ISSUES IN THE DESIGN OF STORE BUFFERS IN DYNAMICALLY SCHEDULED PROCESSORS

Ravi Bhargava and Lizy K. John

The University of Texas at Austin
Laboratory for Computer Architecture
{ravib,ljohn}@ece.utexas.edu

Abstract

Processor performance can be sensitive to load-store ordering, memory bandwidth, and memory access latency. A store buffer is a mechanism that exists in many current processors to accomplish one or more of the following: store access ordering, latency hiding, and data forwarding. Different policies that govern store buffer behavior can affect overall processor performance. However, the performance impact of various store buffer policies is not clearly analyzed in available literature. In this paper, we look into various store buffer issues such as i) where to place it in the pipeline, ii) when to remove a store entry from the store buffer, iii) when to allow the stores to be retired, and iv) if, when, and how to set the contention priority of memory operations. These issues are explained in detail while design and performance tradeoffs are assessed. Using a variety of C, C++, and Java benchmarks, we establish how these design policies influence performance. We find that the policies for store entry removal and store buffer pipeline placement have a large effect on the overall performance of a microprocessor. In addition, we see that smaller, well-designed store buffers can achieve comparable performance to larger, basic store buffers. Combining these results with an analysis of the benchmarks can help one fully understand the role of the store buffer and the tradeoffs involved.

1 Introduction

Techniques exist to hide or tolerate the latency of loads and stores. Executing instructions in an out-of-order manner helps hide latency. Compilers can assist by performing loads well before the value is required by other instructions. 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. Despite these and similar techniques, memory access latency is still considered one of the primary bottlenecks in processors today.

Almost all modern processors allow dynamic ordering of load and store instructions. Non-blocking caches and buffering structures such as write buffers, store buffers, store queues, and load queues are typically employed. We find that 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, we examine 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. The challenge is to achieve this performance win without increasing store buffer related stalls (i.e. stalling the pipeline since no new stores can be issued) or other detrimental effects. We find that the lazy store removal scheme can have a large impact on processor performance. In addition, we see that smaller, well-designed store buffers can achieve comparable performance to larger, basic store buffers.

The paper is divided into the following sections. Section 2 provides more background on store buffers and the issues we are addressing. Section 3 discusses the store buffer policies and parameters that we study. Section 4 discusses our methodology and benchmarks. Section 5 discusses the results from different store buffer configurations. Section 6 offers a summary and conclusions.

2 Background

2.1 Related Work

Johnson provides an in-depth discussion of a store buffer which holds stores that conflict with a load for a data cache interface [10]. 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 [17, 18].

One structure that is similar to the store buffer is the write buffer [15, 21, 23]. Our model is a store buffer as in Figure 1 while most of the write buffer literature assumes a structure similar to the one in Figure 2. 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 [23]. Martonosi and Shaw did a study of the effect of compilation techniques on the performance of a write buffer [15]. Jouppi [11] and Bray [2] 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 [21].

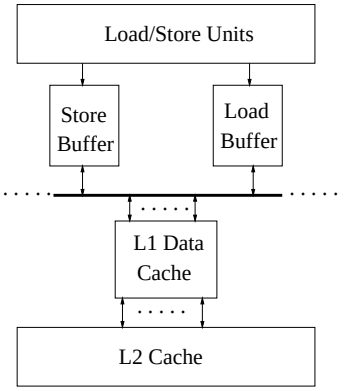

Figure 1: Store buffer model from this paper

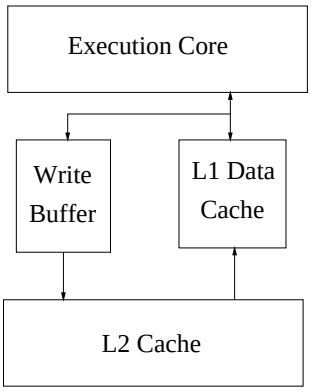

Figure 2: Write buffer model from literature

Due to conditions such as register spills, stores are often closely followed by a load to the same address as the store. Some research use speculation to get maximum usage out of this relationship [12, 19, 26]. Other studies use a buffer, much like a store buffer, to forward the store data [9, 14]. We study methods to allow the store buffer to exploit this property via load forwarding while maintaining the memory-ordering and latency-hiding functionality.

2.2 Current 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 [13].

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) [20].

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 [7, 8].

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 [22].

2.3 Role of Cache Design

The configuration of the data cache has a relevant role in the design of the store buffer. The store buffer and the cache may be affected by the same address translations. A machine with a virtually indexed and physically tagged cache (UltraSPARC-IIi, Alpha 21264, AMD K6) requires an address translation to take place in parallel with the cache access. This does not directly indicate whether the store buffer is physical or

virtual. A store instruction can be translated for the cache access, but due to resource constraints never accesses the cache. This store is placed with its physical address into the store buffer. For example, the K6 store buffer acquires the store instruction's address after the translation [22].

For our study, we assume a physically indexed and physically tagged data cache like that found in the Pentium and AMD K7 architectures. In this style of cache, the address translation takes place prior to the access of the cache [5]. Once again, this type of cache does not imply the use of a virtual or physical store buffer.

3 Store Buffer Policies

Although the store buffer's purpose appears straightforward, there are several interesting design decisions to be made. In this section, we look into several possible policies that govern the behavior of the store buffer. Included in our discussion are the policies of *store priority switching*, *store removal*, and the *store retirement point*, which are related, but deserve separate discussions. A *virtual store buffer* is also explained. note that we refer to the term *active entry* as any entry in the store buffer that is in use and currently contains a store in any state.

Store Retirement Point The retirement point of a store refers to the point in the life of a store instruction where 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 is reached. This is similar to the concept of a *highwater mark* in write buffer design. *Delayed retirement* helps by increasing 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 will determine the effectiveness.

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. We also need to indicate with tags which stores have retired but are still active in the store buffer. Another problem with building up active entries is the possible increase in disambiguation time.

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 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? One, the load is given a higher priority, or *loads first*. Two, the store is given priority, or *stores first*. Three, the oldest instruction is given priority, or *oldest first*.

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.

Virtual Store Buffer The data in store buffers may be accessed by loads before or after the address translation stage of the store. 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, saving the address translation cycles. 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.

Aliasing, or the synonym problem, is an important issue when using virtual addresses to tag the data in a store buffer. Although this problem is infrequent, it does need to be dealt with. These types of problems have been tackled in virtually indexed caches. Software and hardware solutions exist [5, 9, 28]. The UltraSPARC-IIi simply says that "software handles aliasing" with respect to its virtually indexed cache. For a virtual store buffer, a hardware solution would be required. Using virtual addresses requires an easy method to distinguish unique processes and then include this *ID* as part of the address tag. This is common in some of the 64-bit RISC processors, but quite

a challenge in the x86 family of processors. So, the Intel Pentium Pro MOB and the AMD K6 store buffer exclusively use physical addresses.

4 Evaluation Methodology

4.1 Microprocessor Model

To analyze the impact of the store buffer in a dynamically scheduled out-of-order processor environment, we use a detailed, execution-driven, cycle-level, timing simulator that models all resource contention as well as speculative execution. Shade, a simulation tool from Sun [4], is the front-end of the simulator. It takes any SPARC executable (source code not necessary) as input and then drives the execution core with a dynamic stream of instructions. Therefore, the simulator uses the SPARC instruction set architecture [27] and handles the SPARC nuances in a proper fashion, e.g. register windows, conditional instructions, condition code registers, and delay slots. Our model is a four-wide machine, i.e. four-wide issue, decode, execute, and retire. These and other specific parameters may be found in Table 1.

Table 1: Simulated Architecture Parameters

Data memory

· L1 Data Cache:	4-way, 64KB, 1-cycle access
· L2 Unified cache:	4-way, 1MB, +7 cycles
· Non-blocking	2 MSHRs and 1 port
· D-TLB	128-entry, 1-cycle hit, 30-cycle miss
· Store buffer:	32-entry w/load forwarding loads access in 1-cycle
· 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
· Branch target buffer	Perfect
· I-TLB	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

· 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 simulator uses a separate 64-entry reorder buffer (ROB) and register file for floating point and integer instructions respectively, as in the UltraSPARC. Stores are allocated entries in the ROB due to reasons discussed by Johnson [10]. The branch predictor uses the *gshare* prediction scheme as described by McFarling [16]. When there is a branch misprediction, instructions are fetched along the wrong path. These instructions are executed as accurately as possible based on the recent history of execution [1].

The cache hierarchy model is derived from `cachesim5` which is available with the Shade tool set. The L1 instruction cache and data cache are write-back, write-allocate caches with a block size of 32 bytes. They use the LRU replacement algorithm. The L1 data cache is physically indexed and physically tagged. The L2 cache is a unified, write-back, write-allocate cache with a block size of 64 bytes, and uses the LRU replacement scheme. The data cache can handle up to two outstanding requests due to the presence of two miss status holding registers (MSHR).

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.

4.2 Benchmarks

For our study, we conduct simulation experiments on Sun UltraSPARC machines. We use programs from three sets of benchmarks to evaluate the store buffer schemes. Descriptions of the benchmarks and the inputs we use are in Table 2.

Table 2: Benchmark Descriptions

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

The first set of benchmarks is the SPEC95 integer suite [25]. These commonly used C benchmarks are a good point of reference. The next set of benchmarks

is a C++ suite developed for the purpose of studying object-oriented workloads, specifically the effects of virtual functions on performance [3, 6]. These two suites of benchmarks are compiled with gcc 2.8.1. with full optimizations (-O4) and are statically linked. The final group of benchmarks are the Java benchmarks from SPEC [25]. 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. Some relevant numerical characteristics such as number of dynamic instructions, percentage of loads, and percentage of stores of each benchmark may be found in Table 3.

Table 3: Basic Characteristics of the Benchmarks

Benchmark	Dynamic Instr	% Loads	% Stores
gcc	261.1M	19.00	10.15
compress95	383.2M	17.95	15.20
go	477.6M	22.20	7.56
ijpeg	495.4M	17.19	6.54
li	166.0M	22.32	10.18
m88ksim	122.0M	15.61	8.70
vortex	494.7M	18.90	9.57
compress	496.4M	31.78	10.09
db	86.6M	20.97	7.91
jack	495.3M	28.86	9.77
javac	198.7M	22.55	8.10
jess	259.0M	23.68	8.54
mpegaudio	497.6M	30.98	9.48
mtrt	490.3M	26.06	9.52
deltablue	40.7M	25.70	6.29
eqn	47.1M	17.76	9.64
idl	82.8M	22.55	2.74
ixx	29.6M	15.53	7.93
richards	66.0M	28.20	8.38

Benchmarks are run until completion or until 500 million instructions are decoded. *Dynamic Instr* is the number of instructions executed dynamically (does not include SPARC annulled instructions or wrong path instructions).

We analyze several programs from each suite. Due to time considerations, not every program from each benchmark may be used, each program may not be run under every configuration, and programs are terminated if and when they reach 500 million instructions.

5 Results and Analysis

Several experiments were performed varying the individual store buffer policies for the base-line physical store buffer. 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 processor performance. 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.

The performance characteristics of the base model are presented in Table 4. We use the base model store buffer as a reference when ascertaining the impact of the store buffer policies.

5.1 Per Policy results

In Figure 3, 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. We examined different thresholds for these policies in prior simulations and found these to be the best of those examined.

Figure 4 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.

Impact of Lazy Store Removal. This policy of 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%.

Unfortunately, all of these saved loads do not translate directly to 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 MSHR's 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 [24]. It is possible that the loads that are avoided due to increased load forwarding are ones that can tolerate latency.

Table 4: Performance of the Base Model

Benchmark	IPC	% Forw Lds	Ld Hit Ratio	St Hit Ratio	SB Stall %	Avg SB	Addr Trans
cc1	2.28	11.87	98.7	97.9	0.43	5.81	77.6M
compress95	1.90	14.60	98.1	93.2	13.14	11.05	13.2M
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
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
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

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.

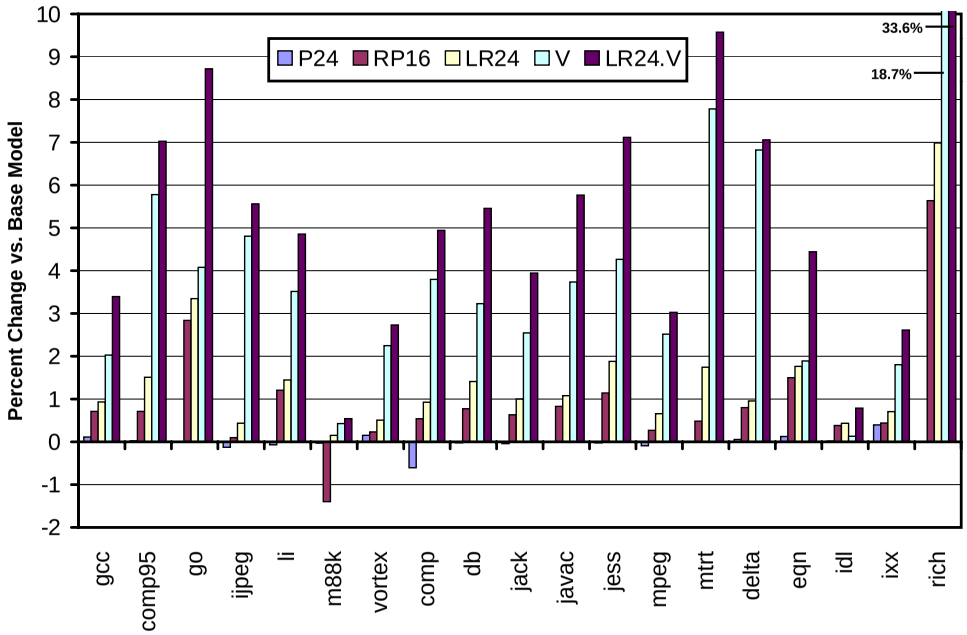

Figure 3: Effects of Isolated Store Buffer Policies on IPC

`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.

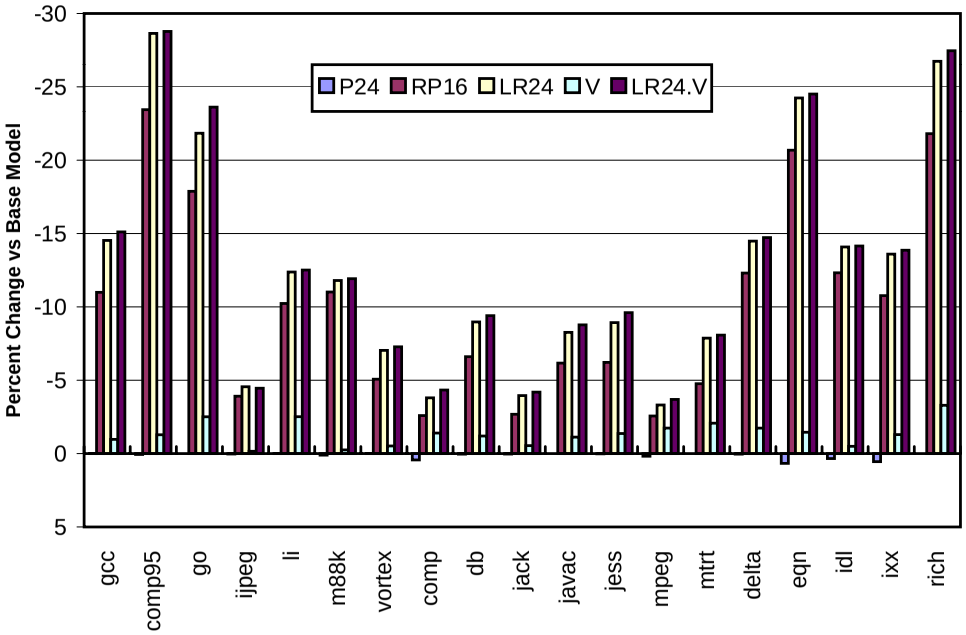

Figure 4: Effects of Isolated Store Buffer Policies on L1 Load Traffic

`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.

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 our 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 the 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 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 virtual address are sufficient to properly determine address dependencies.

For each load that is forwarded, the address translation latency is not incurred (our 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 they felt it was important 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 an average of 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.

5.2 Policies versus Size

This section investigates the effects of store buffer size with and without the improvement from the optimal policies. Figure 5 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 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. 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. The details of the 16-entry configuration with respect to the base model are examined later in Figures 6 and 7.

Figure 5 shows 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 8-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). 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.

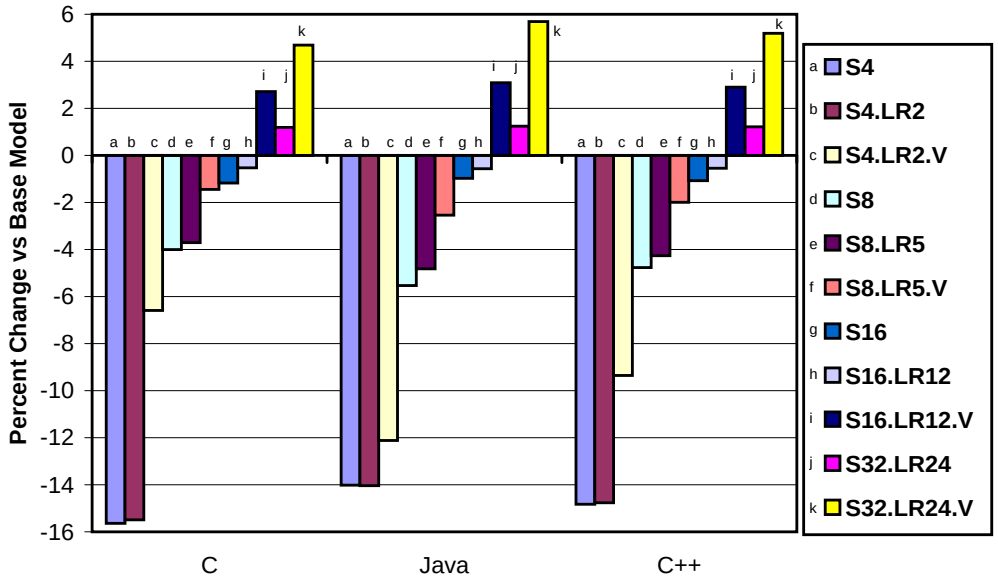

Figure 5: Effects of Store Buffer Size and Policy on IPC

`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.

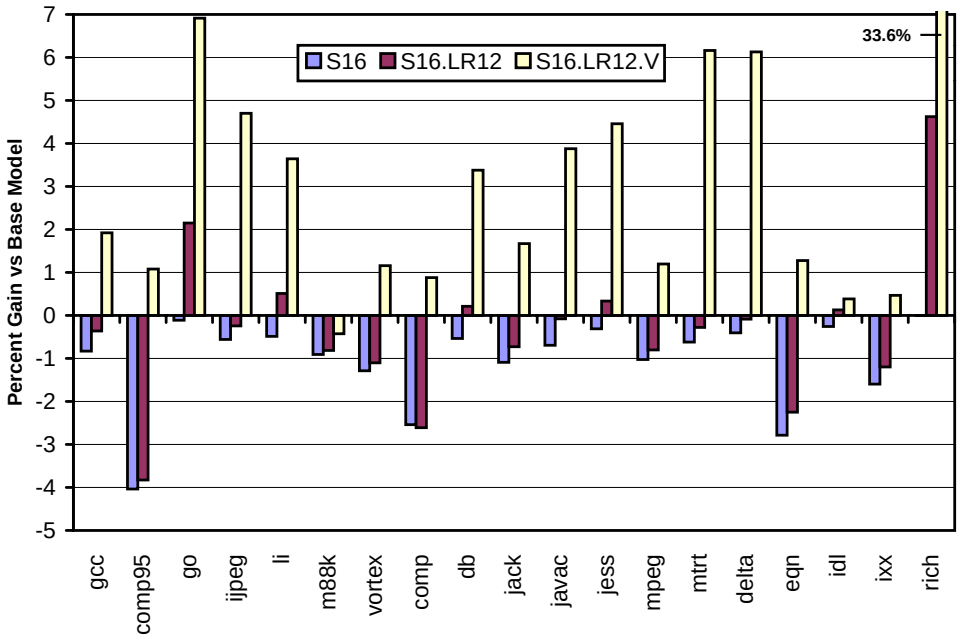

Figure 6: Change in IPC for a 16-entry Store Buffer

`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.

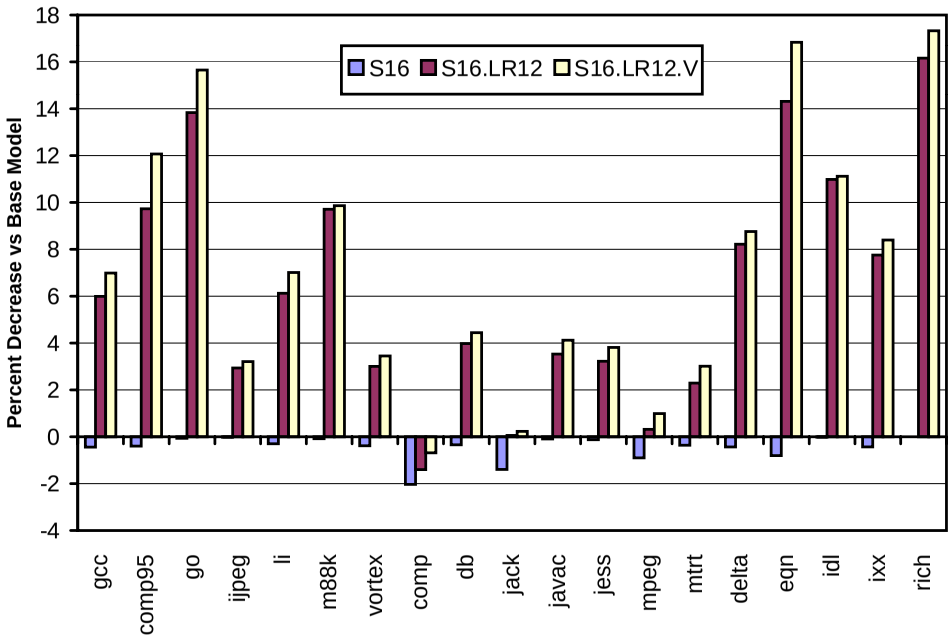

Figure 7: Change in Load traffic for a 16-entry Store Buffer

`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.

Figure 6 begins a closer look at the 16-entry configurations. The IPC numbers indicate that it is the virtual aspect of the 16-entry store buffer that increases performance more than lazy store removal. In Figure 7, 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.

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.

5.3 Best and Worst Behavior

We would like to briefly discuss the benchmarks that exhibit the best and worst performance increase in our optimal store buffer scheme. The C++ program `richards` achieves a 33.6% IPC increase with the lazy removal virtual store buffer versus our 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 4).

The C program `m88ksim` is least affected by the store buffer improvements. The best IPC increase obtained is 0.54%. Tables 3 and 4 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%.

Table 5: 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.

6 Conclusions

Due to a lack of literature on the details of the store buffer, we took this opportunity to delve 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.

• We find that 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 the number of loads that access memory by 12%. 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 both incorporating lazy store removal and making the store buffer virtual, we find that 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 we did not explore due to time considerations. It is possible that some other combination of store removal, store priorities, and a store retirement threshold could create slightly better performance. What this paper should convey to the reader is that there are many store buffer design decisions to make and the subsequent impact on performance is not trivial.

References

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

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

[3] 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.

[4] 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.

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

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

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

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

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

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

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

[12] 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 International Symposium on Microarchitecture*, pages 216–225, November 1998.

[13] R. E. Kessler, E. J. McLellan, and D. A. Webb. The alpha 21264 microprocessor architecture. In *Proc. International Conference on Computer Design*, pages 90–95, Oct 1998.

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

[15] 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.

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

[17] 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 dynamic access ordering. In *Proc. of 27th Hawaii International Conference on Systems Sciences (HICSS-27)*, Jan 1994.

[18] S. A. McKee and W. A. Wulf. Access order and memory-conscious cache utilization. In *Proc. Of First Symposium on High Performance Computer Architecture*, Jan 1995.

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

[20] 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.

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

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

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

[24] S. T. Srinivasan and A. R. Lebeck. Load latency tolerance in dynamically scheduled processors. In *Proc. International Symposium on Microarchitecture*, pages 148–159, Nov 1998.

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

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

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

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