Store Vulnerability Window (SVW): Re-Execution Filtering for Enhanced Load Optimization

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

Abstract

The load-store unit is a performance critical component of a dynamically-scheduled processor. It is also a complex and non-scalable component. Several recently proposed techniques use some form of speculation to simplify the load-store unit and check this speculation by re-executing some of the loads prior to commit. We call such techniques load optimizations. One recent load optimization improves load queue (LQ) scalability by using re-execution rather than associative search to check speculative intra- and inter- thread memory ordering. A second technique improves store queue (SQ) scalability by speculatively filtering some load accesses and some store entries from it and re-executing loads to check that speculation. A third technique speculatively removes redundant loads from the execution engine; re-execution detects false eliminations.

Unfortunately, the benefits of a load optimization are often mitigated by re-execution itself. Re-execution contends for cache bandwidth with store commit, and serializes load re-execution with subsequent store commit. If a given load optimization requires a sufficient number of load re-executions, the aggregate re-execution cost may overwhelm the benefits of the technique entirely and even cause drastic slowdowns.

Store Vulnerability Window (SVW) is a new mechanism that significantly reduces the re-execution requirements of a given load optimization. SVW is based on monotonic store sequence numbering and an adaptation of Bloom filtering. The cost of a typical SVW implementation is a 1KB buffer and a 16-bit field per LQ entry. Across the three optimizations we study, SVW reduces re-executions by an average of 85%. This reduction relieves cache port contention and removes many of the dynamic serialization events that contribute the bulk of re-execution's cost, allows these load optimizations to perform up to their full potential. For the speculative SQ, this means the chance to perform at all, as without SVW it posts significant slowdowns.

1. Introduction

The load-store unit is a complex yet performance critical component of a dynamically scheduled processor. It supplies values to in-flight loads and enforces the appearance of a sequential memory access stream. Both store-load forwarding and memory ordering require expensive associative searches. In forwarding, loads search older in-flight stores. In ordering, stores (in some processors loads too) search younger in-flight loads.

The non-scalability of associative search has led researchers to propose techniques that use speculation to simplify some aspect of the load-store unit and then, immediately prior to commit, re-execute some or all of the loads—comparing the re-executed values to the original ones—to verify the speculation [2, 6, 9]. We dub such techniques ***load optimizations***. One recent load optimization improves load queue (LQ) scalability [6] by using re-execution—rather than associative search—to verify intra- and inter- thread memory ordering speculation. A second optimization improves store queue (SQ) scalability [3, 20] by speculatively filtering both load accesses and store entries from the SQ. Load re-execution verifies both forms of speculation. Redundant load elimination [17, 19, 23] speculatively removes some loads from the execution engine entirely, presenting the load-store unit with a reduced load stream. Re-execution detects false eliminations.

Load re-execution is a simple way to uniformly detect many forms of load mis-speculation. It requires only data cache access and it raises no costly false alarms. However, it has some disadvantages. It contends with store commit for cache bandwidth and introduces a new critical loop [5]: a store may not commit until all previous loads have re-executed successfully. If a given load optimization requires the re-execution of a significant fraction of loads, the resulting contention and serializations may overwhelm the benefit the optimization itself provides. The non-associative LQ and redundant load elimination techniques succeed largely because they have "natural" filters that limit re-executions. The speculative SQ technique has no natural filter and its re-executions degrade performance significantly.

This work introduces the ***Store Vulnerability Window (SVW)***, a filter that significantly reduces the number of loads that must re-execute to support a given load optimization. SVW exploits the observation that even an optimized load (e.g., an eliminated load) need not re-execute if it reads an address that has not been written to in a long time. SVW uses a store sequence numbering scheme and an adaptation of Bloom filtering. By reducing cache bandwidth contention and removing dynamic load re-execution/store-commit serializations, SVW improves the performance of optimizations that have natural re-execution filters and "enables" optimizations that have no natural filter by making them profitable. Performance simulations on the SPEC2000 integer programs show that a 1KB SVW filter reduces re-executions associated with these three load optimizations by an average of 85%, and brings their performance close to what it would be with ideal (instant latency, infinite bandwidth) re-execution.

0-7695-2270-X/05/$20.00 (C) 2005 IEEE

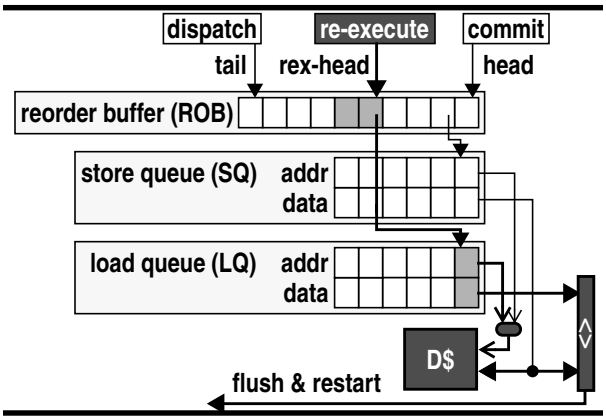

FIGURE 1. Pipeline with load re-execution engine. Re-execution shares a data cache port with store retirement.

2. Background

SVW optimizes load re-execution, which itself is only meaningful when coupled with a load optimization. In this section, we review a simple load re-execution architecture and three load optimizations that exploit it.

2.1. Load Re-Execution

The commit pipeline of a superscalar processor writes stores to the data cache in program order. When a store commits, its corresponding address/value pair is read from the head of the store queue (SQ) and sent to a data cache write port. Processors typically support only one data cache write per cycle as stores typically account for only 15% of dynamic instructions.

Figure 1 shows a conventional pipeline augmented with load re-execution. The added/modified paths/structures are bolded/shaded. The primary addition is a re-execution pipeline which processes loads and stores. Like the commit pipeline, the re-execution pipeline processes instructions in program order and stalls at the first non-completed instruction (load re-execution can actually begin as soon as the load address is available, but this complicates the implementation). A separate ROB pointer, rex-head, decouples the re-execution pipeline from the commit pipeline. To re-execute loads, we convert the data cache write port to a read/write port. Store commit and load re-execution arbitrate for this port with commit having priority. The re-execution pipeline can buffer a few stores internally to allow loads to re-execute before all older stores have committed. The re-execution pipeline flags loads that re-execute successfully, i.e., read the same value as the original execution. The commit pipeline flushes the processor when it encounters a load that re-executed unsuccessfully.

For most load optimizations, the re-execution pipeline reads load addresses and values from the LQ. For optimizations that eliminate the LQ or—like redundant load elimination [17, 19, 23]—filter some loads from it, we elongate the re-execution pipeline to read the base address and value from the register file.

Natural re-execution filters. Some load optimizations only affect a subset of the loads and only this subset needs to re-execute. A load optimization that can naturally identify the loads it affects and prune re-executions—and most can do this to some degree—is said to have a ***natural re-execution filter***. We call loads that must re-execute ***marked loads***.

Performance impact. If no loads re-execute, the re-execution pipeline acts as a trivial one-stage extension to the commit pipeline. If some or all loads re-execute, the elongated pipeline reduces the effective capacities of structures that hold in-flight instruction state: the ROB, LQ, SQ, and register file. Two larger costs are contention for the data cache read/write port—the re-execution pipeline stalls during store commit—and the introduction of a serialization scenario, i.e., a critical loop [5]. Counter-intuitively, this scenario does not involve the re-execution of dependent loads. Dependent loads can re-execute in parallel with the consumer re-executing using producer's original output value as its input; this is "dependence-free checking" [2]. The serializing constraint is that a store may not commit until all prior loads have successfully re-executed. It turns into a critical loop because data cache access is typically a multi-cycle operation. This serialization cannot be mitigated by a store buffer, which exposes stores externally and from which stores cannot be aborted.

2.2. Non-Associative Load Queue (NLQ)

Conventional processors that issue loads speculatively enforce the appearance of sequential memory order using a load queue (LQ). The LQ tracks the addresses of in-flight loads. To enforce intra-thread ordering, completed stores associatively search the LQ for younger loads to the same address that issued prematurely. A match flushes the load and all subsequent instructions. If the LQ contains values in addition to addresses, some flushes may be avoided as the search procedure could ignore ordering violations from silent stores [13]. To enforce inter-thread ordering, the entire LQ is searched when a cache line is invalidated by a write from another thread. Loads to that line that have issued are flushed. Inter-thread searches cannot incorporate values to eliminate false flushes due to silent stores or false sharing since block invalidations are not accompanied by values that can be used for comparison. While neither ordering operation is on the load execution critical path, LQ search is expensive because the LQ is large, searches frequent, and false flushes costly.

To avoid unnecessary flushing associated with inter-thread memory ordering speculation, Gharachorloo et al. proposed replacing associative search with in-order load re-execution prior to commit [9]. He quickly dismissed the idea because without a natural re-execution filter, the bandwidth consumed by re-executing all loads outweighs the gains of reduced flushing. Cain and Lipasti made this approach competitive by introducing two heuristics that significantly reduce the number of re-executed loads [6]. To verify intra-thread ordering, only loads that issued in the presence of older stores with unresolved addresses are re-executed. For inter-thread

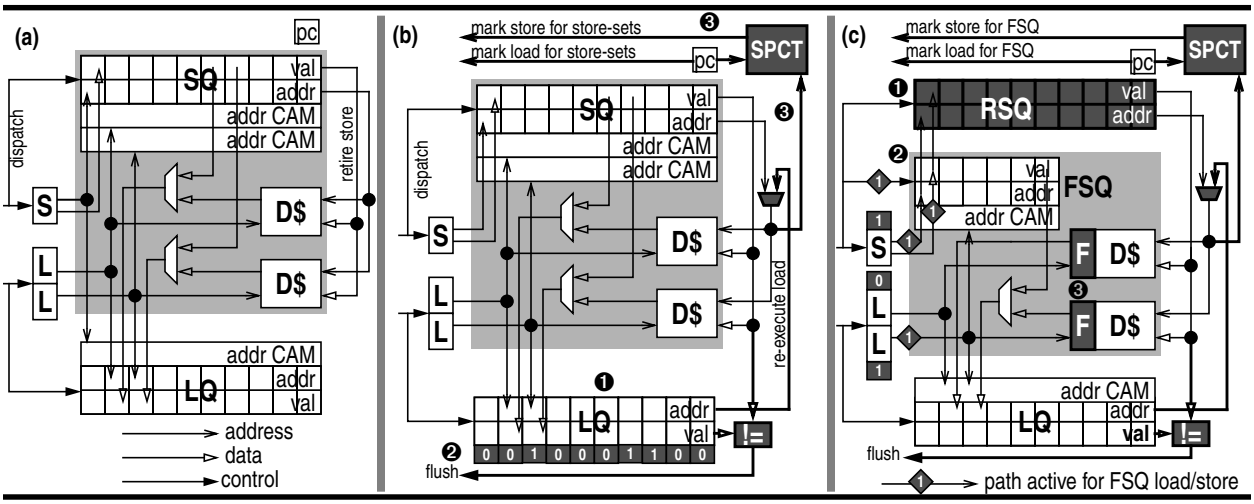

FIGURE 2. LEFT: conventional load-store unit with interleaved data cache, two load ports and one store port. **MIDDLE:** non-associative LQ design replaces LQ searches with re-execution. **RIGHT:** speculative SQ design splits SQ into a small, low-bandwidth forwarding queue (FSQ) and large non-associative retirement queue (RSQ). Re-execution checks that loads and stores are correctly steered to the FSQ. Steering is performed by a predictor which is trained by re-execution mechanism.

ordering, only loads that are in the window during a cache line invalidation—these are identified by remembering the value of LQ tail pointer at the time of an invalidation—are re-executed. Collectively, we call these optimizations the ***non-associative LQ (NLQ)***. Individually, we call them NLQ_{LS} (load speculation) and NLQ_{SM} (shared-memory). Cain and Lipasti show that NLQ_{LS} re-executes 2–15% of the loads and NLQ_{SM} re-executes an additional 20–40% [6].

All of our load optimizations build on the conventional load-store unit in Figure 2a, which executes two loads and one store per cycle. It includes a 2-way interleaved data cache, an SQ with two associative ports (for the loads), and an LQ with one associative port (for the store). The shaded box shows the components that sit on the load execution critical path: the data cache and SQ.

Figure 2b shows the NLQ organization. The added structures/paths are shaded/bolded. Note the absence of the LQ associative port (marker ❶) and the fact that the scheduler marks loads for re-execution (❷). One of the stated limitations of the originally NLQ proposal [6] is that it does not track the identities of stores that trigger flushes and can only train store-blind dependence predictors, rather than more refined store-load pair dependence predictors like store-sets [7]. We overcome this limitation using the ***store PC table (SPCT)***: a small, tagless table indexed by low-order address bits in which each entry contains the PC of the last retired store to write to a matching address. On a flush, the store PC is retrieved from the SPCT using the load address (❸).

2.3. Speculative store queue (SSQ)

Loads read values from either the SQ (if they read an address written by some older in-flight store) or the data cache (otherwise). Cache and SQ are accessed in parallel to minimize load latency and reduce scheduling complexity. Address banking is a straightforward way of providing high-bandwidth cache access, but is not easily reconciled with the age-ordering invariants that an SQ requires [22]. High-bandwidth SQ access is typically provided by multi-porting or replication. Both options are costly, especially for associative structures.

Recently, researchers [3, 20] have observed that an SQ serves two functions: (i) it buffers stores for in-order retirement, 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., to be slow). Implementing both functions in a single structure requires that structure to be both large *and* slow. Figure 2c shows one instance [20] of a design that exploits this observation by dividing the functions of a conventional SQ between two queues. A large retirement SQ (RSQ) contains all stores but does not support associative search. In fact, it is removed from the timing critical load execution path (❶). A small, low-bandwidth forwarding SQ (FSQ) implements forwarding. The FSQ requires fewer associative ports than a conventional SQ because only loads that read values from older stores access it. It requires fewer entries because only stores that forward values to loads are allocated entries in it (❷). Actually, the FSQ handles only a subset of the forwarding cases. A small, 8-entry unordered forwarding buffer that fronts each cache bank (❸) handles simple forwarding cases (i.e., unambiguous ones which execute in order anyway). Loads that execute incorrectly in this structure are subsequently steered to the FSQ. FSQ steering uses a simple predictor, a single bit per instruction in the instruction cache. Initially, all bits are clear and no loads/stores access/enter the FSQ. When re-execution detects a missed forwarding instance, the participating load and store are tagged for future FSQ access/entry. Because forwarding patterns are stable and the static set of forwarding stores and loads is small—these phenomena are well known and exploited by memory-

ordering speculation [7, 14]—the predictor trains quickly and mis-speculation rates are low.

Unlike NLQ, SSQ does not have a natural re-execution filter, i.e., it re-executes all loads (notice loads in the LQ are not explicitly marked). Even though most dynamic loads—over 80% in many applications—are instances of static loads that never read from older stores and don't access the FSQ, they must always re-execute to ensure that the first store-forwarding instance is not missed. Loads that access the FSQ must also re-execute because FSQ store membership is also speculative.

2.4. Redundant load elimination (RLE)

Compilers have a difficult time eliminating dynamically redundant loads due to the limitations of static alias analysis. Several proposed hardware mechanisms dynamically detect redundant loads and remove them from the execution engine, reducing both redundant load effective latency and load unit bandwidth demand.

Redundant load elimination (RLE) comprises two redundancy scenarios. Load reuse exploits redundancy between two loads by renaming the output register of the second load to point to the output register of the first. Speculative memory bypassing exploits store-load communication by setting the output register of the load to point to the data input register of the store. We look at one implementation of load reuse and speculative memory bypassing, register integration [19], which detects reuse scenarios for all instructions using an integration table (IT) that tracks physical register dependences of recent instructions. An instruction is redundant if it performs the same operation on the same physical register inputs as an instruction which has an IT entry.

Redundant, un-executed loads must re-execute to detect false eliminations, e.g., load reuse in the presence of an unaccounted for store that intervenes between the original and redundant loads. Invalidating IT entries by snooping store addresses is insufficient because the addresses of stores older than a redundant load may become available only after the load has been eliminated. Because eliminated loads do not execute and thus have empty LQ entries, RLE requires an extended re-execution pipeline which reads load addresses and values from the register file.

RLE has a natural re-execution filter; only eliminated loads re-execute. However, it typically eliminates 25–40% of all dynamic loads yielding a substantial re-execution stream.

3. The SVW Re-Execution Filter

Store vulnerability window (SVW) is a mechanism that enhances the natural re-execution filter of a given load optimization (e.g., NLQ and RLE), or provides one if the optimization does not have one (e.g., SSQ). SVW exploits the observation that even an optimized load—e.g., a speculative load under NLQ_{LS} or an eliminated load under RLE—should not have to re-execute if it reads an address that hasn't been written (or invalidated) in a long time.

Basic SVW scheme. This basic SVW mechanism is common to all load optimizations that we studied. SVW assigns each dynamic store a monotonically increasing sequence number, the ***store sequence number (SSN)***. For now, we assume that SSNs have infinite width and do not wrap. Store SSN's need not be explicitly represented. It is sufficient to explicitly represent the SSN of the last retired store, $\mathbf{SSN_{RETIRE}}$. The SSN of any in-flight store can be computed using this global value and the store's relative-to-head position in the SQ. For convenience, we also often refer to SSN_{RENAME}, the SSN of the youngest store in the window. This is just $\mathbf{SSN_{RETIRE}}$ **+ SQ.OCCUPANCY**.

SVW uses SSNs to associate with each dynamic load a dynamic window of stores to which that load is made vulnerable by the optimization; we refer to this window of stores as a load's ***store vulnerability window (SVW)***. Intuitively, a load is only vulnerable to stores that are older than itself yet not so old that they committed to the data cache before the associated optimization took effect. For convenience, we define a load's SVW as the SSN of the youngest older store to which the load is *not* vulnerable. We add an SVW field to the LQ; this field is set at dispatch (**ld.SVW = some-SSN**) but can be updated if a subsequent action shrinks the SVW.

Defining an SVW for each dynamic load is only half the equation. The other half is a small, tagless table indexed by low-order address bits—similar to the SPCT—in which each entry holds the SSN of the last retired store to write to any partially matching address. We call this table the ***store sequence Bloom filter (SSBF)***. Here we use the term Bloom filter [4] to mean a filter in which aliasing can only produce false positives.

SVW adds one stage to the re-execution pipeline; this stage immediately precedes data cache access. Recall, the re-execution pipeline processes loads and stores in program order. In the SVW stage, a store writes its SSN into the SSBF entry corresponding to its address (mnemonically, **SSBF[st.addr] = st.SSN**). A marked load

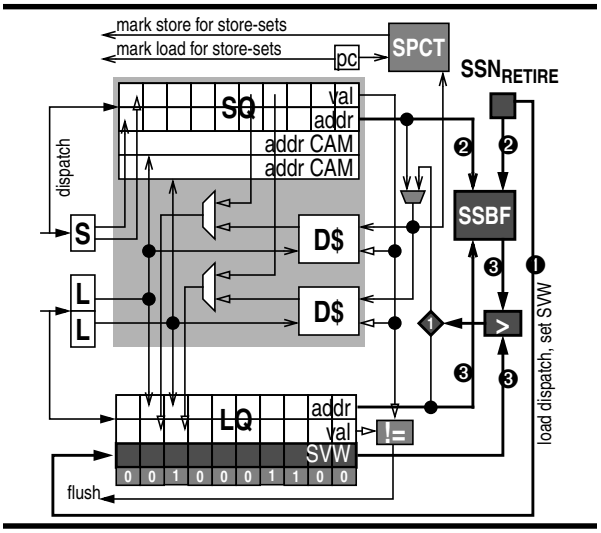

FIGURE 3. NLQ with SVW.

uses its address to read the SSBF and evaluates the re-execution filter test: **SSBF[ld.addr] > ld.SVW**. A positive test implies that the load probably conflicts with a store to which it is vulnerable; the load must re-execute to detect possible mis-speculation. A negative test unambiguously means that no store conflict occured; the load is immediately tagged as having sucessfully re-executed, skips the data-cache access stages in the re-execution pipeline, and may commit the following cycle.

The additional structures and paths are shown in Figure 3 in the context of NLQ. SVW-specific hardware is shaded/bolded. They new structures are: i) the global counter $\mathbf{SSN_{RETIRE}}$, ii) an SVW field in the LQ, iii) the SSBF, and iv) a re-execution test circuit. The new paths are active at load dispatch (❶), store SVW, i.e., SSBF update (❷), and load SVW, i.e., SSBF lookup and re-execution filter test (❸).

3.1. SVW for NLQ_{LS}

NLQ_{LS} re-executes loads that issued in the presence of older stores with unknown addresses. The memory scheduler recognizes and marks these loads. To this natural filter, we add SVW so that a load re-executes only if it issued speculatively *and* if a store to which it is vulnerable wrote to a colliding address.

The key step of implementing an optimization-specific SVW scheme is operationally defining the store window to which a given load is vulnerable. In load-speculation, a load is vulnerable to all older stores that were in-flight at the time it dispatched. Almost no load optimization we can think of makes loads vulnerable to stores younger than themselves. Load speculation in particular does not make loads vulnerable to stores that committed prior to the load being dispatched. The physical definition of the load-speculation SVW for a given load is the SSN of the last retired store. At dispatch, we set $\mathbf{ld.SVW = SSN_{RETIRE}}$.

In load-speculation, store-load forwarding is an action that shrinks the vulnerability window. When a load reads from an in-flight store, it becomes invulnerable to that store and all older stores. To capture this effect, we update the load's SVW to the SSN of the forwarding store, **ld.SVW = st.SSN**.

Working example. Figure 4a shows the four SVW events in the life of one dynamic load. Each snapshot shows three structures. The LSQ is on the left (we show the LQ and SQ as a single interleaved queue both for clarity and to save space, the example still works if they are separate). Values do not contribute to the example and so we only show addresses (letters A, B, C, D) and SSNs (numbers). Older instructions are to the right. Stores are shaded. The second structure is the global $\mathbf{SSN_{RETIRE}}$. Finally, the SSBF shows the SSNs of the last retired stores to write to each of the four addresses. The load of interest is the one on the left end of the LSQ.

In the first snapshot (❶), we dispatch the load and establish its vulnerability window by setting its SVW to $\mathbf{SSN_{RETIRE}}$, 62. The load is vulnerable to all stores younger than (i.e., with SSNs greater than) 62.

In the next snapshot, the LSQ head and tail have advanced—notice the updated state of $\mathbf{SSN_{RETIRE}}$ and the SSBF—but the interesting event is the execution of the load. Notice, the load executes in the presence of older ambiguous stores (64 and 66), and so is marked for potential re-execution by the scheduler. Also notice, the load reads its value from store 65, which also references address A. This action means the load is no longer vulnerable to any stores older than 65, inclusive. We update the load's SVW to reflect this fact (❷).

In the third snapshot, store 66 writes to A, meaning that our load issued over-aggressively and must re-execute to detect this violation. When store 66 retires, it writes its SSN into the SSBF entry for address A (❸).

Finally, in the SVW stage of the re-execution pipeline, the load accesses the SSBF using its own address, A. As expected, the SSBF re-execution test indicates that the load must re-execute because it collided with store 66, to which it was vulnerable (❹).

A Second Example. Figure 4b shows alternatives for the final two snapshots in which the load collides with store 64 to which it is not vulnerable (this store is older than the store that forwarded the value to the load). This time, when the load checks the SSBF, it finds the

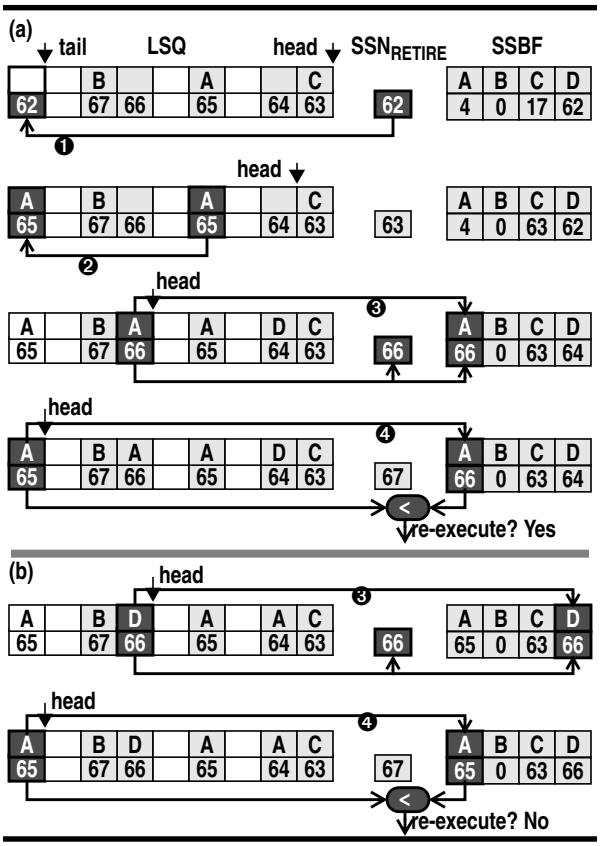

FIGURE 4. NLQ_{LS}+SVW working example shows the relevant SVW steps in the life of one dynamic load, the one on the left-most side of the LSQ. In (a), the steps are load dispatch, load execution, conflicting store retirement/SSBF update, and load SSBF lookup/re-execution. (b) shows alternatives for the last two steps.

SSN 65 in the entry for address A. The load skips re-execution because it is not vulnerable to stores 65 and older. Without SVW, this load re-executes.

3.2. SVW for NLQ_{SM}

NLQ_{SM} enforces inter-thread memory ordering by re-executing loads that are in-flight when a shared-memory coherence invalidation takes place. NLQ_{SM} differs from NLQ_{LS} in two ways. First, information about store addresses is available at the cache line rather than word granularity. Second, stores do not come from the same thread as the loads that are vulnerable to them such that there is no "natural" sequence numbering relationship between the two.

To account for the first difference, an invalidation updates multiple SSBF entries. We divide the SSBF into a number of banks that equals the number of words in a cache line. For store updates, only one bank is write-enabled. For invalidations, all banks are enabled.

To account for the second difference, we formulate a sequence numbering relationship between loads and invalidations. From a load's point of view, an invalidation acts like an asynchronous store from within the same thread. Taking this view, a load is vulnerable to all invalidations that occurred since it was dispatched so the SVW definition is actually the same as in NLQ_{LS}, **ld.SVW = SSN_{RETIRE}**. When an invalidation occurs, we pretend that it is an asynchronous store that potentially affects all in-flight loads. We therefore write into the SSBF an SSN that is higher than that of the youngest in-flight store: **SSBF[inval.addr] = SSN_{RENAME} +1**.

3.3. SVW for SSQ

The speculative SQ (SSQ) optimization uses the same SVW implementation as the one used by NLQ_{LS}. This is intuitive because both optimizations only make a given load vulnerable to older stores that were in the window at the time the load was dispatched. The difference between the two optimizations and their SVW implementations is that while NLQ_{LS} has a natural filter and uses SVW as an enhancer, SSQ uses SVW as an enabler. Without SVW, it would re-execute all loads.

3.4. SVW for RLE

Redundant load elimination (RLE) is also different from the first two optimizations. Under NLQ_{LS} and SSQ, a load is only vulnerable to stores that were in-flight at the time it was dispatched. In contrast, an eliminated load is vulnerable to all stores starting at the older load with which it is redundant. This difference requires a new mechanism for establishing load SVWs.

RLE is coordinated by the register renaming stage. Loads create IT entries that describe the operation they are about to perform and the physical register which will hold the result. Future loads search the IT and are recognized as redundant if they find tuples with matching "operation signatures." To establish SVWs we must pass SSN information from original load to redundant load via the IT. Non-redundant loads attach **SSN_{RENAME}** to the IT entry they create to mark the beginning of the vulnerability window for any future load that may reuse their result. An eliminated load takes its SVW from the IT entry it matches, **ld.SVW = IT-ENTRY.SSN**.

3.5. SVW for Multiple Optimizations

Composing the re-execution streams of multiple active load optimizations is easy: a load re-executes if it is marked by any optimization. Composing the SVW mechanisms of multiple optimizations is also easy; only per-load SVW definitions must be composed. Under all optimizations we have studied, a load is vulnerable to some contiguous window of stores that is immediately older than itself. In other words, for all optimizations the younger end of the window is fixed (it is the load itself) and only the older end varies. Intuitively, a load that is subject to multiple optimizations is vulnerable to the largest store window under any of them. Mnemonically, **$ld.SVW_{O1+O2}$ = MIN($ld.SVW_{O1}$, $ld.SVW_{O2}$)**.

An SVW for all four optimizations. Composing the unfiltered re-execution streams for all four optimizations we discussed is simple: SSQ marks all loads for potential re-execution. Setting the SVW for a given dynamic load is also straightforward. The SVW of non-eliminated loads is just **SSN_{RETIRE}**. Eliminated loads are not subject to SSQ or NLQ_{LS} because they do not execute, however they are subject to shared-memory invalidations. The SVW of an eliminated load is **MIN(IT-ENTRY.SSN, SSN_{RETIRE})**.

3.6. Implementation Issues

We now address two SVW implementation issues.

SSN wrap-around. In a real implementation, SSNs will have finite width. Sucessful handling of SSN wrap-around means avoiding comparing SSBF entries (i.e., retired store SSNs) that are slightly larger than zero with load SVWs that are slightly less than zero. A naive application of the re-execution filter test in this case would suggest that the load is safe because its SVW is much larger (i.e., younger) than the SSBF entry. However, in reality the SSBF entry may be slightly younger.

The following simple policy avoids such ambiguous comparisons. When **SSN_{RENAME}** wraps around to zero, we: i) drain the pipeline by waiting for all in-flight instructions to commit, ii) flash clear the SSBF, iii) flash clear the IT if the RLE optimization is enabled, and iv) resume dispatch. The effect of this policy is to ensure that no load has a vulnerability range that crosses the wrap-around point. In other words, no load has both an SVW slightly less than zero and an older store whose SSN is equal to or slightly larger than zero. Loads younger than the store whose SSN is zero are stalled at dispatch until that store reitres. As a result their SVW will be zero. Flash clearing the IT has the same effect on vulnerability ranges for eliminated loads.

Certainly, pausing to drain the pipeline at regular intervals reduces performance. However, the impact is minimal if SSNs are wide enough to make wrap-around infrequent. Our experiments show that performance

with 16-bit SSNs (wrap-around intervals of 64K stores) is only 0.2% lower than with infinite-width SSNs.

The experienced reader may wonder why SVW does not handle wrap-around using the "**log(Q.size)+1**-bit" scheme most processors use to compare the relative ages of instructions in circular queues. Simply, these schemes only work in queues where the logical ages they compare are never more than **Q.size** apart. The SSBF is not a queue. This allows it to be non-associative but makes it incapable of maintaining temporal distance guarantees.

Speculative SSBF updates. Figure 4 shows load SVW/re-execution and store SVW/commit as atomic relative to one another, a store does not update the SSBF until all previous loads have retired. Forcing this atomicity would actually exacerbate the load-to-younger-store serialization that re-execution filtering tries to avoid. In practice, we force loads and stores through the SVW stage in order, but allow stores to update the SSBF speculatively. This works because SSNs increase monotonically and because high SSBF entries are interpreted conservatively. If a wrong path store erroneously writes a high SSN into the SSBF, the only result may be a few superfluous re-executions. No necessary re-executions will be missed. Empirically, speculative SSBF updates increase re-executions relatively by 1–2% (e.g., by 0.1% if the base rate is 10%), a small price to pay for avoiding elongated serializations.

3.7. SVW vs. MCB/ALAT

Conceptually, SVW—which conservatively tracks address conflicts between every load and a given window of stores to which that load is vulnerable—is similar to the Memory Conflict Buffer (MCB) [8] or Intel Itanium's Advanced Load Alias Table (ALAT) [11]. However, SVW has an advantage over MCB/ALAT in that it does not require a load's address to be known before conflicts with that load can be tracked. SVW achieves this by splitting conflict tracking into two pieces: the SVW definition is attached to each load; the SSBF tracks store conflicts in a load indpendent way.

This organization allows SVW to support a wider range of load optimizations. Consider the three we discuss here. NLQ enforces memory ordering and requires loads to be tracked from the time they execute. Since load addresses are available at execution, an MCB/ALAT can be used. As store addresses become available, MCB/ALAT would check them against all tracked load addresses and force re-executions on matches. In order to not invalidate loads when they collide with younger stores, the MCB/ALAT could delay store address matching until store commit. This extension—which allows an MCB/ALAT to track load speculation in the presence of out-of-order stores and which is an application of the SVW concept—was proposed by Onder and Gupta [16]. In contrast, SSQ and RLE optimize value forwarding and require load-store collisions to be tracked starting at load dispatch. MCB/ALAT cannot be used for these optimizations.

In general, it seems that MCB/ALAT is appropriate for software optimizations where it logically tracks addresses between the commit of a speculative load and its non-speculative counterpart. SVW is more appropriate for hardware optimizations.

4. Experimental Evaluation

We use timing simulation to evaluate SVW's impact on three load optimizations: NLQ_{LS}, SSQ, and RLE. Our simulation infrastructure does not execute shared-memory programs so we do not evaluate NLQ_{SM}. Our goal is to show that SVW produces correct execution—flags all mis-speculations for re-execution—while reducing re-execution overhead.

Benchmarks. Our benchmarks are the SPEC2000 integer suite. We compile them using the Digital OSF C compiler with optimization flags –O3. We run them to completion, on the training input sets using 5% periodic sampling with 5% cache and branch predictor warm-up. Each sample contains 10M instructions.

Performance simulator. Our simulator executes the Alpha AXP user-level instruction set using the ISA and system call modules from SimpleScalar 3.0. We model a superscalar processor with MIPS-style register renaming, out-of-order execution, aggressive branch prediction and a two level on-chip memory system.

Because NLQ and SSQ target wide machines and RLE targets narrower machines we use two processor configurations. The common aspects are the memory system, branch predictor, and gross pipeline structure. The fetch unit has an 8K-entry hybrid direction predictor and a 2K entry, 2-way set-associative BTB, and can fetch past one taken branch per-cycle. The instruction and data caches are 32KB, 2-way set-associative, 2-cycle access. The L2 is 2MB, 8-way set-associative, 15 cycle access. Memory latency is 150 cycles. The L2 and memory buses are both 16B wide, the latter is clocked at one quarter processor frequency. The base pipeline has 15 stages pipeline (3 fetch, 2 decode, 2 rename, 2 schedule, 3 register read, 1 execute, 1 writeback, 1 commit). Re-execution adds two stages to the SSQ and NLQ_{LS} pipelines and four to RLE. Both configurations use store-sets [7] to manage load speculation. Both have a single store retirement port; dual ports improve the performance of only one benchmark (*vortex*) by 6% on the 8-wide machine. Where it is used, SVW adds an additional stage. Our baseline SVW configuration uses 16-bit SSNs and a 512-entry (1KB) SSBF. SSBF read/write bandwidths match load/store issue bandwidth.

Our NLQ/SSQ configuration is 8-way issue with a 512-entry ROB, 128-entry LQ, 64-entry SQ, 200 issue queue entries, and 448 registers. It issues 5 integer, 2 FP, 2 load, 2 store, and 1 branch per cycle. Our RLE configuration is 4-wide with 128-entry ROB, 32-entry LQ, 16-entry SQ, 50 issue queue entries, and 160 registers. It issues 3 integer, 1 FP, 1 load, 1 store and 1 branch.

4.1. SVW's Impact on NLQ_{LS}

Figure 5 shows load re-execution rates (percent of retired loads) and speedups (percent IPC improvement)

of four configurations. The baseline is the 8-way superscalar with a 128-entry LQ and one associative port, i.e., the ability to issue one store per cycle.

The first configuration is NLQ, in which the associative LQ port is replaced by re-execution, allowing this configuration to execute two stores per cycle. With a good natural filter—only loads that issue out-of-order with respect to older stores re-execute—the average re-execution rate is 7.4%. Only three programs re-execute more than 10% of their loads and only *twolf* re-executes 20%. These numbers are higher than those initially reported [6] because our window is larger and our machine wider. We also use a different ISA, Alpha as opposed to PowerPC.

Note, the filtered re-execution rate is (much) greater than the mis-speulation rate. Re-executions due to silent stores cannot be avoided. Also, the SSBF tracks SSNs at an 8-byte granularity and so is vulnerable to "false sharing" due to non-overlapping sub-quad writes.

With low re-execution rates the average gain from the additional store port are 0.3%. Nine programs suffer slight (less than 1%) slowdowns and *parser* shows a 3.5% slowdown stemming from an 8.5% re-execution rate. Slowdowns do not directly reflect re-execution rates because a large negative effect of re-execution is load re-execution/store retirement serialization; performance is not degraded if many loads re-execute but few are closely followed by stores.

The next two configurations add SVW. *SVW–UPD* does not update a load's SVW to the SSN of a forwarding store. *SVW+UPD* adds this simple extension. Even without forward updates, SVW reduces the average load re-execution rate from 7.4% to 2.0% with a maximum of 8.1% (*perl.d*). Most of the remaining re-executions can also be filtered using the "update SVW on store-forward" technique, which reduces re-executions further to 0.6% of all loads, with a maximum of 2.6% (again *perl.d*). This is a 92% reduction in the number of re-executions. With the forwarding optimization NLQ's performance improvement climbs to 1.3% with only one program (*gzip*) showing a slowdown of –0.2%.

The point of these experiments is not to show how good NLQ_{LS} is, but rather how close to ideal SVW allows NLQ_{LS} to perform. The final experiment (*+PERFECT*) implements NLQ_{LS} with perfect, zero-latency, infinite bandwidth re-execution. The average performance improvement of the ideal NLQ_{LS} is 1.4%. With SVW (1.3%), NLQ_{LS} nearly achieves that ideal.

4.2. SVW's Impact on SSQ

Figure 6 shows the results of similar experiments that measure SVW's impact on the speculative SQ (SSQ). Our baseline configuration is the 8-wide machine with a 64-entry associative SQ and two load ports, i.e., the SQ has two associative ports. CACTI simulations show that at 90nm, an SQ of this size has 1.7 times the access time as an 8KB single-ported data cache bank and its input/output routing network. Although our baseline configuration uses a 2-cycle cache, loads take 4 cycles due to the associative SQ.

The first configuration presented relative to this baseline (*SSQ*) is the design we sketched in Section 2.3. The 64-entry associative SQ is replaced with a 64-entry non-associative retirement store queue (RSQ) and a 16-entry single-ported forwarding store queue (FSQ). Again, CACTI simulations show that an FSQ of this size is required to match the access time of the cache banks. Here, two loads may issue per cycle, but only one may access the FSQ. Loads execute in 2 cycles.

Recall, SSQ has no natural re-execution filter; it re-executes 100% of the loads. In the graph, we break down re-executions to distinguish loads that access the FSQ (black portion at the bottom of the bar) from loads that either use best-effort store-forwarding or no store-forwarding at all (shaded portion at the top). Because all loads re-execute, this configuration—despite the 2 cycle reduction in load latency—yields an average slowdown of 16%, the maximum slowdown is 83% (*vortex*). Programs with high baseline IPCs like *bzip2*, *eon*, and *vortex* suffer the most. A single store retirement port does not supply enough bandwidth to retire all stores *and* re-execute all loads while matching execution throughput.

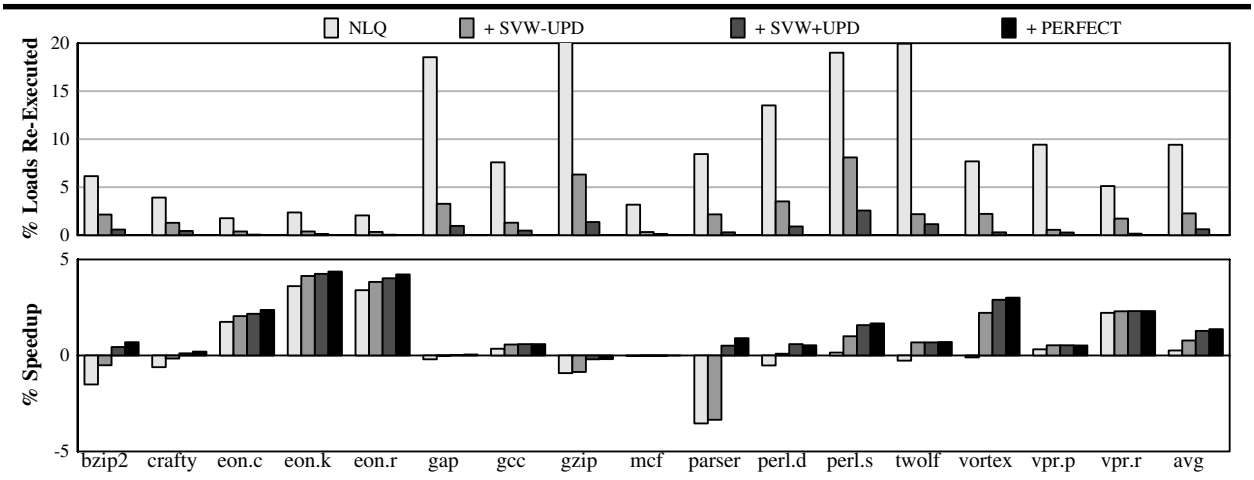

FIGURE 5. NLQ_{LS} re-execution rate (top) and performance (bottom).

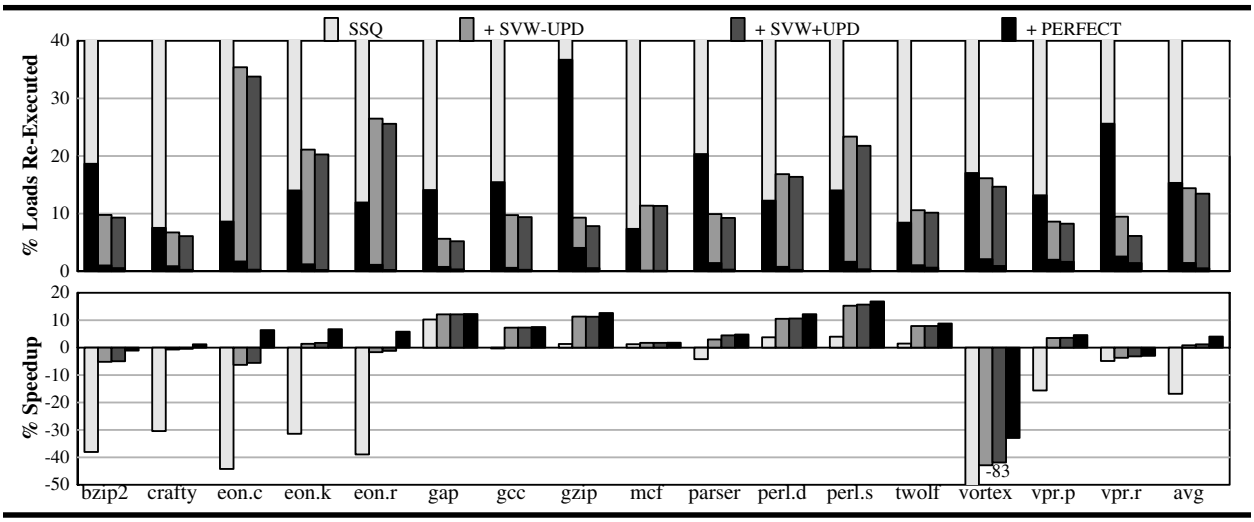

FIGURE 6. SSQ re-execution rate (top) and performance (bottom).

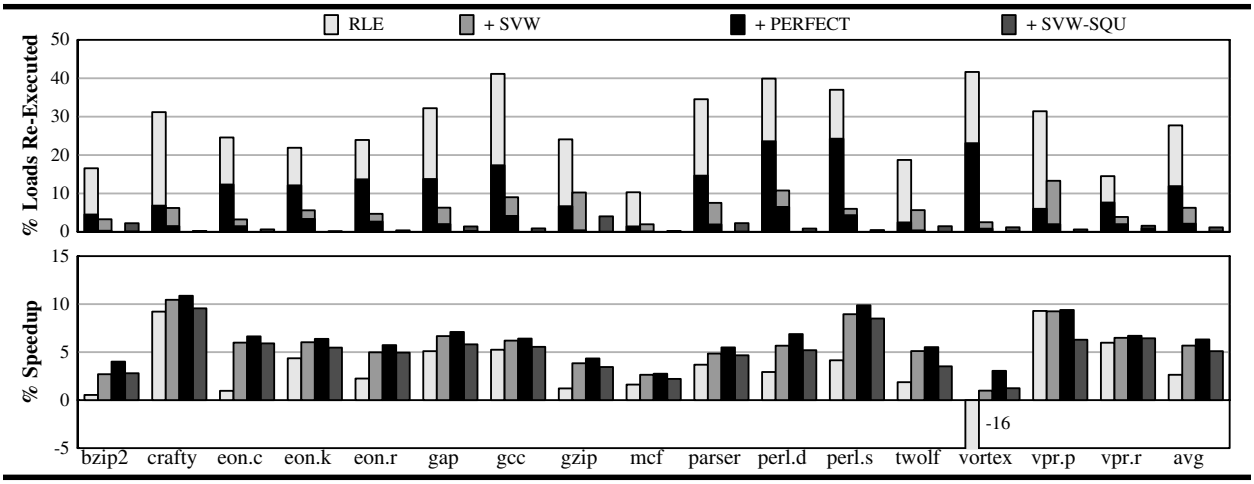

FIGURE 7. RLE re-execution rate (top) and performance (bottom).

The average re-execution rates for the *SVW–UPD* and *SVW+UPD* configurations are 15% and 13%, respectively, with maximum rates of 33% and 33% (both *eon.cook*). The "update SVW on store-forward" optimization does not help much here because it applies only to loads that are steered to the FSQ. It cannot be performed on loads that use best effort forwarding, which does not maintain the invariants required to support it. With an 87% reduction in re-execution, the average performance impact of SSQ turns from a 16% loss to a 1.2% gain. A few programs still post losses, but these are reduced. *Vortex* posts a 41% loss and no program and only *eon.cook* shows a loss as high as 6%.

The 1.2% average improvement with SVW is close to the 4% improvement SSQ can achieve even with perfect re-execution (our last configuration, *+PERFECT*). In fact, here we see that re-executions are only partially responsible for the large *vortex* slowdown. Even with perfect re-execution, *vortex* posts a 32% slowdown because it needs more ordered forwarding capacity than a 16-entry FSQ provides.

4.3. SVW's Impact on RLE

Figure 7 shows SVW's impact on redundant load execution (RLE). The baseline is the 4-wide machine with no elimination. The first configuration relative to this baseline (*RLE*) adds a 512-entry 2-way set-associate IT and a four stage re-execution pipeline. Eliminated loads do not fill their corresponding LQ entries with addresses and values. For re-execution, we read these from the register file which has a 2-cycle read latency. We add a single dedicated read port to the register file for this purpose. The address is read first; the value is read in time to be compared with the reloaded value.

Since RLE eliminates an average of 28% of the loads in a program (the maximum rate is 42% for *vortex*), this is also the re-execution rate. We break down re-executions according to whether the load was eliminated by redundancy with an older load (shaded portion at the top) or by speculative memory bypassing from an older store (black portion at the bottom). The corresponding average performance improvement is 2.6%, with a peak of 9.2% (*crafty* and *vpr.p*). The only pro-

gram to post a slowdown is *vortex* (16%). Again, *vortex* has a combination of high IPC and a high load elimination rate; a single cache port does not provide enough bandwidth to retire all of its stores and re-execute all of its eliminated loads.

RLE eliminates load latency from the execution dataflow graph and load bandwidth from the out-of-order execution core. It does not actually eliminate load execution, because all eliminated loads re-execute. We change that by adding SVW to the re-execution pipeline. With SVW, average re-execution rate drops to 6.3%, a 78% relative reduction. Average performance climbs to 5.7%, with a peak of 10.5% (*crafty*). *Vortex*'s slowdown disappears. Again with perfect re-execution (+PERFECT), average redundant load elimination performance improvement is 6.3%.

In our experiments, we noticed that most re-executions that SVW cannot filter correspond to eliminations of loads that are redundant with squashed versions of themselves, i.e., squash reuse [21]. SVW is disabled for squash reuse because of a corner case—a forwarding store exists on the squashed path but not the correct path—which the SSBF cannot capture. Our final experiment, *SVW–SQU*, disables squash reuse in the RLE configuration. Although re-executions drop markedly (from 6.3% to 1.2%) performance also drops slightly (from 5.7% to 5.1%). Eliminating a few last re-executions does not justify forfeiting squash reuse.

4.4. SSBF Configuration Sensitivity Analysis

Our default SVW configuration uses 16-bit SSNs and a 512-entry SSBF. We have already argued that SSN width has little performance impact. Here, we measure the effect of SSBF organization.

Figure 8 shows re-execution rates for a subset of the benchmarks and the SSQ optimization (which has the highest re-execution rates of the three). We measure six SSBF configurations: "simple" SSBFs with 128 (*128*), our default 512 (*512*), and 2K entries (*2048*); a configuration that uses a second 512 entry SSBF—this one indexed by the next 9 address bits—and re-executes a load only if "hits" in both filters (*Bloom*), a configuration that tracks conflicts at 4-byte rather than 8-byte granularity (*4-byte*), and an infinite 4-byte granularity SSBF (*Infinite*).

In SSQ, the average per-load SVW size is between 5

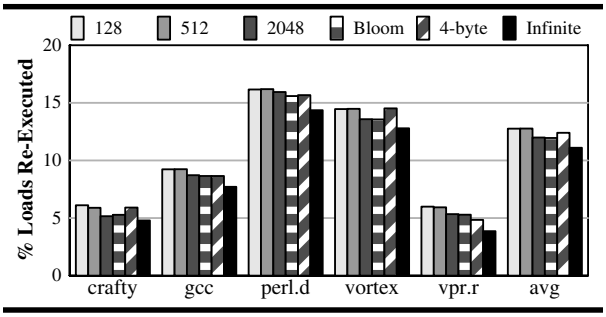

FIGURE 8. SSBF organization and re-execution rate.

and 15 stores. Clearly, the chances of a store to a non-conflicting address aliasing in a 512 entry table are small, 1/512. The chances that one of 15 stores has aliases are 1/32. Sophisticated SSBF configurations can reduce this aliasing rate, but because it is *a priori* so low, their impact on overall re-execution rate—and performance—is minimal. The largest performance difference for any program between a simple 512-entry 8-byte granularity SSBF and an infinite 4-byte granularity SSBF is 1.6% (*vpr.r*). The average is 0.3%.

5. Related Work

Re-execution and re-execution filtering. In-order pre-commit re-execution is the fundamental mechanism of the DIVA microarchitecture [2]. By providing a uniform way for detecting execution errors, it enables core designs that trade correct execution on rare corner cases for simplified design, higher performance, and lower power. Gharachorloo et al. [9] propose re-execution as a way of reconciling speculative execution with sequential consistency, but dismiss the idea as too inefficient relative to LQ snooping, which is used in most processors that support strong memory models [10, 24]. Cain and Lipasti [6] introduce filtering heuristics to reduce the number of re-executions and make re-execution competitive with snooping. Their mechanism can also detect intra-thread memory ordering violations, obviating the need to search the LQ for any reason. A general re-execution filtering mechanism for software optimization is the Memory Conflict Buffer (MCB) [8] or Advanced Load Alias Table (ALAT) [11]. Onder and Gupta show how the SVW concept can be applied to optimize these structures [16].

Load and Store Queue Optimizations. Several researchers have observed the difficulties in scaling the LQ and SQ to large sizes and bandwidths. Sethumadhavan et al. [22] reduce SQ access bandwidth by guarding the SQ with a Bloom filter that encodes the addresses of in-flight stores. They also reduce LQ size and search bandwidth using similar Bloom filter guarding. Park et al. [18] achieve the same bandwidth reductions using a store-load dependence predictor [7, 14, 25] rather than a Bloom filter and scale SQ size by chaining small SQ segments. Akkary et al. [1] use hierarchy to improve SQ scalability. A fast SQ holds the most recent stores while a larger, second-level SQ holds all in-flight stores. A Bloom filter eliminates most second-level SQ searches. The speculative SQ design we study here—combining a large, non-associative retirement queue and a small, associative forwarding queue—was proposed by Roth [20] and by Baugh and Zilles [3].

SVW uses Bloom filters in a different way than they have previously been used [1, 22]. Previous techniques use Bloom filters to guard SQ access. These filters are managed speculatively and out-of-order meaning their contents are difficult to maintain precisely and they are vulnerable to false positives from loads that match younger stores. They are also accessed on the load exe-

cution critical path. SVW's Bloom filter guards load re-execution, not SQ access. It is managed in order and only contains information about older stores; it is not vulnerable false positives from younger stores. It is not accessed on the load execution critical path.

Redundant Load Elimination. Our RLE implementation is register integration [19] which detects elimination opportunities using register dependences. Other implementations use address matching or memory dependence speculation [12, 15, 17, 23]. SVW can reduce re-executions in the two more general mechanisms [17, 23]. The two restricted speculative bypassing implementations [12, 15] operate strictly within the instruction window and can detect false eliminations using coventional memory ordering hardware.

6. Conclusions

The load-store unit is one of the most complex and non-scalable pieces of the execution core. Load optimizations use speculation to simplify some aspect of the core load-store unit. In-order pre-commit re-execution of some or all of the loads verifies the speculation. Recent examples of load optimizations include a non-associative load queue, a speculative store queue, and redundant load elimination. Unfortunately, re-execution itself dampens the benefits of load optimizations by contending for cache bandwidth with store retirement and by serializing load re-execution with subsequent store retirement. If a particular optimization requires a sufficient number of loads to re-execute, it may be swamped by their aggregate cost.

Store Vulnerability Window (SVW) reduces the re-execution requirements of a given load optimization significantly, by an average of 85% across three optimizations we have studied. By reducing cache port contention and removing many dynamic serialization events, SVW allows load optimizations to perform close to their full potential. SVW "enables" the speculative SQ optimization, which would not have been profitable otherwise. SVW is based on monotonic store sequence numbering and a novel adaptation of Bloom filtering. The cost of an SVW implementation is a 1KB table and an additional 16-bit field per LQ entry.

We are currently exploring SVW's impact on other load optimizations and evaluating its energy trade-offs. We are also studying SVW's potential as a replacement—not just a filter—for re-execution. In this setup, we forgo re-execution completely and simply use hits in the SSBF to trigger pipeline flushes and train the appropriate predictors.

Acknowledgements

We thank the reviewers for their comments. David Albonesi provided feedback on an early version of this work. Joel Emer and Milo Martin helped improve the final manuscript. This work was supported by NSF CAREER award CCF-0238203.

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] L. Baugh and C. Zilles. "Decomposing the Load-Store Queue by Function for Power Reduction and Scalability." In *IBM P=AC^2 Conference*, Oct. 2004.

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

[5] E. Borch, E. Tune, S. Manne, and J. Emer. "Loose Loops Sink Chips." In *HPCA-8*, Jan. 2002.

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

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

[8] D. Gallagher, W. Chen, S. Mahlke, J. Gyllenhaal, and W. Hwu. "Dynamic Memory Disambiguation Using the Memory Conflict Buffer." In *ASPLOS-6*, Oct. 1994.

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

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

[11] Intel Corporation. *IA-64 Application Developer's Architecture Guide*, May 1999.

[12] S. Jourdan, R. Ronen, M. Bekerman, B. Shomar, and A. Yoaz. "A Novel Renaming Scheme to Exploit Value Temporal Locality Through Physical Register Reuse and Unification." In *MICRO-31*, Dec. 1998.

[13] K. Lepak and M. Lipasti. "On the Value Locality of Store Instructions." In *ISCA-27*, Jun. 2000.

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

[15] A. Moshovos and G. Sohi. "Streamlining Inter-Operation Communication via Data Dependence Prediction." In *MICRO-30*, Dec. 1997.

[16] S. Onder and R. Gupta. "Dynamic Memory Disambiguation in the Presence of Out-of-Order Store Issuing." In *MICRO-32*, Nov. 1999.

[17] S. Onder and R. Gupta. "Load and Store Reuse using Register File Contents." In *ICS-15*, Jun. 2001.

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

[19] V. Petric, A. Bracy, and A. Roth. "Three Extensions to Register Integration." In *MICRO-35*, Nov. 2002.

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

[21] A. Roth and G. Sohi. "Register Integration: A Simple and Efficent Implementation of Squash Re-Use." In *MICRO-33*, Dec. 2000.

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

[23] A. Sodani and G. Sohi. "Dynamic Instruction Reuse." In *ISCA-24*, Jun 1997.

[24] K. Yeager. "The MIPS R10000 Superscalar Microprocessor." *IEEE Micro*, Apr. 1996.

[25] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan. "Speculation Techniques for Improving Load-Related Instruction Scheduling." In *ISCA-26*, May 1999.