structure that is similar to the store buffer is the write buffer [35, 51, 53]. Our model is a store buffer as in Figure 7.1.a while most of the write buffer literature assumes a structure similar to the one in Figure 7.1.b. These write buffers are accessed in parallel with the on-chip cache and have the ability to combine several stores with contiguous addresses or the same address. Skadron and Clark discuss the issues and tradeoffs involved in such a write buffer [53]. Martonosi and Shaw did a study of the effect of compilation techniques on the performance of a write buffer [35]. Jouppi [29] and Bray [5] consider structures they call write caches with similar properties. The issues addressed in these papers and similar papers focus on reducing the number of writes that are performed off-chip and sometimes on-chip. It is possible that mechanisms like the write buffer can be referred to as a store buffer [51].

Of course, there are many works that encouraged and influenced both my study and the above studies. Johnson talks about data forwarding where load instructions can load from a store in the store buffer instead of waiting for the store to proceed and then loading from that address [26]. Austin and Sohi outlined an early pipeline, fast-address calculation scheme, exploiting the use of a precomputed base and immediate offset to achieve the quick calculation [2]. Gallagher et al. [18] and Franklin and Sohi [13] proposed methods for recovering from true memory-dependencies when addresses are computed out-of-order. Lozano and Gao discuss a data forwarding scheme to exploit the short life-span of many variables involving compiler analysis and a simple architectural extension [34]. The study and exploitation of memory access characteristics has also been published [27, 48], including a selective bypassing of the data cache scheme by Tyson et al. [56].

7.2 Memory Speculation

Studies have found that due to conditions such as register spilling, stores are often closely followed by a load to the same address as the store. Some research uses speculative methods to try to get maximum use out of this relationship [30, 40, 57]. Other studies use a buffer, much like a store buffer, to forward the store data [25, 34]. This thesis enables the store buffer to exploit this fact via load forwarding while maintaining its memory-ordering and latency-hiding functionality.

Moshovos and Sohi propose using *data dependence prediction* to link loads and stores in the instruction stream [40]. They use data dependence identifying when a load and store can communicate speculatively, eliminating the address calculation, disambiguation, and data cache access, and also to identify when a load and store may be eliminated totally if they are encountered in the proper sequence. They also present a Transient Value Cache (TVC) which stores intermediate memory addresses and relies on the data dependence prediction for management. In their studies they use a 256 entry fully-associative TVC. They do not indicate the resources required for their assumption of "perfect dependence and dependence status prediction within the 256 most recent stores." Also, they assume perfect memory disambiguation in all cases. They report a potential performance increase ranging from 5% to 15%.

Tyson and Austin propose a technique called *memory renaming*, which is similar to the above TVC [57]. Memory renaming combines dependence prediction with value prediction. They use a store-load cache to store the speculative values of the memory addresses, where tightly coupled loads and stores access the same entry in the cache. All of their memory renaming experiments were performed with a 1024 entry, 2-way set-associative store/load cache and a 512 entry value file with LRU replacement. They find that memory renaming can be applied to 30% to 50% of all memory references resulting in a 41% improvement in execution time. The memory renaming work is extended by the Loadmark Architecture. Reinman et al. use profiling and compiler analysis to provide support to the memory renaming architecture and find that the benefits of memory renaming can be further improved [47].

Jourdan et al. [30] introduced a "one value, one location" approach where a memory renamer and physical renamer are combined to provide unification of data value storage. They also suggest predicting values and addresses in an attempt to eliminate identical data values from occupying more than one register. Their research concentrates on the X86 architecture. Among the many optimizations suggested in this paper, their speculative techniques also perform memory-dependency collapsing. Finally, they suggest that a store buffer is no longer required in such an environment and that on average 32% of load memory traffic can be eliminated in a 128-deep processor.

7.3 Commercial Implementations

Commercial processors have been implementing store buffers or similar ideas for many years. Although in-depth analysis of performance tradeoffs are not readily available, there are some indications of the type of policies that are being currently implemented. Words in quotations indicate processor specific terminology.

The Alpha 21264 microprocessor has a 32-entry *speculative store buffer* where a store remains until it is “retired”. A store must first enter the speculative store buffer before its data is sent to the level-one cache. Stores forward their data to loads when they are in the speculative store buffer [31].

The Sun UltraSPARC-IIi processor contains a load/store unit (LSU). The LSU is responsible for calculating load and store virtual addresses as well as “decoupling” loads and stores from the pipeline by using both a load buffer and a store buffer. The pipelines are not fully decoupled so that the UltraSPARC-IIi can support precise traps. Stores in the store buffer normally have a lower priority than loads in the load buffer, but the CPU will eventually raise the priority when a “lock-out condition” is reached. There is no mention of the ordering of the loads and stores or of the possibility of load forwarding. Finally, the LSU allows stores to be combined if they have been marked with a “write-gathering attribute,” but this is not done automatically (as it would be in a write buffer) [41].

The Pentium III processor is said to have twelve store buffers, where each store buffer can temporarily hold a store to memory. This is essentially one twelve-entry store buffer. It allows other instructions to continue executing while the stores are waiting to be efficiently written to memory. These stores can forward data to waiting reads. The P6 architecture contains a memory reorder buffer (MOB) as well. The MOB works with physical addresses and affects memory accesses that are going to the level-two cache [22, 23]. The AMD K6 has a store queue. Entries are placed into the store queue with their physical address while a cache access is being attempted [52].

7.4 Wide-Issue Memory Approaches

Using the fill unit to improve the performance of trace cache lines is first introduced by Friendly, Patel and Patt in [15]. This work focuses on compiler-like optimizations that could be performed dynamically in the fill-unit within a trace cache entry. The fill unit determines optimizations that can be performed non-speculatively and then rearranges and/or rewrites the instructions accordingly. These optimizations are not present in our simulated architecture, but could exist in the VSQ architecture. Tokens could potentially aid these optimizations.

The memory renaming and data dependence prediction studies appeared concurrently and are not trace cache related. They deal mainly with address memory dependence speculation while the VSQ deals with non-speculative memory dependences as well. Compared to the tokenized VSQ, they are larger, more complex, but more robust methods for speculating on related memory operations. These methods could be complimented or improved by the load tokens..

Another scheme that incorporates the trace cache to reduce the impact of loads is a scheme from Gabbay and Mendelson [17]. They make the very interesting discovery that the potential of value prediction can best be exploited in high bandwidth instruction-fetch architectures. They propose a complicated but fast distribution network to communicate predicted values directly to the instructions in the traces. This requires a highly-interleaved prediction table, a trace addresses buffer, an address router, a value distributer, a hybrid value predictor [16], and pipelined access of the prediction table. This study takes liberties with the resources to prove a very interesting point which we apply to a constrained resource environment.

John et al. propose a *code coalescing unit* (CCU) that works in conjunction with the SRRB [25]. The data and addresses from stores are buffered in the SRRB. This buffer then forwards the data to subsequent loads. This buffer along with the CCU facilitates the removal and/or renaming of loads and stores that are tightly coupled. They report that 24% to 42% of all load operations could be eliminated with these mechanisms in place. This study does not report any cycle-level, full-processor simulation-based performance increase potential, although they suggest that it can be significant, especially in the register-limited X86 architecture. They also suggest that the CCU in conjunction with a trace cache on an X86 processor should provide increasing performance gains. Though performance impact is not studied, they suggest that for optimal performance the CCU should be used to rewrite instructions non-speculatively into a trace cache with assistance from compiler hints. This thesis opts for a hardware-only method with speculation.

Chapter 8

Conclusion

This thesis makes contributions to the understanding of dependent store/load pairs and potential designs to reduce their impact. Using an advanced simulation environment, a wide variety of modern workloads are studied, including Java, C++, and C programs. Characteristics of memory operations including dependent store/load pairs and transient memory values are revisited and expanded upon. Microarchitecture proposals for dynamically scheduled out-of-order microprocessors based on these characterizations are evaluated. The proposed techniques include advanced store buffer design and *virtual store queues* for modern microprocessors. For future wide-issue processors, the implementation of a speculative *load token* method is described and studied. This work provides design insight and options for microarchitects interested in reducing the negative impact on performance due to transient memory values and corresponding store/load pairs.

8.1 Memory Characterization

Characterizing the behavior of workloads is important in understanding changes in microprocessor performance. This thesis first studies general memory characteristics. In addition to a basic characterization, dynamic memory metrics such as the average latency of memory operations, port utilization, store buffer occupancy, and store distances are analyzed.

Load instructions represent 21.3% of the dynamic instruction stream, significantly more than store instructions. In general, there are twice as many load instructions as store instructions. However, port utilization studies show that, due to the arbitration policy, load instructions spend fewer cycles waiting for memory resources.. An examination of store buffer occupancy shows that, on average, 3.09 of the 16 store buffer entries are active each cycle. This low occupancy leaves room for improvement in load forwarding schemes.

Somewhat surprisingly, many store/load pairs are only a few instructions apart in the dynamic instruction stream. About 50% of all forwarding stores in the C++ benchmarks are less than four instructions from a dependent load. This percentage is approximately 41% for the SPECint benchmarks, and 25% for the SPECjvm benchmarks. Also, 19.8% of stores write to memory but never have their data read by a load instruction in the same program. Learning how to differentiate between bad and good stores could improve performance of specialized memory dependence mechanisms.

This analysis reveals that there is ample opportunity to reduce the latency of load instructions. This is important because memory reads are critical to overall instruction throughput. Chains of dependent instructions often start with the loading of a value from memory. Therefore, reducing the latency of load instructions enables groups of instructions to proceed more quickly.

8.2 Store Buffer Design

Due to a lack of literature on the details of the store buffer, the first portion of this thesis delves into the issues involved in designing a store buffer for a dynamically scheduled, out-of-order processor. These store buffer issues include size, store removal policy, store retirement point, store priority switching, and virtual store buffers.

- Incorporating a lazy store removal policy alone substantially increases the amount of load forwarding that takes place, yet does not greatly increase the number of store buffer stalls in a 32-entry store buffer. This increase in load forwarding reduces by 12% the number of load instructions that access memory. This leads to a performance improvement (in IPC) ranging from 0.15% to 6.9%. A 16-entry store buffer with this policy can approach and in some cases surpass the performance of a 32-entry store buffer. This policy has less effect on store buffers of four and eight entries.

- Switching from the base model to a virtual store buffer model improves performance by reducing the number of address translations that take place before useful memory access work can be performed. Forwarded loads now avoid the address translation latency. The IPC increases by an average of 4.1% in this case.

- By incorporating both lazy store removal and making the store buffer virtual, the IPC of the processor can increase by an average of 5.11% over all benchmarks for a store buffer of size 32 and by as much as 33% in specific cases. On average, a 16-entry store buffer with these policies outperforms a normal 32-entry store buffer. Even an eight-entry store buffer outperforms a 32-entry store buffer for certain benchmarks. Four- and eight-entry store buffers with this implementation, on average, approach but do not exceed the next larger size studied.

There are, of course, many combinations of policies, configurations, and parameters that were not explored due to time considerations. It is possible that some other combination of store removal, store priorities, and/or the store retirement threshold could create slightly better performance. However, these results convey that there are many store buffer design decisions to make and the subsequent impact on performance is not trivial.

8.3 Virtual Store Queue

This thesis proposes a low-resource, simple mechanism for exploiting the relatively short memory dependence distance between a store to an address and a load from that same address, the Virtual Store Queue (VSQ). The load-store access characteristics of a benchmark suite are evaluated to determine the best approach for implementing the VSQ. The results on the Sun UltraSPARC platform were very similar to those found on other platforms. Many loads and stores are generated just for the purpose of passing *transient values.* It is common for a store to be followed by a load or a store to the same address. The VSQ is a mechanism to exploit the former.

The implementation of the VSQ behaves like a forwarding buffer, forwarding the data from a store to a load at the same address. When this happens, there is no longer a need for address translation and memory access. The VSQ is non-speculative unlike some other recently proposed schemes that approach this phenomenon. Therefore, the resource requirements and logic are dramatically less in the absence of large prediction tables. The buffer uses virtual addresses to avoid address translation and possible associated penalties. Techniques are discussed to avoid the problems associated with using virtual addresses.

- A small eight-entry FIFO virtual store queue can reduce the load traffic to the memory hierarchy by 20.7% on average. A 32-entry LRU can eliminate 33.3% of the L1 load accesses. In some cases, this configuration can reduce load traffic to the L1 data cache by one-half.

- With a 32-entry VSQ, a relatively aggressive, out-of-order processor can reduce load traffic to the L1 data cache by 27.1% and reduce port utilization by 16.9%. At the same time, the duty of handling load forwarding is taken on by the VSQ instead of the store buffer and memory control logic. This provides a marginal improvement in IPC for a four-wide machine.

- While the performance of the VSQ falls slightly short of the performance of some speculation-based methods for memory dependence collapsing, the non-trivial performance increase and lower cost of implementation makes the VSQ an attractive method.

8.4 Wide-Issue Solutions

By using the VSQ, the performance of a 16-wide trace cache processor can be increased by 7.2% overall as a result of the bypassing of the data cache by load instructions. To reduce the impact of data dependencies caused by load instructions, *load tokens* trigger an early speculative access from the VSQ. This provides an additional 3.2% overall performance improvement (a total of 10.4% improvement over the base model) at the expense of some additional trace cache memory but little new logic and no new structures.

In addition to the addressing of potential design issues of the trace cache and fill unit, the effects of recovery methods, VSQ size and data cache properties are analyzed. Using load tokens with three-cycle, selective re-execution recovery provides additional improvement to the VSQ, while eight-cycle squash recovery is not sufficient for implementing token-based speculative VSQ accesses. A 128-entry VSQ does not provide significant improvement over a 64-entry VSQ. A VSQ also has a larger impact in a processor with a multiple-cycle data cache, achieving 10.5% improvement versus 7.2% with a single-cycle data cache. Finally, although the techniques proposed are effective, they have still not exploited all the potential performance gain of the ideal VSQ. Based on this analysis, the VSQ could have a large impact on performance in deeply pipelined, high frequency trace cache processors of the future.

Bibliography

[1] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus ipc: The end of the road for conventional microarchitectures. In *27th International Symposium on Computer Architecture*, pages 248–259, Jun 2000.

[2] T. M. Austin and G. S. Sohi. Zero-cycle loads: Microarchitecture support for reducing load latency. In *28th Annual International Symposium on Microarchitecture*, pages 82–92, November 1995.

[3] R. Bhargava and L. K. John. Issues in the design of store buffers in dynamically scheduled processors. In *2000 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 75–87, Apr. 2000.

[4] R. Bhargava, L. K. John, and F. Mathus. Accurately modeling speculative instruction fetching in trace-driven simulation. In *Proc of Int. Performance, Computing, and Communications Conference*, pages 65–71, Phoenix, AZ, Feb 1999.

[5] B. K. Bray. *Specialized Caches to Improve Data Access Performance.* PhD thesis, Stanford University, May 1993.

[6] D. Burger and T. Austin. The simplescalar tool set, version 2.0. Technical report, University of Wisconsin, Madison, WI, 1997.

[7] D. Burger, J. R. Goodman, and A. Kagi. Memory bandwidth limitations of future microprocessors. In *23rd International Symposium on Computer Architecture*, pages 78–89, May 1996.

[8] B. Calder, D. Grunwald, and B. Zorn. Quantifying behavioral differences between c and c++ programs. Technical Report CU-CS-698-94, University of Colorado, Boulder, Jan 1994.

[9] M. Charney and T. Puzak. Prefetching and memory system behavior of the spec95 benchmark suite. *IBM Journal of Research and Development*, 41(3):265–286, May 1997.

[10] R. F. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. Technical Report SMLI 93-12 and UWCSE 93-06-06, Sun Microsystems Laboratories, Incorporated, and the University of Washington, 1993.

[11] H. G. Cragon. *Memory Systems and Pipelined Processors.* Jones and Bartlett Publishers, 1996.

[12] K. Driesen and U. Holzle. The direct cost of virtual function calls in c++. In *OOPSLA-96*, pages 306–323, San Jose, Calif., Oct 1996.

[13] M. Frankline and G. Sohi. A hardware mechanism for dynamic reordering of memory references. *IEEE Transactions on Computers*, C-45(5):552–571, May 1996.

[14] D. H. Friendly, S. J. Patel, and Y. N. Patt. Alternative fetch and issue policies for the trace cache fetch mechanism. In *30th International Symposium on Microarchitecture*, pages 24–33, Dec. 1997.

[15] D. H. Friendly, S. J. Patel, and Y. N. Patt. Putting the fill unit to work: Dynamic optimizations for trace cache processors. In *31st International Symposium on Microarchitecture*, Dallas, TX, November 1998. to appear.

[16] F. Gabbay and A. Mendelson. Can program profiling support value prediction? In *30th International Symposium on Microarchitecture*, pages 270–280, Dec 1997.

[17] F. Gabbay and A. Mendelson. The effect of instruction fetch bandwidth on value prediction. In *25th International Symposium on Computer Architecture*, pages 272–281, June 1998.

[18] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. W. Hwu. Dynamic memory disambiguation using the memory conflict buffer. In *6th International Conference on Architectural Support for Programming Languages and Operating S ystems*, pages 183–193, Oct. 1994.

[19] J. Gonzalez and A. Gonzalez. Speculative execution via address prediction and data prefetching. In *International Conference on Supercomputing*, pages 196–203, July 1997.

[20] J. Gonzalez and A. Gonzalez. The potential of data value speculation. In *12th International Conference on Supercomputing*, 1998.

[21] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufman, CA, 1996.

[22] Intel Corporation. *Intel Architecture Software Developer's Manual*, 1997.

[23] Intel Corporation. *Intel Architecture Optimization Reference Manual*, Feb 1999.

[24] Q. Jacobson, E. Rotenberg, and J. E. Smith. Path-based next trace prediction. In *30th International Symposium on Microarchitecture*, pages 14–23, Dec. 1997.

[25] L. K. John, Y. Teh, F. Matus, and C. Chase. Improving memory access performance using a code coalescing unit. In *International Conference on Computer Design*, pages 550–557, Austin, TX, Oct. 1998.

[26] M. Johnson. *Superscalar Microprocessor Design.* Prentice Hall, 1990.

[27] T. L. Johnson and W.-M. W. Hwu. Run-time adaptive cache hierarchy management via reference analysis. In *Proc. of Int. Sym. on Computer Architecture*, June 1997.

[28] G. P. Jones and N. P. Topham. A comparison of data prefetching on an access decoupled and superscalar machine. In *30th International Symposium on Microarchitecture*, pages 65–70, Dec 1997.

[29] N. P. Jouppi. Cache write policies and performance. In *Proc. 20th Intl. Sym. on Computer Architecture*, pages 191–201, May 1993.

[30] S. Jourdan, R. Ronen, M. Beckerman, B. Shomar, and A. Yoaz. A novel renaming scheme to exploit value temporal locality through physical register reuse and unification. In *31st Int. Symp. on Microarchitecture*, pages 216–225, Dallas, TX, November 1998.

[31] R. E. Kessler, E. J. McLellan, and D. A. Webb. The alpha 21264 microprocessor architecture. In *Proc. of the Intl. Conf. on Computer Design*, pages 90–95, Austin, TX, Oct 1998.

[32] M. H. Lipasti and J. P. Shen. Exceeding the dataflow limit via value prediction. In *29th International Symposium on Microarchitectures*, pages 226–237, Dec 1996.

[33] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *17th International Conference on Architectural Support for Programming Languages and Operating Systmes*, pages 138–147, Oct 1996.

[34] L. A. Lozano and G. R. Gao. Exploiting short-lived variables in superscalar processors. In *Proceddings of 28th Int. Symp. on Microarchitecture*, pages 292–302, 1995.

[35] M. Martonosi and K. Shaw. Interactions between application write performance and compilation techniques: A preliminary view. In *Proc. Workshop on Interaction between Compilers and Computer Architectures*, pages II.1–6, Feb 1997.

[36] S. McFarling. Combining branch predictors. Technical Report TN-36, Digital Western Research Labs, Palo Alto, Calif., Jun 1993.

[37] S. A. McKee. Dynamic access ordering : Bounds on memory bandwidth. Technical Report CS-94-14, University of Virginia, April 1994.

[38] S. A. McKee, R. H. Klenke, A. J. Schwab, W. A. Wulf, S. A. Moyer, J. H. Aylor, and C. Y. Hitchcock. Experimental implementation of dyanamic access ordering. In *Proc. of 27th Hawaii Int. Conf. on Systems Sciences (HICSS-27)*, Jan 1994.

[39] S. A. McKee and W. A. Wulf. Access order and memory-conscious cache utilization. In *Proc. Of First Sym. on High Performance Computer Architecture (HPCA-1)*, Jan 1995.

[40] A. Moshovos and G. S. Sohi. Streamlining inter-operation memory communication via data dependence prediction. In *30th International Symposium on Microarchitecture*, pages 235–245, Dec. 1997.

[41] K. B. Normoyle, M. A. Csoppenszky, A. Tzeng, T. P. Johnson, C. D. Furman, and J. Mostoufi. Ultrasparc-iii: Expanding the boundaries of a system on a chip. *IEEE Micro*, pages 14–24, Mar/Apr 1998.

[42] S. J. Patel, M. Evers, and Y. N. Patt. Improving trace cache effectiveness with branch promotion and trace packing. In *25th International Symposium on Computer Architecture*, pages 262–271, June 1998.

[43] S. J. Patel, D. H. Friendly, and Y. N. Patt. Critical issues regarding the trace cache fetch mechanism. Technical report, University of Michigan, 1997.

[44] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, , and K. Yelick. A case for intelligent RAM. *IEEE Micro*, 17(2):34–44, Mar/Apr 1997.

[45] A. Peleg and U. Weiser. Dynamic flow instruction cache memory organized around trace segments independent of virtual address line. U.S. Patent Number 5,381,533, 1994.

[46] S. S. Pinter and A. Yoaz. Tango: a hardware-based data prefetching technique for superscalar processors. In *29th International Symposium on Microarchitecture*, pages 214–225, Dec 1996.

[47] G. Reinman, B. Calder, D. Tullsen, G. Tyson, and T. Austin. Classifying load and store instructions for memory renaming. In *International Conference on Supercomputing*, pages 399–407, June 1999.

[48] J. A. Rivers and E. S. Davidson. Reducing conflicts in direct-mapped caches with a temporality-based design. In *1996 International Conference on Parallel Processing*, pages 154–163, August 1996.

[49] E. Rotenberg, S. Bennett, and J. E. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. *Proc. 29th Intl. Sym. on Microarchitecture*, pages 24–34, Dec. 1996.

[50] Y. Sazeides and J. E. Smith. The predictability of data values. In *30th International Symposium on Microarchitecture*, pages 248–258, Dec 1997.

[51] L. Schaelicke and A. Davis. Improving i/o performance with a conditional store buffer. In *Proc. of Intl. Sym. on Microarchitecture*, pages 160–169, Dec 1998.

[52] B. Shriver and B. Smith. *The Anatomy of a High-Performance Microprocessor: A Systems Perspective.* IEEE Computer Society Press, 1998.

[53] K. Skadron and D. W. Clark. Design issues and tradeoffs for write buffers. In *Proc. of Third Int. Symp. on High-Performance Computer Architecture*, pages 144–155, February 1997.

[54] S. Srinivasan and A. Lebeck. Load latency tolerance in dynamcially scheduled processors. In *31st International Symposium on Microarchitecture*, pages 148–159, Nov 1998.

[55] Standard Performance Evaluation Corporation. Spec benchmarks. *http://www.spec.org/*.

[56] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun. A new approach to cache management. In *Proc. of 28th Int. Symp. on Microarchitecture*, November 1995.

[57] G. S. Tyson and T. M. Austin. Improving the accuracy and performance of memory communications through renaming. In *30th International Symposium on Microarchitecture*, pages 218–227, Dec. 1997.

[58] D. L. Weaver and T. Germond. *The SPARC Architecture Manual (Version 9).* Sparc International, Englewood Cliffs, NJ, USA, 1995.

[59] B. Wheeler and B. N. Bershad. Consistency management for virtually indexed caches. In *Proc. Fifth Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 124–136, Oct 1992.

Vita

Ravindra (Ravi) Nath Bhargava was born in Akron, Ohio on April 10, 1975 to T.N. and Christine Bhargava of Kent, Ohio. He lived in Kent until graduating from Kent Theodore Roosevelt High School in June of 1993. The following August he entered Duke University in Durham, North Carolina. In May 1997, he graduated from the Engineering School with a B.S.E. in Electrical Engineering. The following fall, he began classes at the Graduate School at The University of Texas at Austin. He has gained industry experience through summer internships at Advanced Micro Devices in Austin (1999) and Intel Corporation in Austin (2000). In April 2000, Ravi became a Ph.D. candidate in the Electrical and Computer Engineering Department at The University of Texas at Austin.

Permanent Address: 606 West Lynn Street #10
Austin, Texas 78703

This thesis was typed by the author.