January 2004

A High-Bandwidth Load-Store Unit for Single- and Multi-Threaded Processors

Amir Roth
University of Pennsylvania, amir@cis.upenn.edu

A High-Bandwidth Load-Store Unit for Single- and Multi-Threaded Processors

Abstract

A store queue (SQ) is a critical component of the load execution machinery. High ILP processors require high load execution bandwidth, but providing high bandwidth SQ access is difficult. Address banking, which works well for caches, conflicts with age-ordering which is required for the SQ and multi-porting exacerbates the latency of the associative searches that load execution requires.

In this paper, we present a new high-bandwidth load-store unit design that exploits the predictability of forwarding behavior. To start with, a simple predictor filters loads that are not likely to require forwarding from accessing the SQ enabling a reduction in the number of associative ports. A subset of the loads that do not access the SQ are re-executed prior to retirement to detect over-aggressive filtering and train the predictor. A novel adaptation of a Bloom filter keeps the re-execution subset minimal. Next, the same predictor filters stores that don't forward values to nearby loads from the SQ enabling a substantial capacity reduction. To enable this optimization and maintain in-order store retirement, we add a second SQ that contains all stores, but only to retirement and Bloom filter management; this queue is large but isn’t associatively searched. Finally, to boost both load and store filtering and to handle programs with heavy forwarding bandwidth requirements we add a second, address-banked forwarding structure that handles "easy" forwarding instances, leaving the globally-ordered SQ to handle only "tricky" cases. Our design does not directly address load queue scalability, but does dovetail with a recent proposal that also uses re-execution to tackle this issue.

Performance simulations on SPEC2000 and MediaBench benchmarks show that our design comes within 2% (7% in the worst case) of the performance of an ideal multi-ported SQ, using only a 16-entry queue with a single associative lookup port.

Keywords

computer science, processors

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-04-09.

A High-Bandwidth Load-Store Unit for Single- and Multi- Threaded Processors

Amir Roth
Department of Computer and Information Science, University of Pennsylvania
amir@cis.upenn.edu

Abstract

A store queue (SQ) is a critical component of the load execution machinery. High ILP processors require high load execution bandwidth, but providing high bandwidth SQ access is difficult. Address banking, which works well for caches, conflicts with age-ordering which is required for the SQ and multi-porting exacerbates the latency of the associative searches that load execution requires.

In this paper, we present a new high-bandwidth load-store unit design that exploits the predictability of forwarding behavior. To start with, a simple predictor filters loads that are not likely to require forwarding from accessing the SQ enabling a reduction in the number of associative ports. A subset of the loads that do not access the SQ are re-executed prior to retirement to detect over-aggressive filtering and train the predictor. A novel adaptation of a Bloom filter keeps the re-execution subset minimal. Next, the same predictor filters stores that don't forward values to nearby loads from the SQ enabling a substantial capacity reduction. To enable this optimization and maintain in-order store retirement, we add a second SQ that contains all stores, but only to retirement and Bloom filter management; this queue is large but isn't associatively searched. Finally, to boost both load and store filtering and to handle programs with heavy forwarding bandwidth requirements we add a second, address-banked forwarding structure that handles "easy" forwarding instances, leaving the globally-ordered SQ to handle only "tricky" cases. Our design does not directly address load queue scalability, but does dovetail with a recent proposal that also uses re-execution to tackle this issue.

Performance simulations on SPEC2000 and MediaBench benchmarks show that our design comes within 2% (7% in the worst case) of the performance of an ideal multi-ported SQ, using only a 16-entry queue with a single associative lookup port.

1. Introduction

In a dynamically scheduled processor, loads read values from one of two structures. Loads that read addresses written to by older in-flight stores forward values from a store queue (SQ). All other loads read their values from the data cache. Cache and SQ access are performed in parallel to minimize latency and reduce scheduling complexity. High ILP processors require high bandwidth, low latency load execution units. Providing high bandwidth access to data cache can be done in a relatively straightforward way using address-interleaving (banking). Providing high-bandwidth access to the SQ is more difficult [14] as address banking and age-ordering are not easily reconciled. High bandwidth SQ access is therefore typically provided by multi-porting, a costly proposition given that load access searches the SQ associatively. Replicating the SQ is another costly option. A separate, but not unrelated challenge is partitioning an SQ for multithreaded execution.

In this paper, we propose a new load-store unit design that exploits the following two observations. First, an SQ serves two functions: (i) it buffers speculative stores for in-order retirement to the cache, and (ii) it forwards values from in-flight stores to younger loads. The first function requires an SQ to hold all in-flight stores, i.e., to be large. The second requires it to be associatively searched, i.e., slow. Implementing both functions in a single structure combines the requirements and produces a structure that must be both large and slow. Second, even for forwarding purposes a queue structure is only necessary if either (i) load/store execution is naturally mis-ordered or (ii) multiple stores to the same address exist in flight simultaneously. When this is not the case, a simpler, faster structure suffices.

Per the first observation, we divide the two functionalities of a conventional SQ between two separate queues. For in-order retirement we use a large retirement store queue (RSQ) and enter all stores into it in the usual manner. Since the RSQ is not used to forward values, it does not come equipped with associative-search machinery, and in fact is removed from the timing critical load execution path. Per the second, we further divide value forwarding responsibilities between two structures. An address-interleaved forwarding store cache (FSC) provides high bandwidth forwarding for most loads. The FSC is a small associative cache with word-size blocks and FIFO replacement; each FSC bank fronts the corresponding data cache bank. All stores are also entered into the FSC, but only as their addresses become available. A small subset of stores and loads cannot correctly communicate via a mechanism as undisciplined as a an FSC. For these, we supply a conventional age-ordered, non-interleaved, fully-associative forwarding store queue (FSQ). The FSQ is essentially a conventional SQ, only in miniature. It requires fewer ports than a conventional SQ because only those loads that have previously failed in FSC access it. It requires many fewer entries because only those stores that write values required by these loads are allocated entries in it. Loads and stores are steered to FSC and FSQ using a simple predictor. One last piece is required to make the whole enterprise work. Both FSQ and FSC are speculatively accessed and, in some sense, also speculatively populated. We detect and recover from potential mis-forwardings (and non-forwardings) by re-executing a minimal subset of the loads immediately prior to retirement to verify the forwarded values. The identification of this minimal subset is performed using a novel adaptation of Bloom filtering [3]. The entire mechanism supports multithreading easily.

Our design does not address load queue scalability directly, but meshes well with one recent design that does, value based memory ordering (VBMO, our acronym) [4]. Both schemes rely on filtered load re-execution as an enabling mechanism. We show how the two schemes can be implemented together naturally, and that our Bloom filtering techniques improve on the initial VBMO design in two important ways.

Performance simulations on the SPEC2000 and MediaBench programs in both single-threaded and multi-threaded mode show that our new load-store unit achieves performance that is within 2% of that of an ideal two-ported associative SQ with infinite capacity, using only a queue with only a single associative port and 16 entries.

The next section provides background and motivation for our design, which is described in Section 3. Section 4 contains a simulation-driven evaluation of our scheme. Section 5 discusses related work.

2. Background

High ILP processors require high-bandwidth, low-latency load-store units. High ILP multithreaded processors require high bandwidth, low-latency load-store units that can be effectively partitioned among multiple threads. A typical load store unit has two major components: a data cache and a store queue (SQ). A data cache supports high bandwidth access and multi-thread partitioning in a straightforward way. High-bandwidth access is provided effectively via address-interleaving (i.e., banking). Dynamic, fine grain partitioning is achieved naturally for a physically-tagged cache and using simple, thread or process ID tag extensions for a virtually-tagged cache. Banking and partitioning an SQ, however, is more difficult.

Address-interleaving considerations. The SQ is logically age-ordered. In a centralized scheme, this ordering is enforced by allocating queue entries at dispatch, before addresses are known. In an address-interleaved scheme, age ordering (within a bank) is awkward to enforce. Bank assignment can only be performed when an address is ready and these may become available out-of-(age)-order. Age must therefore be identified explicitly using a tag (e.g., ROB index plus wrap-around bit) rather than implicitly using position in a queue. This in and of itself is not bad, but in-order retirement and squashing due to branch and load mis-speculations de-allocate entries in (age) order, leaving holes in the bank queue and complicating future allocations. There is also the matter of low queue utilization due to bank imbalance. Several researchers have examined the scalability of centralized and distributed SQs; for an up to date summary consult Sethumadhavan et. al. [14].

Thread partitioning considerations. There are three basic queue partitioning policies [13]. One uses a single set of head/tail pointer and interleaves entries from different threads in the queue. This scheme is simple to implement and has perfect utilization, but induces cross-thread stalls. In static partitioning, a *T*-thread *N*-entry queue is treated at *T* independent *T/N*-entry queues; the T head/tail pointer pairs need not be cross-checked for collisions, each pair wrapping around locally in its own partition. In dynamic partitioning, the individual head/tail pointer pairs all circulate along the full length of the queue, and are cross-checked to ensure that the head of one does not collide with the tail of the next. Queue portions of individual threads can grow as large as N, but portions remain contiguous. Static partitioning is simple, ensures against thread resource starvation, and guarantees certain baseline levels of utilization and performance. However, it often results in low average utilization. Dynamic partitioning provides more flexible resource allocation and usually yields better utilization and (under the right conditions performance), although the contiguity requirement can leave large unusable holes in the queue and starvation is a problem. For these reasons, real multithreaded machines (like the Intel Pentium4 [10]) typically use static partitioning. Static partitioning can effectively reduce SQ and create capacity pressure on this timing critical structure. Increasing SQ size may relieve this pressure and increase pipeline utilization, but may also negatively impact load latency reducing both single-thread and multi-thread throughput.

CACTI latency estimates. To put our latency argument on more concrete footing, we ran simple CACTI 3.2 [15] simulations to estimate the access times of the two major components in a load store unit. It is important to note that these simulations are not accurate in an absolute sense (and in any case do not account for custom design) but should be accurate relative to one another. CACTI shows that in 90 nm technlogy, a 32KB, 4-way interleaved, 2-way set-associative data cache with 32B blocks, one read port and one read/write port has an access latency of 0.71 ns. At the same time, a 64-entry store queue with 2 associative match ports and a single write port has an access latency of 0.92 ns. The latency split is 0.75 ns for the address CAM, 0.10 ns for the data RAM and 0.07 ns for output driver; in contrast with the load queue latency calculations [4], store queue data RAM latency cannot be ignored (we did ignore the 0.10 ns delay of the sense-amplifiers which would probably not be used in a real design). The general point is clear: a multi-ported store queue can be slower than a reasonably sized data cache. In fact, CACTI indicates that for an SQ to have latency comparable to a 32KB cache, it must have 16 entries and only a single match port. Even then, its latency is 0.75 ns. For strictly lower latency—according to CACTI at least—8 entries are the limit.

Performance sensitivity to SQ parameters. With a better idea of the practical limitations of SQ size and access bandwidth, Figure 1 shows performance sensitivity to changes in three SQ parameters. The modeled machine is a deeply pipelined, 8-way dynamically scheduled superscalar processor with a 512-entry re-order buffer, aggressive branch predictor, 32KB primary caches, and a 2MB L2. A fuller description is contained at the beginning of Section

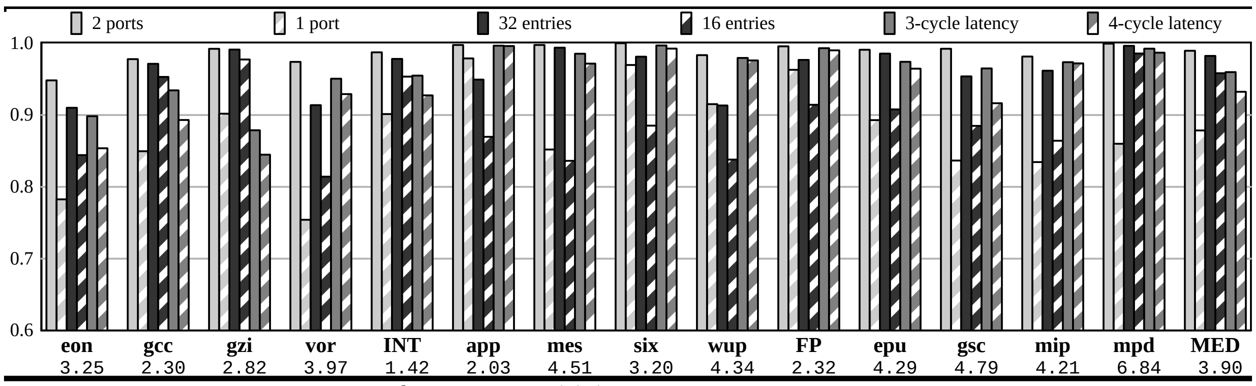


FIGURE 1. SQ parameter performance sensitivity.

4. Measurements are shown for four "forwarding intensive" benchmarks from each of the SPECint (INT), SPECfp (FP), and MediaBench (MED) [9] suites as well as the per-suite averages. Section 4 also contains a fuller description of the benchmarks and our simulation methodology. The chart shows IPC normalized to that of a baseline machine with a 512-entry (i.e., effectively unbounded) 3-port SQ with a 2-cycle access latency. The IPC of this configuration is printed below each benchmark. In each bar group, each pair of shaded bars (solid and striped) measures sensitivity to a different parameter: the left pair to the number of match ports (2 and 1), the middle pair to capacity (32 and 16 entries), and the right pair to access latency (3 and 4 cycles).

Reducing the number of SQ match (i.e., load) ports from 3 to 2 (effectively a reduction in load execution bandwidth) has a noticeable but not overwhelming effect on most benchmarks. However, a reduction to 1 port yields average slowdowns of 10%, 4%, and 13% in the INT, FP, and MEDIA suites respectively, with performance degradations of 20% and 30% not uncommon. Programs are generally less sensitive to reductions in SQ capacity, with slowdowns of 5%, 9%, and 5% respectively for a 16-entry SQ. Slowdowns of 10–15% are observed, however. Sensitivity to SQ read latency (i.e., load latency) is somewhere between the two—less than bandwidth but greater than capacity—with average slowdowns of 8%, 10%, and 7% across the three suites, respectively, for a 4-cycle SQ. It is not surprising that a wide superscalar with decent branch prediction would be more sensitive to load bandwidth than load latency.

Forwarding centric workload characterization. Our proposed design exploits the forwarding characteristics of programs to simplify the SQ, scaling both its bandwidth and capacity in an attempt to reduce latency. With an eye towards these optimizations, Table 1 shows a characterization of forwarding behavior for our three benchmark suites. For each program, we show the IPC and three metrics.

The ***load forwarding rate (LF%)*** is the fraction of dynamic loads that read their values from older in-flight stores and relates to the number of match ports a load-filtered queue can have. LF% is highly variable, ranging from 7% to 23% on the integer benchmarks, 0–43% on the floating point codes, and 0–44% on the media programs. These diagnostics suggest that if forwarding behavior is predictable—loads that forward almost always forward—we should be able to reduce the number of SQ ports by at least one half with no noticeable performance degradation. The ***store forwarding rate (SF%)*** is the fraction of dynamic stores that forward their values to younger loads. Combined with the average ***store queue occupancy (SQO)***, SF% can be used to estimate the SQ size reduction we should be able to achieve. For instance, for the integer benchmarks, an average occupancy of 16 stores and a forwarding rate of 30%

INT	**IPC**	**LF%**	**SF%**	**SQO**	**FP**	**IPC**	**LF%**	**SF%**	**SQO**	**MEDIA**	**IPC**	**LF%**	**SF%**	**SQO**
bzip2	3.44	11.3	32.4	26.8	ammp	1.17	15.0	37.3	26.6	adpcm.dec	2.93	0.0	0.0	1.1
crafty	3.61	6.9	34.9	9.5	**applu**	**2.03**	**18.9**	**22.4**	**47.3**	adpcm.enc	2.14	0.0	0.0	1.2
eon	**3.08**	**22.5**	**28.7**	**22.4**	apsi	1.54	6.8	7.6	53.0	epic.epic	3.89	0.5	5.0	14.9
gap	1.39	9.5	23.5	26.4	art	0.50	3.8	8.5	42.5	**epic.unepic**	**4.25**	**30.2**	**37.8**	**19.8**
gcc	**2.25**	**9.6**	**19.9**	**15.4**	equake	1.88	4.4	34.3	25.7	g721.dec	3.97	8.1	29.0	2.5
gzip	**2.80**	**18.5**	**50.4**	**8.7**	facerec	2.01	3.1	5.1	48.0	g721.enc	3.61	8.8	31.8	2.2
mcf	0.24	2.5	12.4	10.7	galgel	4.83	1.8	5.7	29.5	**ghostscript**	**4.75**	**21.6**	**33.4**	**12.7**
parser	1.82	13.9	39.9	13.7	lucas	3.11	0.0	0.0	26.9	gsm.dec	5.46	2.6	4.1	7.1
perlbmk	2.44	13.1	23.3	6.5	**mesa**	**4.50**	**24.7**	**42.8**	**36.6**	gsm.enc	6.09	3.9	25.2	6.6
twolf	2.05	9.9	34.3	12.5	mgrid	3.02	6.9	32.9	24.8	jpeg.dec	3.12	2.1	4.0	11.7
vortex	**3.86**	**23.1**	**33.4**	**22.8**	**sixtrack**	**3.20**	**42.8**	**79.2**	**44.7**	jpeg.enc	4.27	4.0	8.4	11.7
vpr	2.89	7.8	27.0	10.4	swim	1.87	4.3	7.2	38.5	**ms.mipmap**	**4.13**	**43.7**	**82.3**	**25.7**
mean(h,a,a,a)	1.40	12.4	30.0	15.5	**wupwise**	**4.27**	**21.3**	**44.7**	**36.4**	ms.osdemo	4.44	14.8	43.1	17.6
					mean(h,a,a,a)	1.80	11.8	25.2	37.0	ms.texgen	4.19	38.2	65.8	24.5
										mpeg2.dec	**6.84**	**20.3**	**66.7**	**8.8**
										mpeg2.enc	5.38	6.0	53.3	6.2
										pegwit.dec	2.95	1.5	6.8	4.1
										pegwit.enc	2.93	1.5	6.7	4.1
										mean(h,a,a,a)	3.86	11.6	28.0	10.1

TABLE 1. Store queue/forwarding behavior workload characterization.

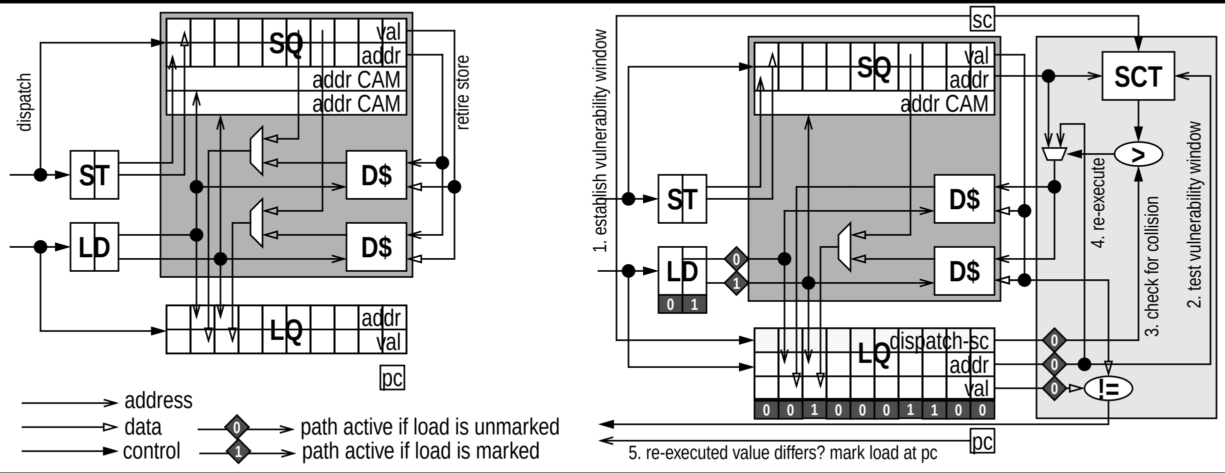
would suggest that we may be able to get by with a 5-entry SQ. Of course, these are just high level arguments. LF%, SF% and SQO are non-uniform within a given benchmark much less across benchmarks, and SF% and LF% are lower bounds on the fractions of stores and loads our predictor will steer to the SQ. However, with average store forwarding rates of 25–30% and accounting for predictor conservatism, we should be able to reduce SQ capacity by half.

3. New Load-Store Unit Design

The left side of Figure 2 shows a schematic of a conventional load-store unit with a two-way interleaved data cache, a store queue (SQ), and load queue (LQ). ST and LD are the store and load scheduling queues (reservation stations). This processor can execute one store and two loads per cycle; correspondingly the SQ has two address CAMs. For (relative) simplicity, we do not show the other functional units, register file, bypasses, or reorder buffer. The shaded background box encloses the “load execution critical path”.

We propose a new high-bandwidth, low-latency load-store unit design. Our design has a few more components than a traditional one. However, these components are either smaller and with fewer ports than their corresponding conventional components and more amenable to both address interleaving and multithreading, or off the load scheduling and execution critical paths. Our final design includes: (i) a conventional data cache, (ii) a non-associative age-indexed retirement SQ (RSQ), (iii) a small associative age-indexed forwarding SQ (FSQ), (iv) an address-indexed forwarding store cache (FSC), and (v) load re-execution pipeline stages and re-execution filtering structures.

Value forwarding is performed by the FSC and FSQ. The address-interleaved FSC provides high forwarding bandwidth and handles most forwarding cases. The associative FSQ handles that small subset of forwarding cases for which FSC-based communication fails. The RSQ provides in order retirement for stores. All dynamic stores access the RSQ and FSC; a subset of the dynamic stores also access the FSC. Dynamic loads access the data cache and



dispatch
ST
LD
SQ
val
addr
addr CAM
addr CAM
D$
D$
retire store
LQ
addr
val
pc
address
data
control
path active if load is unmarked
path active if load is marked
1. establish vulnerability window
sc
SCT
2. test vulnerability window
3. check for collision
4. re-execute
dispatch-sc
5. re-executed value differs? mark load at pc

FIGURE 2. A load-store unit using a two-way interleaved data cache with two load ports and one store port. LEFT: conventional design in which SQ has two read ports. **RIGHT:** modified design in which the SQ has a single load port. The SQ access stream is filtered by a predictor which itself is trained by a load re-execution mechanism (lightly shaded box).

either FSC or FSQ. Loads and stores are steered to the various structures by a simple predictor which is trained by the re-execution process.

This section is divided into three logical parts. First, we incrementally construct our mechanism (in three steps) without discussing load queue particulars or multithreading. Next, we show our design can easily be combined with value-based memory ordering (VBMO) [4]. Finally, we describe how our mechanism and VBMO can be extended to efficiently support multithreading.

3.1. Reducing Store Queue Search Bandwidth

The first aspect of our design is the reduction the demand for—and then supply of—SQ search bandwidth. The required modifications are shown on the right side of Figure 2. Notice, although we have added some machinery overall, there is less machinery in the shaded box that represents the load execution critical path.

The basic idea is simple. In a given program, only a fraction (1–50%) of all loads will read values from older in-flight stores. If we filter out loads that will not need to access the SQ, a faster, single-ported SQ should be able to handle the remaining accesses. There are several ways of implementing this filter, and two of them have already been examined. In one [14], a load does not access the SQ if its address “misses” in a Bloom filter that conservatively encodes the current address contents of the SQ. In the other [12], a load is filtered if any store from its store set [6] is currently in the SQ.

Load SQ-access prediction. We use a third scheme which does not require testing any aspect of the load against the current contents of the SQ. Since store-forwarding is a property of program structure—if one static instance of a load reads a value from an older in-flight store it is likely that future instances of the same static load will also require store queue forwarding—we simply learn from past behavior which static loads should access the SQ. We represent this information using a single bit per load. A load whose bit is set is said to be ***“marked”***; marked loads access the SQ, unmarked loads do not. Mark bits can be attached to instruction cache lines or reside in a separate structure. Since most dynamic loads do not require forwarding, a load is unmarked default.

Misprediction detection via load re-execution. Detection wrong SQ non-accesses—and learning mark bits—is done by re-executing filtered loads prior to retirement. A load whose re-executed value differs from the one stored in its LQ entry elicits a squash similar to one triggered by a mispredicted branch. The load is also marked.

We implement load re-execution in the same way it is implemented in [4], by adding three pre-commit pipeline stages, *filter* and *re-execute.* Re-execute may take multiple stages depending on data cache access latency. Value comparison (simple exclusive-or) is folded into the last re-execution stage. In these stages, unmarked loads are re-executed while all other instructions, including marked loads (i.e., loads that accessed the SQ) do nothing. Load re-execution shares data cache ports used for store retirement; these are converted to read/write ports. For more details on re-execution, consult [4].

Reducing re-executions. The addition of two pipeline stages does not result in significant slowdown (< 0.5% for most programs) and re-execution is "dependence free" [2] such that even dependent loads can be re-executed in parallel. The real cost of re-execution is increased contention for the store retirement ports which can slow commit and create pipeline backpressure. Fortunately, only a tiny subset of all loads require re-execution. Marked loads are assumed to be handled correctly by the SQ and are not re-executed, but we can also forgo re-execution for the majority of unmarked loads.

A given dynamic load is "vulnerable" to only a small contiguous window of older stores. The oldest store in this window is the oldest store in the re-order buffer at the time the load was dispatched. The youngest store in this window is the youngest store that is also older than the load, i.e., the load itself terminates the window. A load need only be re-executed if any store within this ***window of vulnerability*** wrote to the same address.

To implement this re-execution filter, we add a global counter called the ***committed store counter (SC).*** The counter increments whenever a store retires and never decrements, simply wrapping around. When a load is dispatched, the current value of the CS is copied into a new field in the LQ called ***dispatch-SC***. Dispatch-SC effectively defines the window of vulnerability for each load. When the load retires, we check for address collisions with all retired stores whose own CS is greater than the load's dispatch-SC.

The address collision check itself is performed using an address-indexed table of SC values called the ***store counter table (SCT)***. A retiring store increments the global SC counter and writes this value into the SCT at the position indexed by its address. Thus at any point, each slot in the SCT contains the SC of the last store to write to any address that hashes into the slot. When an unmarked load retires, it uses its address to index the SCT. An SC value greater than its own dispatch-SC signals a probable address collision and the load is re-executed. A collision is only probable because the value may belong to a store to a different address that hashes to the same SCT slot. A lesser SC value indicates a definite non-collision and re-execution is skipped.

SC wrap-around handling builds on the observation that the maximum size of the window of vulnerability is the size of the SQ. When the global SC wraps around the SCT is flash-cleared and re-execution proceeds unfiltered (i.e., without SCT consultation) until the first load whose dispatch-SC is greater than the size of the SQ. The width of the SC counter should be set so that wrap-around is infrequent and the fraction of time the SCT is unusable is small. For instance, a 16-bit SC will wrap around every 64K stores. With a 64-entry SQ, the SCT will be unusable for 64 of those 64K stores, or about 0.1% of the time. This is quite acceptable.

In the figure, the steps of establishing the vulnerability window, testing for collisions, potentially re-executing, and potentially marking the load are numbered and labeled.

SCT as Bloom filter. Our SCT acts as a Bloom filter [3], an approximation of a lookup function that trades size and speed for false positives. Here the approximated function is window-of-vulnerability address-collision. Note, this is different use of a Bloom filtering than [14]. There, the Bloom filter guards load access to the SQ (i.e., it is on the load execution critical path), is managed speculatively, and contains information about stores both older and younger than a given load making it more vulnerable to false positives. Our filter guards load re-execution (i.e., it is not on the load execution critical path), is managed non-speculatively, and only contains information about older stores.

3.2. Reducing Store Queue Capacity

Our method for reducing SQ search bandwidth built on the observation that in-flight store-load communication is a structural property of the combination of program and dynamic window size and, hence, predictable. We can use the same observation to reduce the demand for SQ capacity—and to subsequently reduce that capacity—by using past (non-)forwarding behavior to filter stores from the SQ. The modifications are shown on the left side of Figure 3. Again, more machinery has been added, but the load execution box has gotten smaller and less complex.

Store SQ-entry prediction. The mechanics of SQ store filtering are simple. A single bit per static store predicts SQ entry. Like loads, all stores are “unmarked” by default. A load re-execution that results in a squash marks both the load and the store that should have forwarded the value. To locate this store, the SCT is augmented with a matching (i.e., similarly organized and indexed) array of store PCs. Each entry in this array, the ***store PC table (SPCT)***, contains the PC of the last retired store to write to the corresponding address.

Retirement and forwarding SQs. A conventional SQ performs two functions: (i) in-order store retirement for stores, and (ii) value forwarding to younger in-flight loads. While our optimization filters stores from the SQ that do not require the second function (forwarding), all stores require the first function (in-order retirement). To sidestep the situation, we implement the two functionalities of a conventional SQ using two structures. The ***retirement store queue (RSQ)*** contains all stores but is only used for in-order retirement and therefore isn’t associatively searched. In fact, it is removed from the critical timing path of the load-store unit. The ***forwarding store queue (FSQ)*** is used for forwarding and does include associative search machinery, but contains only a fraction of the dynamic stores and is therefore small.

RSQ and FSQ both resemble a conventional SQ. They both have address, data, and ready-bit fields. Stores are allocated entries in the RSQ and FSQ in program order at the dispatch stage. All stores are allocated RSQ entries while only marked stores are allocated FSQ entries. Dispatch stalls when either RSQ or FSQ is full. At retirement, stores are dequeued from the RSQ and written to the data cache. Entries are also dequeued from the FSQ at retirement, but the latter does not have a data cache interface.

Re-execution modifications. Filtering stores from the FSQ introduces a mis-speculation mode for marked loads. Previously, a marked load could not read a wrong value and did not require re-execution. Now, however, a marked load can read a wrong value if the forwarding store was erroneously filtered from the FSQ.

As in the case of unmarked loads, not all marked loads need to be re-executed. Using the SCT would work but

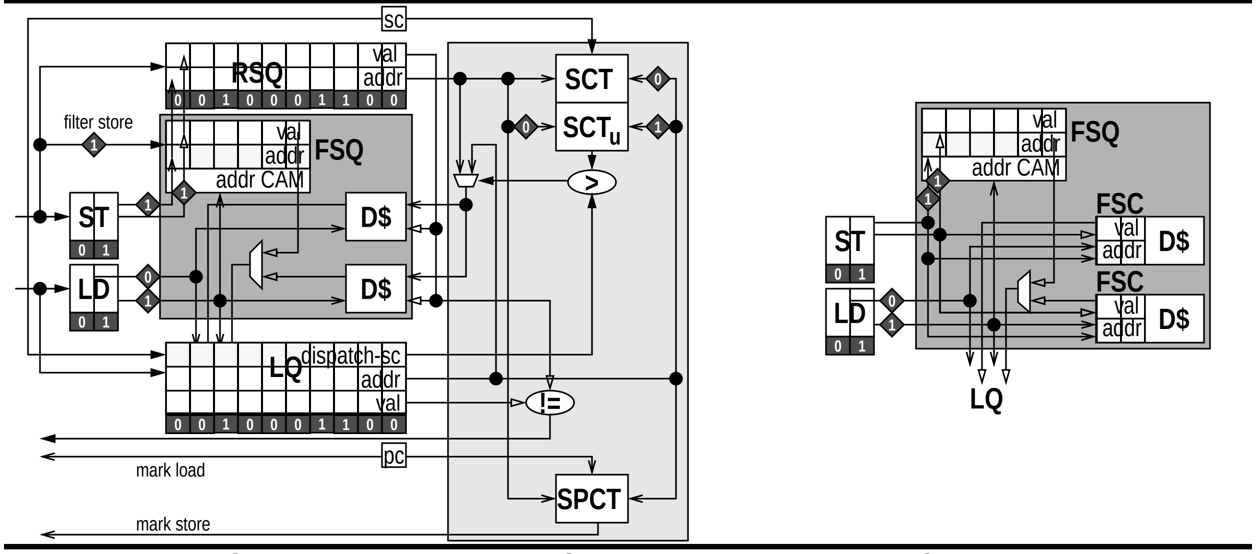


FIGURE 3. A modified load-store unit. LEFT: modified design with single-ported load filtered SQ conventional design in which store queue (SQ) has two load ports. **RIGHT:** modified design in which store queue has a single load port.

would result in many unnecessary re-executions. The SCT contains the SC values of all stores, whereas marked loads are only vulnerable to unmarked stores. We exploit this fact by filtering the re-execution of marked loads using a second SCT (SCT$_u$) which only contains the SC values of unmarked stores.

3.3. Restoring Forwarding Bandwidth

Our FSQ capacity and bandwidth optimizations rely on most programs having relatively moderate forwarding requirements. In a window of 512 instructions, fewer than 25% of loads and 50% of stores participate in forwarding. However, as we saw in Section 2.1, a few programs (e.g., *crafty, mesa,* and *sixtrack*) require much more forwarding entries and bandwidth. A small, single-ported FSQ will degrade performance for these programs. The final aspect of our design involves restoring some of the forwarding bandwidth that was trimmed by our first two optimizations, but doing so in a more scalable way, without forfeiting the FSQ simplifications. The right side of Figure 3 show the final set of modifications. Here we show only the load execution box. The remaining structures are essentially unmodified from the left side of the figure.

The observation that drives our final modification is that there are two kinds of forwarding scenarios: simple ones and difficult ones. In a simple forwarding scenario, the store naturally completes before the load and at any given time there is only one store to the address in the window. In these scenarios—stack save/restore pairs are an obvious example, but there are others—any forwarding mechanism, however fragile, is likely to work correctly. A difficult forwarding scenario involves out-of-order completion of the participating store and load, or multiple in-flight stores to the same address. A more robust forwarding structure like an FSQ is needed to perform difficult forwarding correctly.

We exploit this observation by adding to each data cache bank a small, simple forwarding structure called the ***forwarding store cache (FSC)***. The FSC off-loads the simple forwarding scenarios from the FSQ, leaving the latter to handle only the difficult ones. Stores and loads that forward correctly via the FSC are viewed as correct by the filtered re-execution process and never marked for FSQ entry. The marking process does not distinguish between correct forwarding via the FSC and no forwarding at all.

Forwarding store cache. While both the FSQ and FSC are used for forwarding purposes only, there are major differences between them. The FSC can be thought of as the address-indexed counterpart of the age-indexed FSQ. The FSC is implemented as a cache with word-sized blocks and FIFO replacement. For high bandwidth access, the FSC is address-interleaved in a fashion that mirrors the interleaving of the data cache. Like a cache, each FSC bank can contain several address indexed sets. Stores are allocated FSC entries at the execution stage; at the address execution stage if store address and data are written to the SQ separately. All stores are written to the FSC.

Unmarked loads, which previously only accessed the corresponding cache bank, now access the attached FSC bank as well. A load reads the value of the youngest matching store that is also older than itself that currently resides within the set. FSC entries represent age explicitly. Age tags are implemented as ROB indices. Relative age and proximity can be determined by subtracting the store entry ROB indices from the load ROB index (with appropriate wraparound consideration) and selecting the smallest positive number. If store execution is split into address and data actions, and data is not present for the matching store, the load sleeps and is re-awakened when the store data arrives, when it becomes the oldest instruction in the window, or when the set's FIFO tail pointer passes the store.

Difficult forwarding scenarios. The difficult forwarding scenarios that the FSC cannot handle fall into three categories: (i) store-load execution mis-ordering, (ii) set overflow, and (iii) stale entries from squashed paths.

The execution mis-ordering scenario is analogous to load mis-speculation in a conventional SQ, except that in an FSC there is no reserved slot to indicate that the store exists. Set overflow and the forwarding of squashed data are unique to the FSC, and both occur because FSC entries are not allocated in age order. Overflow occurs because retiring entries from the set requires continuous tracking of order, which is expensive. Without explicit freeing of entries, there is no notion of a set being "full". FSC entries are allocated in FIFO fashion using a pointer associated with each set. A new entry typically displaces stale (retired or squashed) entry but may displace an entry belonging to an active store as well. Just as it is difficult to retire entries from an FSC set in age order, it is also difficult to recover them in age order on a branch mis-prediction. The FSC ignores mis-predicted branches, the FIFO pointers are not reset. Overflow cannot happen in the FSQ which stalls when it is full.

Loads that mis-speculate as a result of any of these conditions are caught by the re-execution mechanism. Future instances of these loads (and the participating stores) are handled by the age-indexed FSQ, which is not vulnerable to these mis-speculation forms.

Re-Execution modifications. Since the FSC is unordered and there is no efficient way to track its contents, we cannot rely on any forwarding to be implemented correctly. FSC or not, an unmarked load is re-executed if a collision is detected in the SCT. However, the FSC does have one mis-speculation scenario which the SCT filter will not capture: forwarding a value from a squashed store. Since squashed stores are not retired, their existence is never noted in the SCT. To capture this scenario, we must re-execute any load that forwarded from the FSC, regardless of SCT "hit/miss" status. This change only represents one additional bit of logic and does not result in any additional re-executions. If a load forwarded a value from the FSC and hit in the SCT, re-execution would have taken place anyway. However, if the same load missed in the SCT, we know that the forwarded value came from a squashed store. We can squash and mark the load without re-execution.

3.4. Adding (To) Value-Based Memory Ordering

Earlier we noted that our technique can be implemented together with value-based memory ordering (VBMO), a scheme that attacks load-queue scalability [4]. VBMO uses a non-associative LQ that is the counterpart of our RSQ. Loads are re-executed prior to retirement and mismatches between the re-executed value and the LQ value elicit squashes. The re-execution stream is filtered using several heuristics, each of which captures a vulnerability to a particular form of mis-speculation. Loads vulnerable to intra-thread memory ordering violations are detected using LQ bits which are set whenever a load executes in the presence of older stores with unknown addresses. This heuristic was actually used earlier to filter loads from the LQ [12]. Loads vulnerable to inter-thread (i.e., consistency model) violations are detected by tracking coherence events (misses and/or invalidations) and re-executing all loads within the window of vulnerability—all loads up to the youngest load in the processor—of such an event. A heuristic that captures both intra- and inter-thread violations re-executes all loads that executed out of order with respect any older store or load; this heuristic results in a large number of replays, so combination of the other three is suggested instead.

Combining our technique with inter-thread (consistency model) VBMO. Our design accommodates an adapted (actually improved) version of VBMO's inter-thread re-execution heuristics. VBMO's scheme trigger load re-execution regardless of whether the load accesses the block manipulated by the coherence event. We suspect that this is done because tracking multiple block addresses and multiple vulnerability windows is complex, especially if speculative execution is accounted for. However, we already have a method for tracking and computing address collisions within individual vulnerability windows: the SCT. To flag inter-thread re-execution scenarios, we add a third SCT—SCT_{mp}—in which entries correspond to cache blocks rather than words. On a miss or invalidation, the SCT_{mp} entry corresponding to the block address is marked with a value that is equal to the current global SC plus the size of SQ. This high value effectively tags all in-flight loads for re-execution. However, since loads check the SCT_{mp} prior to re-execution, only those that read the corresponding block are actually re-executed.

Combining our technique with intra-thread (load speculation) VBMO. Our scheme is orthogonal to the address disambiguation policy used by the FSQ. Loads can wait for all older store addresses before accessing the FSQ (conservative), execute immediately and force older stores to check for ordering violations (opportunistic), or wait for only a subset of older stores, specifically those that produced past collisions (intelligent).

In VBMO, potential intra-thread violations are flagged for re-execution by the scheduler. We can use that same heuristic as is, but as in the case for inter-thread violations, we have the opportunity to improve on it. One of the stated limitations of VBMO is that it does not track the identities of stores that trigger squashes and thus can only be used to train a simple store-blind dependence predictor, rather than a more refined store-load pair predictor like store-sets [6]. Again, we already have the solution, this time in the form of the SPCT which we need to mark stores for entry into the FSQ. The SPCT can supply the store PC for the store-load pair predictor.

3.5. Adding Multithreading

Our load-store unit is easily extended to support multi-threading. The RSQ is statically partitioned and made somewhat larger to avoid causing under-utilization in the rest of the machine. A reasonable size increase is not a problem because the RSQ is not associatively searched and is not on the execution critical path. VBMO's non-associative LQ can be enlarged and statically partitioned for similar reasons.

The FSQ also tolerates a moderate size increase in the name of pipeline utilization, but for a different reason than the RSQ. The FSQ is associatively searched during load execution. However, it is small and single-ported such that prior to this increase, it should not have been the slowest component in the load execution path.

The FSC supports multithreading differently. Recall, the FSC is actually a cache with FIFO replacement, not a true queue. It is therefore treated like a cache, simply by adding a thread identification tag to each entry. Each FSC set has a single global FIFO pointer which is shared by all threads.

In a multithreaded processor, the SCT, SCT_u, and SPCT can be replicated along with the SC counter. However, that is not necessary. All threads can share a single SCT/SCT_u/SPCT and a single global SC counter. The counter increments on a retirement of a store from any thread; the SCT/SCT_u/SPCT are similarly updated. The shared counter and in-order retirement ensure that values in the SCT/SCT_u increase monotonically and that no necessary re-executions are missed. There would be increased re-execution due to cross-thread aliasing in the SCT/SCT_u, and some incorrect store marking due to aliasing in the SPCT, but these effects should be small. The SCT_{mp} is naturally shared.

4. Experimental Evaluation

We evaluate our new load-store unit design using timing simulation on the SPEC2000 integer and floating point benchmarks. Our performance simulator executes the Alpha AXP user-level instruction set using the instruction definition and system call modules supplied with SimpleScalar 3.0. We model single- and multi- threaded superscalar processors with MIPS-style register renaming, out-of-order execution, aggressive branch prediction and a two (or three) level on-chip memory system complete with buses and finite outstanding miss handling resources.

Processor configuration. Our base processor configuration is an 8-way issue superscalar 512 entry reorder buffer, 240 reservation stations, and 448 physical registers. The base pipeline has 15 stages pipeline (3 fetch, 2 decode, 2 rename, 2 schedule, 3 register read, 1 execute, 1 writeback, 1 commit). To our modified configurations, we add three re-execution stages: one for SCT access and two for data cache access. Our processor has five integer ALUs, 2 store ports, 3 load ports, 2 FP adders, and one integer/FP multiply/divide unit. We use intelligent load speculation managed by a 64-entry store-set table. There is one retirement store port. The fetch unit includes an 8K entry hybrid gShare/bimodal predictor, a 2K entry, 2-way set-associative target buffer, and a 32-entry RAS. The instruction cache is four-way interleaved and fetch can proceed past one taken branch per-cycle. The primary instruction and data caches are 32KB, 2-way set-associative, 2-cycle access, with 32B blocks. The L2 is 2MB, 8-way set-associative, 15 cycle access, with 128B blocks. The instruction and data TLBs are 64-entry, 4-way set-associative. Memory latency is 150 cycles. The L2 and memory buses are both 16B wide, the latter is clocked at one quarter processor frequency.

Our simulator also supports simultaneous multithreading (SMT) [17] configurations which we use simulate a multi-programmed workload. Our environment does not support shared memory multiprocessing. In Section 4.5, we model an SMT with two threads and 512 physical registers (the additional 64 are needed to hold the architectural mappings of the second context). We use the ICOUNT [16] policy to manage fetch and round-robin policy for retirement. The back-end queues—ROB, LQ, and SQ or RSQ/FSQ—have per-thread head and tail pointers to minimize cross-thread stalls and backpressure. Both static and dynamic partitioning is modeled [13].

Benchmarks. Our benchmarks are taken from SPEC2000 and MediaBench [9] suites. We run the benchmarks to completion, on the training input sets for SPEC and the supplied inputs for MediaBench, using 5% periodic sampling with 5% cache and branch predictor warm-up. Each sample contains 10M instructions for SPEC and 2M instructions for MediaBench. For multi-programmed simulations, we use a variant of this sampling methodology. Each sample contains a fixed (and equal) number of instructions per program. If one program finishes its sample first, the remaining active programs re-partition resources among themselves. This setup has the advantages that higher performance cannot be achieved simply by biasing high-IPC threads and that running the programs in a sample sequentially rather than concurrently is a legal option (as it would be in real processors). Empirically, concurrent execution is faster.

Except for a brief characterization in Section 4.1, our results are shown in graph form. Since there is not sufficient space to show results for all 43 programs, we show only five bars for each benchmark suite: 4 programs with extreme forwarding behavior (these are shown in bold in Table 1) and the average over the entire suite. Our techniques have little impact—positive or negative—on programs with sparse forwarding; there is no forwarding bottleneck to remove and since there is little forwarding there is also virtually no re-execution.

4.1. Reducing SQ Accesses and Ports

Figure 4 shows performance relative to that of a machine with a 512-entry, 3-match ported SQ of four configurations. The first two configurations use a 512-entry SQ with 2 and 1 match ports respectively and no load filtering. The baseline machine effectively has 3 load ports. Without SQ access filtering, these two configurations effectively have 2 and 1 ports. Given realistic branch prediction and the fact that loads account for 20-25% of the dynamic instruction stream, most programs cannot exploit a third load port; their performance with two ports is often identical to—and always within 5% of—their performance with three. However, with only a single port, performance degrades sharply. Average slowdowns are 10% for integer, 4% for floating-point, and 13% for media, slowdowns of 20% are common and *crafty* experiences a 32% slowdown.

The right bars show two load filtering configurations. Each uses a 512-entry SCT and a 64-entry SQ with a single match port. The first configuration has two load ports—in a given cycle, it can execute two loads, but only one can be marked; the second has three load ports. On top of each load filtering experiment bar, two numbers are printed. The number that precedes the letters *la* is the ***load access rate***, the fraction of loads that were steered to the SQ by the predictor. The number that precedes *lr* is the ***load replay rate***, the fraction of loads that had to be replayed. Thus for the 2-port configuration, *eon*'s SQ load access rate is 49.0% and the replay rate is 1.7%. We do not print the ***load squash rate***, the percentage of loads that resulted in squashes. The squash rate is less than 0.01% for all programs except for vortex, for which it is 0.06%. The squash rate is low because each static load is effectively squashed only once. After that it is marked and conservatively accesses the SQ.

Load filtering can effectively compensate for the loss of one SQ ports. The filtered 2 load slot/1 SQ port configuration performs as well as the unfiltered 2 load slot/2 SQ port configuration for all benchmarks except for *eon*. It is more difficult to compensate for the loss of 2 SQ ports. The filtered 3 load slot/1 SQ port configuration is an average

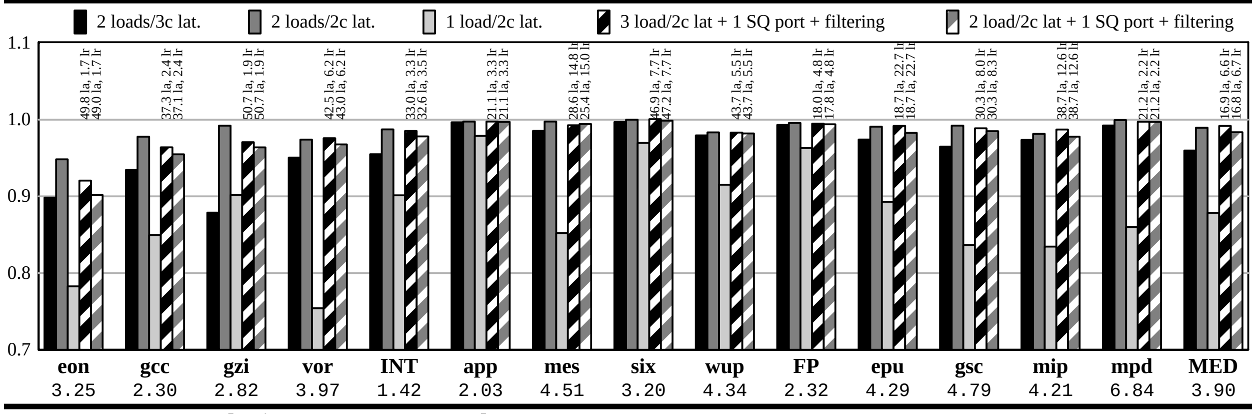


FIGURE 4. Reducing SQ accesses and ports.

of 2% slower than the 3 load slot/3 SQ port configuration with a slowdown of 8% for *eon*. Note, slowdown is proportional to the load access rate. In the case of *eon*, with only 50% of loads filtered from the SQ, we can understand why an access bandwidth reduction of 67% is not fully compensated for.

Notice, the average load access rates for the INT, FP, and MEDIA suites are 33%, 18%, and 17%. These are somewhat higher than the respective load forwarding rates (LF%) from Table 1: 12%, 12%, and 12%. The access rate is higher than the forwarding rate because it is driven by a predictor which conservatively steers any load that required forwarding in the past to the SQ. If forwarding behavior is consistent, the number of spurious accesses is small. However, if forwarding behavior is path dependent—many static loads require forwarding along some paths and not on others—our simplistic predictor can result in many superfluous accesses. Intuitively, path dependent forwarding behavior is much more prevalent in the INT benchmarks. We are experimenting with a simple path-based extension to our predictor that associates 2^h bits with every load where each bit marks the load for a different *h*-bit branch history. A path-based predictor will lower the access rate, but will also increase the replay and squash rates.

Re-executions rates are also reasonable: 3%, 5%, and 7% for INT, FP, and MEDIA with highs of 6% (*vortex*), 17% (*mesa*), and 45% (*epic.unepic*). These are comparable to the 0.02 re-executions per retired instruction (our rates are per retired load) reported by Cain and Lipasti for their uni-processor runs [4]. Notice, re-execution and access rates are inversely related to one another, since it is a fraction of the unmarked loads that are re-executed.

4.2. Reducing SQ Occupancy and Capacity

Figure 5 shows performance of six configurations relative to a baseline with 2 load ports, 2 cycle load latency, a 64-entry, single-ported SQ with load filtering. The three left-most configurations (solid) use SQs with 64, 32, and 16 entries. The three to the right (striped) add store filtering to each of the three SQ (now FSQ) sizes. Each uses a 64-entry RSQ, and 512-entry SCT, SCT_u and SPCT; the total size of the last three structures is 4KB.

Figure 1 previewed program sensitivity to SQ size. A 32-entry SQ does not significantly impact performance overall but does have an impact on selected programs, e.g., *vortex* and *wupwise*. A 16-entry SQ—sized to roughly match the access latency of the data cache—induces slowdowns as large as 17% (*mesa*) but average slowdowns are only 3% for INT and MEDIA and 8% for FP. An 8-entry SQ degrades performance sharply for several programs (by more than 30% on *vortex* and *mipmap*) with a 10% performance degradation across the board.

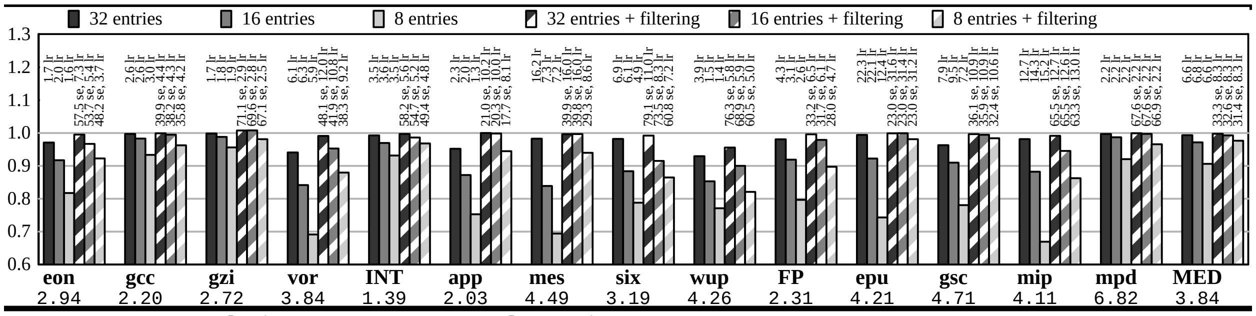


FIGURE 5. Reducing SQ occupancy and capacity.

Store entry filtering can alleviate much of this capacity pressure, with a filtered FSQ of a given size often outperforming an unfiltered RSQ with twice, and sometimes four times, the number of entries. Store entry rates, shown on top of the bars as numbers that precede the letters *se*, help explain. On average, store filtering removes 40%, 70%, and 70% of the stores from the FSQ for the INT, FP, and MEDIA benchmarks, respectively (the store entry rate shown is the fraction of stores not filtered). The store entry rates are less comparable to the store forwarding rates (SF%) from Table 1 in Section 2, then load accesses rates are to load forwarding rates. The reason for this is that SQ size affects the forwarding rate; a larger SQ results in more forwarding (at one limit, a zero-sized SQ results in none). Nevertheless, anecdotally, store entry rates are tighter to the bounds established by the store forwarding rates than load access rates are to their bounds. The reason is that in most path-dependent memory communication, the store is guarded with respect to the load. A history based predictor is therefore not likely to benefit SQ capacity as much as SQ bandwidth.

As noted earlier, store filtering can increase the load re-execution rate, shown on top of the bars as number that precedes the letters *lr*, as some marked loads will now need to be re-executed as well. This effect is generally mild. The re-execution rate, generally under 7% to begin with, rarely grows by 5% absolute. For the remainder of our evaluation we will use the 16-entry FSQ. It performs nearly as well as a 64-entry SQ for most programs, and its access latency is sufficiently close to that of the data cache.

4.3. Restoring Forwarding Bandwidth using an FSC

As shown in Figures 4 and 5, load and store filtering can, respectively, compensate for reduced SQ load bandwidth and capacity in many cases. As we showed in Table 1, some benchmarks have extremely high forwarding requirements involving many dynamic stores and loads and the FSQ simplifications we propose will invariably slow them down, even accounting for the gain in load latency. In Figure 6, we show the load and store filtered, 16-entry FSQ configuration from the last section and three configurations that add an FSC to off-load both load accesses and store entries from the FSQ. The baseline is the same as it was in the previous section: a 2-cycle, single-ported, 64-entry SQ with load filtering. This is not the ideal baseline of our first experiment, so performance higher than this baseline is possible.

Recall, the FSC is address interleaved and each bank parallels a data cache bank. The three FSC configurations use 64 total entries and two associative ways per bank (512 B of storage), 128 total entries and 4 ways (1KB), and 256 entries and 8 ways (2KB). The 8-way configuration is probably unrealistic, because its estimated access time, 0.67 ns, does not leave enough room for age tag comparison (which is not modeled by CACTI). The estimated latency of the

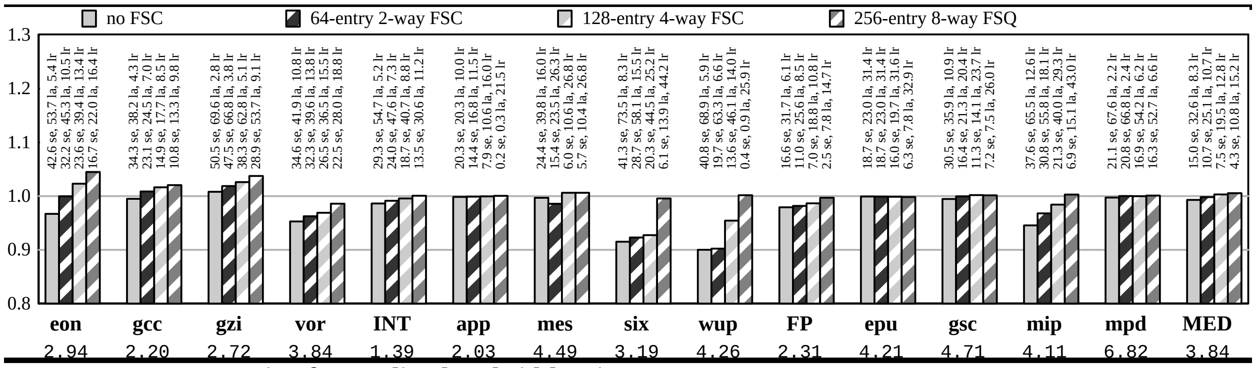


FIGURE 6. Restoring forwarding bandwidth using an FSC

4-way configuration is 0.59 ns, more comfortably below the 0.71 ns data cache access latency.

Even a 4-way set-associative FSC can significantly improve the performance of some benchmarks, e.g., *eon*, *crafty*, *wupwise*, and *mipmap*. The diagnostics that support this performance improvement are shown on top of each bar. An FSC dramatically reduces the FSQ store entry rate (*se*)—on *eon*, for instance, from 42% to 23% for a 4-way FSC—as well as the FSQ load access (*la*). Generally speaking, in benchmarks where adding the FSC improves performance up to previous baseline levels, the FSC is compensating for reduced FSQ capacity. If FSC performance exceeds baseline performance, the FSC is compensating for reduced FSQ bandwidth. The store entry and load access rates support this assertion.

The extremely low store entry and load access rates for the 8-way FSC, 2% and 7% respectively for the FP benchmarks, suggest that there are indeed many “easy” forwarding instances that do not require a synchronized, globally ordered structure for correct handling. As it turns out, it may be possible to achieve this degree of siphoning even without an 8-way FSC. One limitation of our current design is that all stores enter the FSC. One possible optimization would be to filter non-forwarding stores from the FSC as well in an attempt to reduce overflow and increase effective associativity. This would require a three-state prediction mechanism and is left for future investigation.

Another negative aspect of the FSC is that siphoning a load from the FSQ turns it from a load that probably will not need to be re-executed to a load that will be re-executed. This can be seen as an increase in the load re-execution rate (*lr*) as the FSC is first added and then enlarged. See *mipmap* for an extreme example of this behavior. With two store-retirement, load-re-execution ports, the added re-executions are not a problem. However, as we show in Section 4.5, with more limited re-execution bandwidth they can become one. Another area of future work is the examination of cheap techniques for guaranteeing the correctness of some loads that forward via the FSC so as to avoid re-execution for these loads.

Summary of cumulative effects. The current limitations of our technique aside, the combination of our three techniques is quite effective. Although not directly shown in any of our graphs (we change baselines due to the serial nature of the evaluation), the three techniques combine to achieve performance that on average is 2% lower than that achieved with an ideal, 2-cycle access, 2-ported, infinite capacity SQ, using only a 16-entry, single ported SQ. In the worst case, *vortex*, performance is only 7% below ideal. This same performance represents an average 3% improvement over a realistic 3-cycle access, 2-ported, 64-entry SQ with peak improvements of 12% (*gzip*). These ranges are

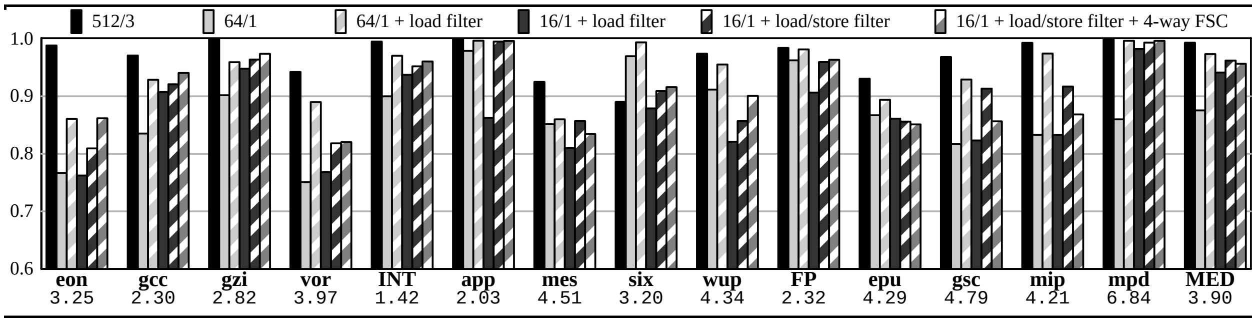


FIGURE 7. Constrained re-execution bandwidth.

magnified when we remember that these averages include many programs that are quite insensitive to SQ parameters.

4.4. Limited Store Retirement/Load Re-Execution Bandwidth

With two store-retirement/load-re-execution ports and highly filtered re-execution, re-execution bandwidth is not a bottleneck. In Figure 8, we repeat a selected set of experiments for a processor with only a single store-retirement/load-re-execution port. Performance is shown relative to a baseline with an “ideal” SQ (512-entry 3-ports) and two store-retirement ports. The first experiment uses the ideal SQ but only a single store-retirement port. This bar is included to isolate the effect of port contention on the baseline machine. Except for a few, high IPC, high store ratio FP benchmarks (e.g., *mesa* and *wupwise*) a single port is not a bottleneck if used for store retirement only.

The next two light bars show the effect of load access filtering. The solid bar uses a 64-entry 1-port SQ; the striped bar adds the load filter. Because load filtering only re-executes unmarked loads, a single store port is still not a serious bottleneck even though it is now shared between store retirement and re-execution. The following two dark bars measure the effect of store entry filtering on a load-filtered SQ. The solid bar uses a 16-entry SQ; the striped bar adds store entry filtering. Again, these bars have the same qualitative relationship to each other and the ideal baseline as the corresponding bars for a machine with two retirement/re-execution ports. The reason is

We start to see a qualitative difference in the last experiment, which adds an 8-way FSC to the store and load filtered SQ. As we explained in Section 4.4, the FSC siphons forwarding accesses and entries from the FSQ. This is good for FSQ scalability, but bad because the forwarding through the FSC requires re-execution but forwarding through the FSQ may not. On a machine with two store-retirement/load-re-execution ports, the contention from the added re-executions is not destructive. With only a single port, however, contention from the added re-executions is a real problem and may overwhelm the benefits of the FSC itself. This is seen in several benchmarks in all three suites: *e.g., vortex*, *mesa*, and *ghostscript*. In fact, in the MEDIA suite, added FSC re-executions produce an average slowdown (albeit only 0.5%) over all benchmarks. The lesson is clear. Re-execution latency is in general not a problem. However, when re-execution bandwidth is limited, techniques that increase re-executions should be applied with care.

4.5. Multi-Programming Experiments

Figure 8 shows the effect of our techniques in the setting of a multithreaded processor executing a multi-programmed workload. This is not meant to be an exhaustive study, so we simply we paired SPEC integer benchmarks together. Our multi-programmed simulation and sampling methodology was described earlier. The six experiments

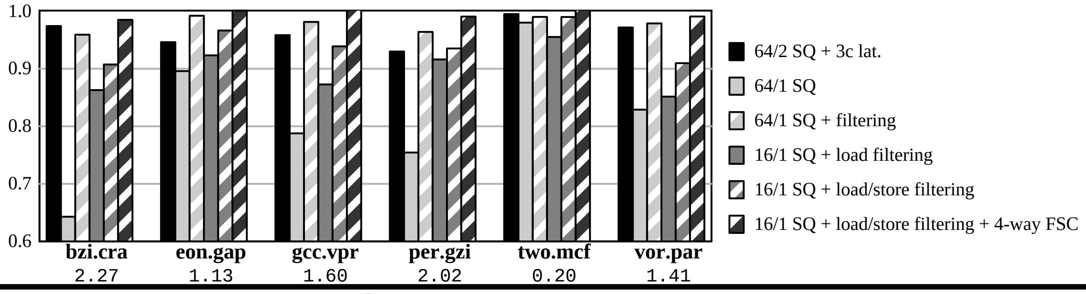


FIGURE 8. Multi-programmed performance.

shown all show performance relative to a baseline machine with statically partitioned queues, and a 64-entry, 2-cycle access SQ with two load ports. The aggregate performance of this configuration is printed under each benchmark pair.

The first of the six experiments, uses a 64-entry, dual-ported SQ with a 3-cycle latency. Not surprisingly, multi-threaded execution is less sensitive to load latency than single-threaded execution, but slowdowns of 8% are still observed. Multithreaded execution is sensitive to reduced load bandwidth as shown in the second experiment which uses a 64-entry SQ with a single read port, but not more so than single-threaded execution. As in single-threaded execution, load filtering—performed using a shared SCT as described in Section 3.5—is an effective solution.

Multi-threaded execution, however, is more sensitive to SQ capacity reductions, as static partitioning magnifies queue capacity effects. In single-threaded execution mode, moving from a 64-entry SQ to a 16-entry SQ resulted in an average 3% performance degradation on the INT benchmarks. Here, the same reduction results in a slowdown of 12%. The effect would be magnified if we were to run four threads concurrently rather than only two. The final two experiments add store filtering and a 4-way FSC to a 16-entry FSQ. Store filtering is somewhat effective at increasing effective FSQ capacity. However, the bigger boost comes from the FSC, which diverts many more stores away from the FSQ and restores performance all the way to unrealistic baseline levels. One reason the FSC is so effective in a multithreaded setting is that multithreaded execution is less speculative from the point of view of each thread and forwarding instances that surround unpredictable branches are less prone to mishap, marking, and future FSQ inclusion.

5. Related Work

Several researchers have observed the difficulties in scaling the load and store queues to both large sizes and a large number of ports. Sethumadhavan et. al. [14] studied the scalability of both age-indexed and address-indexed store queues and show a design that reduces SQ access bandwidth. Their SQ is guarded by a Bloom filter that conservatively encodes the addresses of in-flight stores. Only loads whose own addresses “hit” in this filter access the SQ. This scheme is generally effective, but suffers from several drawbacks. Specifically, the Bloom filter is managed speculatively and out-of-order meaning that its contents are difficult to maintain precisely and that it is vulnerable to false positives from loads that match younger (i.e., non forwarding) stores. It also needs to be accessed on the load execution critical path. The same authors also show how LQ search bandwidth and load queue size can both be reduced using similar Bloom filter guarding. Concurrently, Park etl. al. [12] proposed a design that achieves the same access bandwidth reductions, but using a store-load dependence predictor [11, 6, 19] rather than a Bloom filter. They also showed how SQ size can be scaled by chaining small SQ segments together. Each segment can be accessed quickly, but segments are accessed serially so that loads that communicate with distant stores execute more slowly than loads that communicate with nearby stores. Note, loads that don't communicate with any stores also execute quickly. These are spared from searching the segmented SQ by the store-load dependence predictor. Akkary et. al. [1] attacked SQ scalability in a similar way using a two-level SQ. A fast first-level queue holds the most recent stores while a smaller, second-level queue holds all in-flight stores. A fast filter eliminates most searches to the second-level queue.

Dependence-free checking—via in-order pre-retirement re-execution—is the fundamental mechanism of the DIVA microarchitecture [2, 5]. By providing a cheap and uniform way for detecting any execution errors, it enables the construction of structures that trade correct execution on rare corner cases for simplified design, higher performance, and lower power. Gharachorloo et. al. [7] proposed pre-retirement re-execution as a way of reconciling speculative execution with sequential consistency, but dismissed the idea as too inefficient relative to load queue snooping, which is now used in most processors that support sequential (or nearly sequential) consistency [8, 18]. Recently, Cain and Lipasti [4] have revived this idea, but used aggressive filtering to greatly reduce the number of re-executions and make the approach highly competitive with snooping. Serendipitously, their mechanism can also be used to detect intra-thread memory ordering violations (i.e., overly aggressive scheduling of loads with respect to older stores), obviating the need to snoop the LQ for any reason.

Our proposed implementation dovetails with that of Cain and Lipasti [4]. We use a similar technique—filtered load re-execution—to reduce store-queue and store-load forwarding complexity and to increase load execution bandwidth, rather than to increase (both internal and external) store bandwidth. We showed that the two techniques can be implemented together to form a data memory unit that efficiently handles both data forwarding (our contribution) and address disambiguation and memory consistency checking (theirs).

6. Conclusions

High-bandwidth, low latency load execution is an important component of high ILP processing and a scalable store-forwarding mechanism is a critical aspect of that component. The conventional forwarding mechanism is an associative store queue (SQ), but scaling an SQ effectively is challenging. On one hand, large windows and wide issue require large multi-ported SQs that can buffer many in-flight stores and service multiple loads per cycle. On the other, simple CACTI simulations show that given current technology and data cache sizes, an SQ must be small and single ported to avoid impacting load latency.

In this paper, we propose a new forwarding unit design. Our general approach is to remove as much complexity as possible from the load execution critical path, creating a forwarding unit that is fast and high bandwidth but some epsilon less than 100% correct. To this we couple a lightweight verification mechanism, specifically dependence-free re-execution, to detect, recover from, and learn to avoid the incorrect epsilon. This is the DIVA philosophy [2]. Our specific design exploits the fact that store-load forwarding is a stable property of program structure and thus is easily predictable. First, we filter loads which we predict not to need forwarding from accessing the queue, enabling the use of a single ported structure. Then, we filter store which we predict will not require forwarding predicted to not-forward load accesses to the SQ, enabling the use of a single ported structure. Then, we use the very same predictor to filter store entries from the SQ, enabling the use of a smaller structure. Both predictors are trained by the load re-execution mechanism. We also add a high-bandwidth, address-interleaved forwarding structure, the forwarding store cache (FSC), to allow "easy" forwarding instances to proceed without the SQ. The combination of small size and a single associative read port results in a queue that doesn't stretch load latency, while load and store filtering permit parallel load access and a large window respectively. Some added machinery is required to make the whole mechanism function smoothly. A retirement store queue (RSQ) that does not participate in execution holds the unfiltered store stream and guarantees in-order retirement for stores, while a novel adaptation of Bloom filtering dramatically reduces the number of loads that must be re-executed to ensure correctness. Our technique meshes well with value based memory ordering (VBMO), a recently proposed scheme that uses limited load re-execution to address load queue scalability, and actually improves that technique in two ways.

Performance simulations on the SPEC2000 and MediaBench programs show that our load and store filtering techniques achieve performance that is within 2% of an ideal 2-cycle, 2-ported, infinite-entry SQ using an associative queue with only 16-entries and a single read port. Load filtering effectively compensates for the lost read port, store filtering effectively compensates for the lost capacity, and the FSC boosts both techniques. Capacity optimizations are magnified in multithreaded execution scenarios which suffer from under-utilization due to static queue partitioning.

Certainly, there are improvements that can be made to our mechanism, and we have touched on some of them. FSC store filtering is one. Reducing re-executions for loads that forward using the FSC is another. Path-based load access and store entry prediction is yet another. These, and others yet not conceived of, are left for future investigation. Also left for future work is an exploration of the power and energy aspects of our design. With replays being relatively rare, and with careful adjustment of the auxiliary structures, we expect our technique to at worst be energy neutral, i.e., it may increase power consumption slightly but compensate for it in the form of reduced execution time. Experiments and analysis are needed to verify this expectation.

7. References

[1] H. Akkary, R. Rajwar, and S. Srinivasan. "Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors." In *MICRO-36*, Dec. 2003.

[2] T. Austin. "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design." In *MICRO-32*, Nov. 1999.

[3] B. Bloom. "Space/time tradeoffs in hash coding with allowable errors." *CACM*, 13(7):422–426, Jul. 1970.

[4] H. Cain and M. Lipasti. "Memory Ordering: A Value Based Definition." In *ISCA-31*, Jun. 2004.

[5] S. Chatterjee, C. Weaver, and T. Austin. "Efficient Checker Processor Design." In *MICRO-33*, Dec. 2000.

[6] G. Chrysos and J. Emer. "Memory Dependence Prediction using Store Sets." In *ISCA-25*, Jun. 1998.

[7] K. Gharachorloo, A. Gupta, and J. Hennessy. "Two Techniques to Enhance the Performance of Memory Consistency Models." In *ICPP*, Aug. 1991.

[8] Intel Corporation. *Pentium Pro Family Developer's Manual*, 1996.

[9] C. Lee, M. Potkojnak, and W. Mangione-Smith. "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems." In *MICRO-30*, Dec. 1997.

[10] D. Marr, F. Binns, D. Hill, G. Hinton, D. Koufaty, A. Miller, and M. Upton. "Hyper-threading Technology Architecture and Microarchitecture." *Intel Technology Journal*, 6(1), Feb. 2002.

[11] A. Moshovos, S. Breach, T. Vijaykumar, and G. Sohi. "Dynamic Speculation and Synchronization of Data Dependences." In *ISCA-24*, Jun. 1997.

[12] I. Park, C. Ooi, and T. Vijaykumar. "Reducing Design Complexity of the Load/Store Queue." In *MICRO-36*, Dec. 2003.

[13] S. Raasch and S. Reinhardt. "The Impact of Resource Partitioning on SMT Processors." In *PACT-12*, Sep. 2003.

[14] S. Sethumadhavan, R. Desikan, D. Burger, C. Moore, and S. Keckler. "Scalable Hardware Memory Disambiguation for High ILP Processors." In *MICRO-36*, Dec. 2003.

[15] P. Shivakumar and N. Jouppi. "CACTI 3.0: An Integrated Cache Timing, Power, and Area Model." Technical report, COMPAQ Western Research Laboratory, 2001.

[16] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor." In *ISCA-23*, May 1996.

[17] D. Tullsen, S. Eggers, and H. Levy. "Simultaneous Multithreading: Maximizing On-Chip Parallelism." In *ISCA-22*, Jun.

1995.
[18] K. Yeager. “The MIPS R10000 Superscalar Microprocessor.” *IEEE Micro*, Apr. 1996.
[19] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan. “Speculation Techniques for Improving Load-Related Instruction Scheduling.” In *ISCA-26*, May 1999.