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

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

Abstract

A high-bandwidth, low-latency load-store unit is a critical component of a dynamically scheduled processor. Unfortunately, it is also one of the most complex and non-scalable components. Recently, several researchers have proposed techniques that simplify the core load-store unit and improve its scalability in exchange for the ***in-order pre-retirement re-execution*** *of some subset of the loads in the program..We call such techniques* ***load/store optimizations****. One recent optimization attacks load queue (LQ) scalability by replacing the expensive associative search that is used to enforce intra- and inter- thread ordering with load re-execution. A second attacks store queue (SQ) scalability by speculatively filtering some load accesses and some store entries from it. The speculatively accessed, speculatively populated SQ can be made smaller and faster, but load re-execution is required to verify the speculation. A third uses a hardware table to identify redundant loads and skip their execution altogether. Redundant load elimination is highly accurate but not 100%, so re-execution is needed to flag false eliminations.*

Unfortunately, the inherent benefits of load/store optimizations are mitigated by re-execution itself. Re-execution contends for cache bandwidths with store retirement, and serializes load re-execution with subsequent store retirement. If a particular technique requires a sufficient number of load re-executions, the cost of these re-executions will outweigh the benefits of the technique entirely and may even produce drastic slowdowns. This is the case for the SQ technique.

Store Vulnerability Window (SVW) *is a new mechanism that reduces the re-execution requirements of a given load/store optimization significantly, by an average of 85% across the three load/store optimizations we study. This reduction relieves cache port contention and removes many of the dynamic serialization events that contribute the bulk of re-execution's cost, and allows these techniques to perform up to their full potential. For the scalable SQ optimization, this means the chnace to perform at all. Without SVW, this technique posts significant slowdowns. SVW is a simple scheme based on monotonic store sequence numbering and a novel application of Bloom Filtering. The cost of an effective SVW implementation is a 1KB buffer and an 2B field per LQ entry.*

1. Introduction

The load/store unit is one of the most complex yet performance critical components in a dynamically scheduled processor. The load/store unit supplies values to in-flight loads and enforces the appearance of a sequential memory access stream within a single thread. Both tasks, ***value forwarding*** and ***memory-ordering***, are complicated by the need to perform non-scalable associative searches against in-flight memory operations: value forwarding against older in-flight stores, memory-ordering against younger in-flight loads.

The non-scalability of associative search has led to the development of techniques that simplify some aspect of the core load-store unit in some way in exchange for the ***in-order pre-retirement re-execution*** [2] of some or all of the loads. We dub such techniques ***load/store optimizations***. One recent load/store optimization targets load queue (LQ) scalability [5]. It eliminates the associative search function of the LQ and instead uses re-execution to detect memory-ordering violations of both the intra-thread and inter-thread variety. A second load/store optimization targets store queue (SQ) scalability [3, 18]. This optimization implements the associative forwarding functionality of the SQ using a small SQ to which both load access and store entry are speculatively filtered. Load re-execution is required to verify this speculation. A third load/store optimization is the previously proposed redundant load elimination or load reuse [15, 17, 20]. Redundant load elimination removes some loads from the execution engine entirely, presenting the load/store unit with a reduced load stream. Here, re-execution is used to detect and recover from false eliminations.

Load re-execution is a safe and simple way to detect violations in any load/store optimization. It is conceptually easy to implement because it requires only data cache access and no information from data cache stores. And, by definition, it raises no costly false alarms. However, re-execution has some disadvantages as well. Primarily, it contends with store retirement for cache bandwidth and it introduces a new critical loop [4] as a store may not retire—not even to a store buffer—until all previous loads have re-executed successfully. If a given optimization requires the re-execution of a significant number of loads, the resulting contention and serializations may overwhelm the performance benefit the optimization itself provides. The non-associative LQ and redundant load elimination techniques [5] succeed in large part because they have "natural" filters that limits re-executions and the impact of these effects. The SQ technique has no natural filter and its re-executions can degrade performance significantly.

This work introduces ***store vulnerability window (SVW)***, a filter that significantly reduces the number of loads that must be re-executed to support a given technique. By reducing cache bandwidth contention and removing dynamic load re-execution/store-retirement 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.

SVW is a simple technique that uses a monotonic store sequence numbering scheme and a novel application of Bloom Filtering. Performance simulations on the SPEC2000 integer programs show that a 1KB SVW filter can reduce re-executions associated with the three load/store optimizations we named by an average of 85%, and brings their performance to within 85% of performance with ideal, i.e., instant latency and infinite bandwidth, re-execution.

The next section provides background on re-execution and the three load/store optimizations. Section presents SVW and its implementation in conjunction with each optimization. Section 5 contains a performance evaluation.

2. Background

Figure 1a shows the retirement engine of a dynamically-scheduled *N*-way superscalar processor. This engine is responsible for (among other things) committing stores to the data cache in program order. The engine reads the first *N* ROB entries and retires the first *M* of these up to the first instruction that has not completed. If any of the M instructions is a store, the 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 a single data cache write per cycle as stores typically account for less than 15% of the dynamic instruction stream.

2.1. Load Re-Execution

As figure 1b shows, load re-execution is implemented by adding another pointer to the ROB—call it the ***re-execution head pointer***—and a ***re-execution engine*** that operates on the first *N* ROB entries starting at this pointer. Like the retirement engine, re-execution processes M instructions per cycle up to the first non-completed instruction. All instructions other than loads marked for re-execution are immediately annotated as having re-executed successfully, *i.e.*, with no detected error. Loads that are marked for re-execution are sent down a short re-execution pipeline. If a load/store optimization can naturally identify loads that can safely skip re-execution—most can do this to some degree—we say that the optimization has a ***natural re-execution filter***. The retirement engine is slightly modified so that the head pointer cannot advanced past a load that has been marked for re-execution and has not finished re-exe-

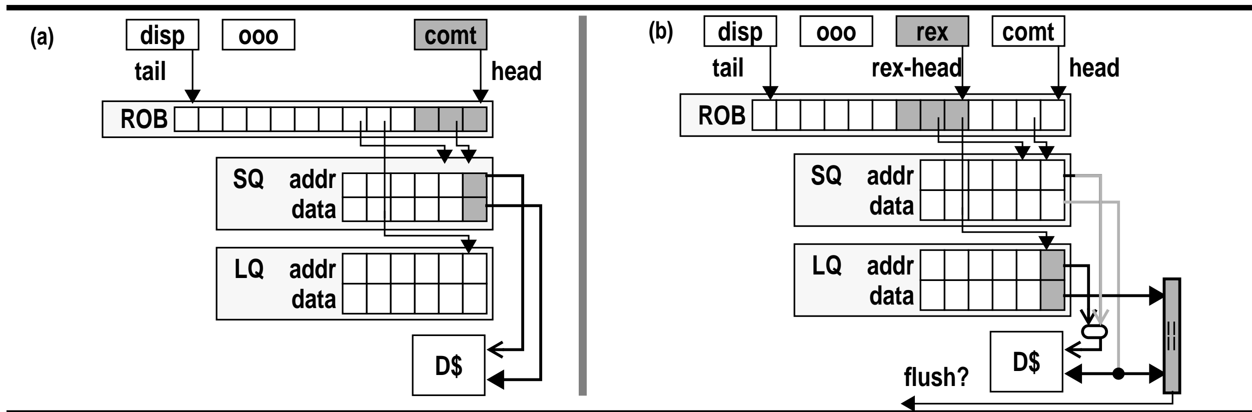

FIGURE 1. LEFT: conventional pipeline. **RIGHT:** pipeline with load re-execution engine. Rexecution shares the data cache write port with store retirement.

cuting and so that re-executed loads with value mismatch annotations trigger recovery actions, *i.e.,* ROB flushes. At any time, the ROB span between the head and re-execution head consists of completed instructions.

The re-execution pipeline re-executes marked loads on the erstwhile retirement data cache write port which is converted to a read/write port. The retirement engine and the re-execution engine arbitrate for this port with the retirement engine having priority; this guarantees that load re-execution and store retirement occur in order. The reloaded value is compared with the original value and mismatches are noted. They typically result in flushes at commit.

The precise structure and depth of the re-execution pipeline depends on the availability of addresses and values (for comparison) of re-executing loads. If these are available from the LQ, then the re-execution pipeline is as long as the data cache read latency. Optimizations that eliminate the LQ or—like load reuse [15, 17, 20]—filter some loads from the LQ require the re-execution pipeline to read the base address and value from the register file and to re-calculate the effective address. The depth of the re-execution pipeline in this case is the sum of the register file read latency and the address-generation/data cache read latency. This scenario also requires two additional register file read ports.

Performance impact. If no loads require re-execution, the re-execution engine essentially acts as a trivial one-stage extension to the retirement engine. If some or all loads must be re-executed, the effective lengthening of the pipeline will reduce the effective capacities of the structures that store in-flight instruction state: the ROB, LQ, SQ, and register file. Two much larger costs are contention for the data cache read/write port and the introduction of a serialization scenarios, i.e., a critical loop [4]. Counter-intuitively, this scenario does not involve the re-execution of dependent loads; a consumer instruction can re-execute in parallel with its data producer using the originally executed output value of the producer; this is “dependence-free checking” [2]. The serializing constraint is that a store may not retire until all prior loads have successfully re-executed, this constraint turns into a critical loop because data cache access is a multi-cycle event. Note, load re-execution/store-retirement serialization cannot be mitigated by a store buffer or any device which exposes the store externally and from which the store cannot be aborted.

2.2. Three Load/Store Optimizations

Load re-execution is only meaningful when coupled with some load/store optimization. We describe three techniques that optimize the core out-of-order load/store unit and use re-execution to detect loads that the optimized design did not handle correctly. Figure 2a shows the load/store unit of a processor that can issue two loads and one

store per cycle. The unit contains a 2-way interleaved data cache a store queue (SQ) with two associative ports (one per load) and a load queue 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.

Load/store optimization I: a non-associative load queue (LQ). The load queue (LQ) stores the addresses (and sometimes values) of in-flight loads. Associative LQ search is used to enforce both intra-thread and inter-thread memory ordering. To enforce intra-thread ordering, completed stores search the LQ for younger loads to the same address that issued pre-maturely. A match flushes the load and all subsequent instructions. If the LQ stores values in addition to addresses, some number of flushes may be avoided as the search procedure could recognize and ignore conflicts with silent stores [12]. 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 cache line that have already issued are flushed. Unlike searches for intra-thread memory ordering, inter-thread searches cannot incorporate values to eliminate false flushes due to silent stores from other threads or simple false sharing. While neither ordering operation is on the execution critical path *per se*, LQ search is typically quite expensive because the LQ is large, searches frequent, and false flushes costly.

To avoid unnecessary flushing associated with inter-thread memory ordering, Gharachorloo proposed replacing associative search with in-order pre-retirement load re-execution [8]. He quickly dismissed the idea because without a natural re-execution filter, the bandwidth consumption and serialization that result from re-executing all loads outweighs the gains of reduced flushing. More recently, Cain and Lipasti revived the idea and made it competitive by introducing two heuristic filters that significantly reduced the number of loads that must be re-executed [5]. To guarantee intra-thread ordering, only loads that issued in the presence of older stores with unresolved addresses are re-executed. To guarantee inter-thread ordering, only loads that are in the window during a cache line invalidation are re-executed. They show that 2–15% of loads must be re-executed to ensure intra-thread ordering and an additional 20–40% must be re-executed to enforce inter-thread ordering.

This organization is shown in Figure 2b. The added structures and paths are shaded/bolded; notice the absence of the LQ associative port. One of the stated limitations of the original non-associative LQ proposal 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]. We overcome this limitation here 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 any partially matching address. On a squash, the load address can be used to retrieve the store PC using the SPCT.

Load/store optimization II: a scalable store queue (SQ). Loads forward values from either the store queue (SQ), if they read an address that is written-to by some older in-flight store, or the data cache, otherwise. Cache and SQ are accessed in parallel to minimize latency and reduce scheduling complexity. High-bandwidth data cache access can be provided in a relatively straightforward way using address-interleaving (banking). Providing high-bandwidth SQ access is more difficult [19] 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 loads search the SQ associatively. Replicating the SQ is another costly option.

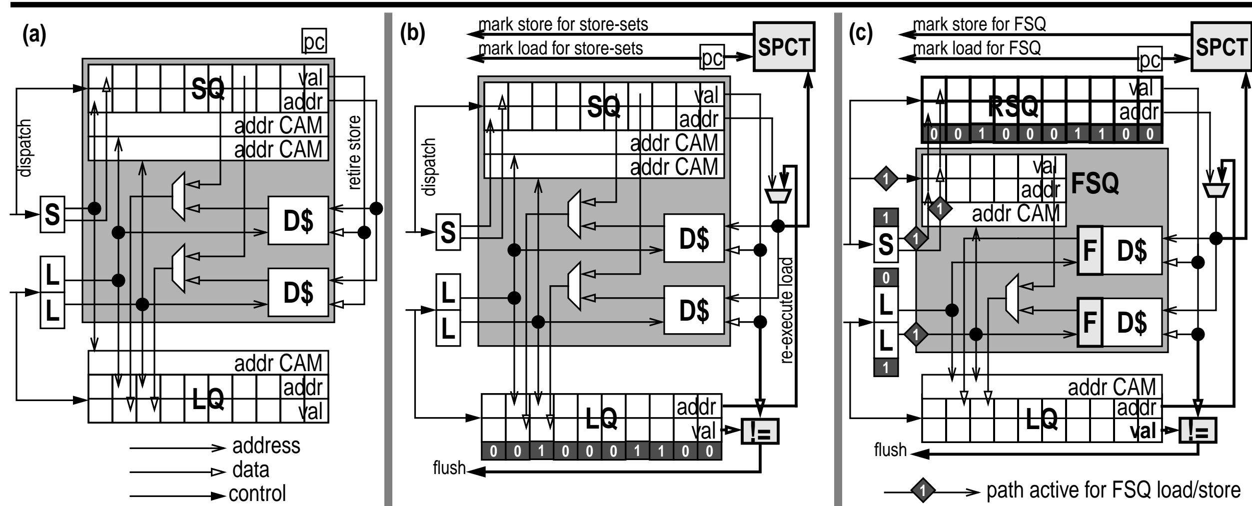

FIGURE 2. LEFT: A conventional load-store unit with two-way interleaved data cache, two load ports and one store port. **MIDDLE:** a non-associative LQ design which replaces LQ searches with filtered re-execution. **RIGHT:** scalable SQ design which splits SQ into a small, low-bandwidth forwarding queue (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 the re-execution mechanism.

Recently, several researchers [3, 18] have made the observation that 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 requires that structure to be both large and slow. Per this observation, an optimized load unit divides the two functionalities of a conventional SQ between two separate queues. This organization is shown in Figure 2c. In-order retirement is implemented using a large ***retirement store queue (RSQ)***, which contains all stores. Since the RSQ is not used to forward values, it is not equipped with associative-search machinery, and in fact is removed from the timing critical load execution path (note, it is not in the shaded box). Value forwarding to loads is implemented using a ***forwarding store queue (FSQ)***, which is essentially a conventional SQ but in miniature. It requires fewer ports than a conventional SQ because only loads that forward values from stores. It requires many fewer entries because only stores that forward values to loads are allocated entries in it. Actually, the FSQ only handles a subset of the forwarding cases. Simple forwarding cases—unambiguous ones which execute in order anyway—are handled by a small, 4-entry unordered forwarding buffer that fronts each cache bank. Loads that forward incorrectly in this structure are subsequently steered to the FSQ. FSQ steering uses a simple predictor, a single bit (FSQ or not) per instruction in the instruction cache. Initially, all bits are clear and no loads and stores access the FSQ. When re-execution detects a missed forwarding instance, the participating load and store are both marked for future access/entry to the FSQ: the load is identified trivially, the store PC is retrieved from the SPCT. Because forwarding behavior is stable and because the static set of forwarding stores and loads is small—these phenomena are well known and are exploited by other techniques like intelligent load speculation [13, 6] and speculative memory bypassing [14]—the predictor trains quickly and mis-speculation rates are low.

Unlike the non-associative LQ optimization, the scalable SQ optimization does not have a natural re-execution filter. Most dynamic loads—over 80% in many applications—are instances of static loads that never forward from older

stores; these never access the FSQ but must always re-execute to ensure that a potential forwarding instance is not missed. Loads that access the FSQ must also be re-executed because FSQ store membership is also speculative.

Load/store optimization III: redundant load elimination. Compilers have a difficult time eliminating dynamically redundant loads due to the limitations of static alias analysis and the requirement that any code transformation be valid along all statically possible execution paths. However, there are several proposed hardware mechanisms that can dynamically detect or predict redundancy scenarios and remove redundant loads from the out-of-order execution engine, reducing both the observed latency of the redundant load and the bandwidth demands on the load unit.

There are two different load redundancy scenarios: ***load reuse*** exploits a redundancy between two loads by setting the output of the second load to point to the output register of the first load, ***speculative memory bypassing*** exploits a store-load communication by setting the output register of the load to point to the data input register of the store. We look at a particular implementation of load reuse and speculative memory bypassing, register integration [17], which detects reuse scenarios for all instructions (not just loads) using a table that tracks the operations, physical register inputs, and physical register outputs 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 entry in this table. Register integration is essentially an implementation of dynamic instruction reuse [20] but for microarchitectures with pointer-based renaming. Other implementations of load reuse and speculative memory bypassing rely on memory dependence prediction [11, 14] or value comparisons [15] to identify redundant instances.

Redundant, un-executed loads must be re-executed to detect instances of false reuse, *e.g.*, a load reuse in the presence of an unaccounted for store that intervenes between the original and redundant loads. Invalidating entries in the reuse structure by snooping store addresses is insufficient because the addresses of stores older than a redundant load may become available only after the load has already been eliminated. As discussed earlier, load elimination requires an extended re-execution pipeline which reads load addresses and values from the physical register file. Register integration has a natural re-execution filter insofar as only redundant, un-executed loads must be re-executed. However, register integration often eliminates 25–40% of the dynamic loads in an application, resulting in a non-trivial re-execution stream.

3. The Store Vulnerability Window (SVW) Re-Execution Filter

Store window of vulnerability (SVW) is a new mechanism that enhances the natural re-execution filter of a given load/store optimization, e.g., the non-associative LQ or redundant load elimination, or provides one if the optimization doesn't naturally have one, e.g., the scalable SQ. SVW works by conservatively tracking address conflicts between every dynamic load and a given dynamic window of stores to which that load may be vulnerable. In that regard, SVW is similar to the memory conflict buffer (MCB) [7] or Intel IA-64's advanced load alias table (ALAT) [10]. However, while the definition and implementation of MCB/ALAT makes them suitable as support mechanisms for software load optimizations, SVW's definition and subsequent implementation makes it more suitable for hardware optimizations. A comparison of the two mechanisms is appropriate, and follows our description of SVW.

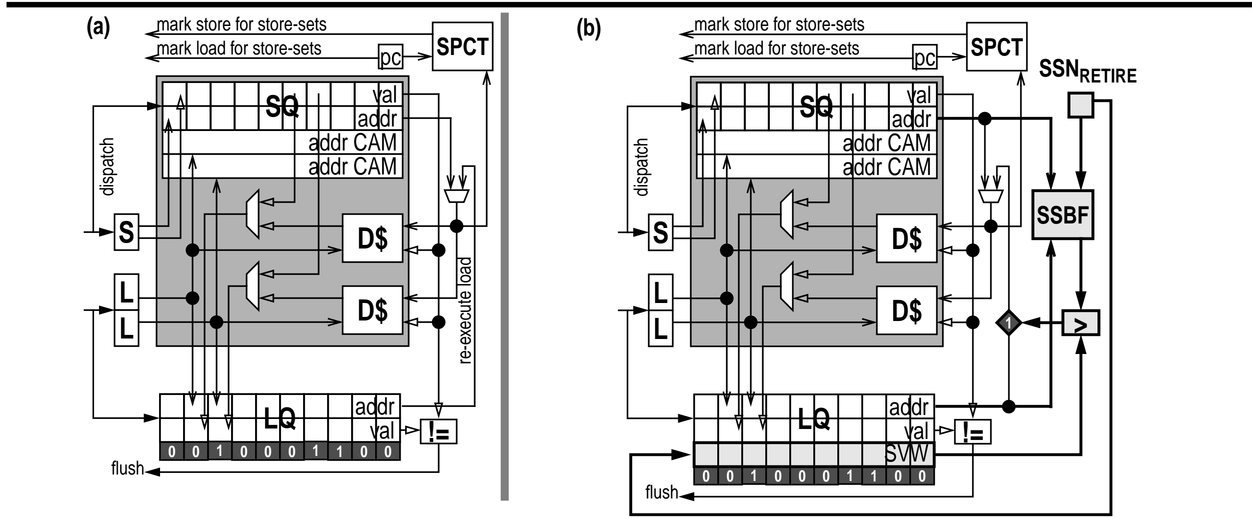

FIGURE 3. LEFT: non-associative LQ. **RIGHT:** non-associative LQ with SVW.

3.1. General Scheme

The basic SVW mechanism is common to all attached load/store optimizations.

Re-execution pipeline modifications. SVW adds a single stage to the re-execution pipeline. This stage immediately precedes data cache access (and follows register read and address re-calculation if those exists). The SVW stage determines whether a given load that was originally flagged for re-execution actually needs to re-execute. If the SVW stage determines the load is safe, the load is immediately marked as having successfully re-executed and may retire the following cycle. Unsafe loads are re-executed as usual. Unlike a conventional re-execution pipeline which only processes loads that are marked for re-execution, an SVW-enhanced re-execution pipeline must also process all stores and do so in-order with respect to re-execution eligible loads. This modification has negligible performance costs because stores only operate in the SVW stage; they do not re-execute and they don’t read the register file and recalculate their address even if a given load/store optimization requires re-executing loads to do so.

A store sequence numbering (SSN) scheme. The SVW mechanism is based on assigning each dynamic store a monotonically increasing sequence number, the ***store sequence number (SSN)***. Two components are built on top of this numbering scheme. The first is an optimization-specific scheme for associating with each dynamic load a dynamic window of stores to which that load is “vulnerable”; 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 retired to the data cache before the associated optimization took effect. As a result, a given load’s SVW can be simply defined as the SSN of the oldest store to which the load is vulnerable or, alternatively and often more conveniently, 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 typically set at the dispatch stage (**load.SVW = some-SSN**) but can be updated later if some action shrinks the vulnerability window.

The second component is implemented at the SVW stage within the re-execution pipeline. This component is a small, tagless table indexed by low-order address bits—similar in structure to the SPCT, actually—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). Recall, SVW forces all stores to go through the re-execution pipeline as well. The only thing stores do in the re-execution pipeline is write their SSN into the corresponding SSBF entry at the SVW stage (mnemonically, **SSBF[store.addr] = store.SSN**. A load that is marked for re-execution reads the SSBF entry corresponding to its own address. If **SSBF[load.addr] > load.SVW**, the load must re-execute because it potentially conflicts with a store to which it is vulnerable. Otherwise, no conflict is possible and the load may skip the final data-cache access stages in the re-execution pipeline and avoid the associated port contention and store serializations.

The additional structures and paths are shown in Figure 3b in the context of the non-associative LQ optimization. SVW-specific hardware is shaded/bolded. Figure 3a shows the non-associative LQ without SVW for comparison.

Quick summary. The SVW mechanism effectively checks for address collisions between a load and a given window of stores, and does so in a novel way. The SSBF implements address collision testing in a window unspecific way by tracking stores. At the same time, individual loads carry the definition of the window of stores they care about with them. The two components come together at the SVW stage. This organization allows a single store tracking structure, the SSBF, to effectively track conflicts for all dynamic loads even though each dynamic load is potentially vulnerable to a different dynamic window of stores. More importantly, it allows a given load's vulnerability window to be defined and tracked before the load's address is known. This is the key difference between SVW and MCB/ALAT and it is this difference that allows SVW to support microarchitectural optimizations.

In the following sections, we show how SVW enhances the re-execution based optimizations we described.

3.2. SVW for the Intra-Thread (Load-Speculation) Non-Associative LQ Optimization

To avoid expensive associative LQ searches on store completion, the non-associative LQ re-executes loads that issued speculatively, i.e., in the presence of older stores with unknown addresses. The memory scheduler recognizes these cases and flags the corresponding loads for re-execution. To this natural filter, we add SVW so that a load re-executes only if it issued speculatively ***and*** if an older store to which it is vulnerable wrote to a colliding address.

A given load/store optimization makes a load vulnerable to a given window of stores and the first step of implementing SVW is operationally defining this window. In load-speculation, a load is vulnerable to all older stores that were in-flight at the time the load was dispatched. Almost no load/store optimization we can think of makes loads vulnerable to stores younger than themselves and load-speculation in particular does not make loads vulnerable to stores that retired prior to the load in question being dispatched. The physical definition of the "load-speculation" SVW for a given load is the SSN of the last retired store, which we refer to by the mnemonic $\mathbf{SSN_{RETIRE}}$. At dispatch, we set the load's SVW by setting $\mathbf{load.SVW = SSN_{RETIRE}}$.

At the SVW stage, a completed store writes its SSN into the SSBF at the entry corresponding to its own address **SSBF[store.addr] = store.SSN**. A load that is flagged for re-execution accesses the SSBF using its own address and re-executes if **SSBF[load.addr] > load.SVW**. A short working example should clarify things.

Working example. Figure 4a shows the four SVW relevant events in the life of a single dynamic load. Each snapshot shows the contents of 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

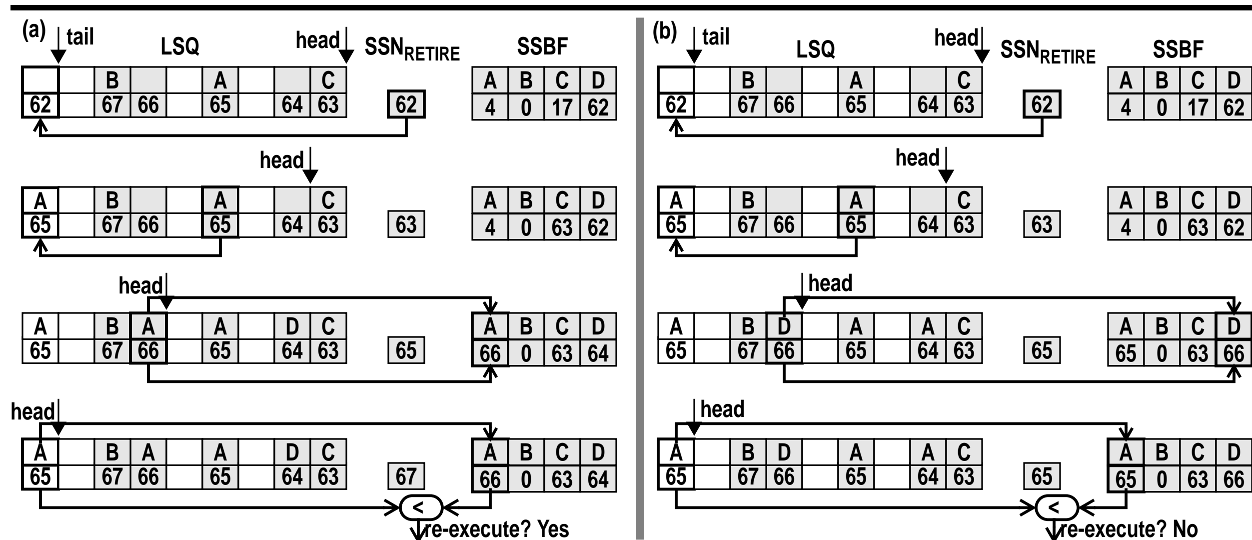

FIGURE 4. Non-Associative LQ + SVW Working Examples. Each example shows the four relevant SVW steps in the life of a single dynamic load, the one on the left-most side of the LSQ.

(towards the LSQ head). Stores are shaded. The second structure is the singleton register **SSN_{RETIRE}**. The final structure is the SSBF which shows the SSNs of the last retired stores to write to each of the four addresses we are tracking.

We begin by dispatching the load of interest, the one at the LSQ tail. At dispatch, we establish the load's vulnerability window by setting its SVW to the SSN of the last retired store, here **62**. The load is vulnerable to all stores younger than **62**, i.e., with SSNs greater than **62**.

In the next snapshot, the LSQ head and tail have advanced—notice the updated state of **SSN_{RETIRE}** and the SSBF—but the relevant event is that the load of interest executes. Notice, the load executes in the presence of older ambiguous stores (**64** and **66**), and so is flagged for subsequent re-execution by the scheduler. Also notice, the load forwards its value from store **65**, which also references address A. Because of this action, the load is no longer vulnerable to any stores younger than **65**, inclusive. We update the load's SVW to reflect this fact.

By the third snapshot, we see that store **66** also 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.

In the final snapshot, the load enters the SVW stage of the re-execution pipeline and accesses the SSBF using its own address, A. As expected, the SSBF re-execution test—**SSBF[load.addr] > load.SVW** which in this instance is **66 > 64**—indicates that the load must re-execute because it collided with store **66**, to which it was vulnerable.

Another working example. The right side of the figure shows a similar sequence of steps, in fact the first two steps are identical. The difference is that in this example, the load collides with store **64** to which it is not vulnerable (this store is older than the store which forwarded the value to the load). This time when the load arrives in the SVW stage and checks the SSBF, it finds the SSN **65** in the entry corresponding to address A. The load skips re-execution because, as its SVW indicates, it is not vulnerable to stores **65** and younger. Without SVW, this load would have re-executed. In fact, it would have re-executed even if neither store **64** or **66** wrote to address A.

Handling SSN wrap-around. The key to handling SSN wrap-around is to realize that SVW is only a re-execu-

tion filter. If the filter test **SSBF[load.addr] > load.SVW** becomes unreliable in the region where **load.SVW** approaches the wrap-around point, the filter can simply be ignored and the load can be "safely" re-executed. In other words, the re-execution test is **(SSBF[load.addr] > load.SVW) || (load.SVW >** $SSN_{MAX} - SVW_{MAX}$**)** where SVW_{MAX} is the maximum size of any load's SVW. The key to effective wrap-around handling is minimizing the region in which the filter is effectively disabled and that means identifying a tight bound for SVW_{MAX}. Fortunately, the load-speculation optimization has a natural bound: the size of the store queue (**SQ.size**); a load cannot be vulnerable to a store that is more than **SQ.size** older than itself. With a bounded SVW_{MAX}, we can set the width of the SSN such that wrap-around is infrequent enough so that the fraction of time the filter is unusable is small. For instance, a 16-bit SSN will wrap around every 64K stores. With a 64-entry SQ, the filter will be unusable for 64 of those 64K stores, or about 0.1% of the time.

Implementation: speculative SSBF updates. The working example in Figure 4 shows load SVW/re-execution and store SVW/retirement as atomic relative to one another, meaning that a store does not update the SSBF until all previous loads have retired. However, forcing this atomicity would have the effect of actually exacerbating the load-to-younger-store serialization that re-execution filtering tries to avoid. In practice, loads and stores are forced through the SVW stage in program order, but stores may update the SSBF speculatively before older loads have finished pre-executing and before older stores have retired. The reason this is allowed is because SSNs increase monotonically and because the interpretation of SSBF entries is conservative on the high side. If a subsequently aborted store (i.e., a wrong path store) erroneously writes a high SSN into the SSBF, no false negatives will result and no re-executions will be missed. The only result may be a superfluous re-execution or two which otherwise could have been avoided. In practice, speculative SSBF updates increase the re-execution rate by 1–2% relative to the base rate (e.g., by 0.1% if the base rate is 10%). This is a small price to pay avoiding elongated serializations.

Implementation: implicit SQ SSNs. Our working example may also give the impression that the SQ must explicitly encode SSNs. This is not the case. An SVW must be explicitly maintained for every load in the LQ because there is no inherent relationship between the vulnerability windows of consecutive loads. However, such a relationship does exist between consecutive stores in the SQ, namely: **SQ[i+1].SSN = SQ[i].SSN + 1**. The SSNs of all stores can therefore be represented using a single global counter, for instance SSN_{RETIRE}. If the SSN of any active store is needed for any reason, e.g., to update the SVW of a load on a forwarding operation as in snapshot 2 of Figure 4, that SSN can be computed as the sum of $SSN_{RETIRE}+1$ and the relative-to-head position of the store in the queue.

3.3. SVW for the Inter-Thread (Shared-Memory) Non-Associative LQ Optimization

The second half of the non-associative LQ optimization enforces inter-thread memory ordering by re-executing loads that were in-flight at the time a shared-memory coherence invalidation took place. Again, this optimization is a little different in that: i) conflicting stores do not come from the same thread as the vulnerable loads and there is no natural sequence numbering relationship between the two, and ii) information about conflicting store addresses is only available at the cache line granularity, not the word granularity.

To filter re-executions for this optimization, we add second Bloom Filter, $SSBF_{SM}$, which tracks coherence invalidations at cache line granularity. Loads which execute in the shadow of an invalidation—these can be identified by remembering the value of LQ tail pointer at the time of an invalidation and re-executing all loads until that pointer

becomes the LQ head—access the $SSBF_{SM}$ and re-execute if **$SSBF_{SM}$[load.addr] > load.SVW**. Of course, we still have to specify how load SVWs are defined for this optimization and what SSNs are written into the $SSBF_{SM}$ by invalidations. These specifications follow naturally when we consider that, from the point of view of re-execution filtering, coherence invalidations act as asynchronous stores 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, **load.SVW = SSN_{RETIRE}**. Similarly, when an invalidation occurs, we pretend that this invalidation is a retired store from within the same thread and perform **$SSBF_{SM}$[invalidation.addr] = SSN_{RENAME} + 1**. Because we treat invalidations as same thread stores and SVWs are bounded by the size of the store queue, wrap-around handling is unchanged.

3.4. SVW for the Scalable SQ Optimization

At base, the scalable SQ optimization uses the same SVW implementation as the one used to filter the re-executions of the intra-thread (load-speculation) non-associative LQ. 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. Really, the big difference between the two optimizations and their respective SVW implementations is that while the non-associative LQ has a natural filter and uses SVW as an enhancer, the scalable SQ optimization has no natural filter and uses SVW as an enabler. Without SVW, a scalable SQ would require the re-execution of all loads.

An SVW filter for the scalable SQ uses the same physical SVW definition, **load.SVW = SSN_{RETIRE}**, and the same filter test, **(SSBF[load.addr] > load.SVW) || (load.SVW > SSN_{MAX} – RSQ.size)**, as its non-associative LQ counterpart. Even wrap-around handling is identical because, like the non-associative LQ, the scope of the scalable SQ optimization and the size of a given load's vulnerability window cannot exceed the size of the SQ itself, specifically the RSQ. It may be stating the obvious, but it is the RSQ which contains all stores, not the FSQ which contains only a subset of the stores, that is used to update the SSBF.

3.5. SVW for the Redundant Load Elimination Optimization

Redundant load elimination is fundamentally different than the first two load optimizations we described and requires a somewhat modified SVW implementation. Under both LQ optimizations and the SQ optimization, a load is only vulnerable to stores that were within the window at the time it was dispatched. In contrast, an eliminated redundant load is vulnerable to all stores starting at the original load, i.e., the older load with which it is redundant. This difference manifests in two ways: it requires a new mechanism for establishing load SVWs and it requires a different policy for handling SSN wrap-around. Another difference between redundant load elimination and the other two optimizations is that eliminated loads have empty LQ entries as these are filled during execution. For re-execution, the base addresses of these loads and their values must be read from the register file. This lengthens the re-execution engine pipeline and increases execution engine queue pressure.

Establishing Load SVWs. Redundant load elimination is coordinated by the register renaming stage. Loads create IT entries that describe the operation they are about to perform—opcode, immediate value and physical register inputs—and the physical register which will hold the result. Future loads search the IT and are recognized as redundant and eliminated if they find tuples with matching operation "signatures". To establish SVWs we must pass SSN information from original load to redundant load via the integration table (IT); these SSNs must be tracked by the reg-

ister renaming stage. Per the invariants that govern in-flight SSNs, we can always compute this value as **SSN_{RETIRE} + SQ.OCCUPANCY**, or we can keep a second counter, **SSN_{RENAME}**.

Non-redundant loads attach **SSN_{RENAME}** to the integration-table 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, **load.SVW = IT-ENTRY.SSN**. The re-execution filter test for an eliminated load is the same as it is for a load under any of the previous optimizations we have seen, the load re-executes if **SSBF[load.addr] > load.SVW**.

Handling SSN Wraparound. SSN wrap-around is easy to handle when the scope of the load/store optimization—and the maximum size of the SVW in terms of number of stores—is bounded by some **SVW_{MAX}**. As we have seen, we disable the re-execution filter when a load's SVW is in the range **SSN_{MAX}–SVW_{MAX}** to 0. Unfortunately, the distance measured in stores between a redundant load and its original counterpart is for all intents and purposes unbounded. Obviously, disabling the filter indefinitely would defeat the purpose of implementing it in the first place. The only thing that can be done is to prevent any redundant load's SVW from including an SSN wrap-around point. We do this by flash clearing the integration table whenever **SSN_{RENAME}** reaches 0. Again, if SSNs are wide enough and wrap-arounds infrequent enough, the number of elimination opportunities lost to this measure will be low.

3.6. Combining SVW Filters for Multiple Optimizations

Composing the re-execution streams of multiple active load/store optimizations is easy: a load is re-executed if it is flagged by any of the optimizations. Although different optimizations require slightly different SVW formulations, the gross structure of SVW is such that the SVW mechanisms of different optimizations can also be easily composed. SVW composition is easy because the semantics of the two SVW components—the per-load SVW definition and the SSBF address-collision detection scheme—are individually composable.

Composing per-load SVW definitions. Under all optimizations that 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, i.e., 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 the optimizations. Mnemonically, **$load.SVW_{OPT1+OPT2}$ = MIN($load.SVW_{OPT1}$, $load.SVW_{OPT2}$)**.

Composing SSBF address-collision tests. While the per-load SVW definition changes from one optimization to the next, the basic structure of the address-collision test itself is constant: **SSBF[load.addr] > load.SVW**. Different optimizations may require Bloom Filters of different granularities, e.g., a cache line granularity Bloom Filter is used to reduce the number of re-executions required to enforce inter-thread memory ordering, or Bloom Filters that track only a subset of the stores, e.g., the SQ optimization uses a second Bloom Filter, $SSBF_{NLSQ}$ which only tracks stores that are excluded from forwarding. However, composing the semantics of these filters is simple. A load is re-executed if it "hits" in any applicable filter.

An SVW implementation for all four optimizations. It is unlikely that a single processor would implement all four optimizations we have described. While the non-associative LQ may be universally applicable, the scalable SQ is only needed on wide superscalar machines—specifically machines that need to execute more than one load per cycle—whereas redundant load elimination is more applicable to narrow ones. Regardless, all four optimizations can

be implemented together along with a single SVW mechanism.

First, a re-execution engine for any machine that implements redundant load elimination must include stages that read the address and value of eliminated loads from the register file. Loads that must re-execute for other reasons may also use these stages, in which case the LQ may be completely eliminated. Composing unfiltered re-execution streams is simple: because of the scalable store queue optimization all loads are universally flagged for potential re-execution and must proceed through SVW to prove otherwise.

Setting the SVW for a given dynamic load is subtle but also straightforward after the first glance. Non-eliminated are easy, their SVW is just **SSN_{RETIRE}**. Eliminated loads are a little trickier. First, observe that eliminated loads are not subject to the scalable SQ or the intra-thread portion of the non-associative LQ optimization because they do not execute, however they are subject to shared-memory invalidations. The SVW of an eliminated load is therefore **MIN(IT-ENTRY.SSN, SSN_{RETIRE})**.

The final step is to compose the Bloom Filters. At base, these three optimizations use two filters: the intra-thread LQ, scalable SQ, and elimination use the basic SSBF, the inter-thread LQ optimization uses $SSBF_{SM}$. A load flagged for re-execution accesses both filters and re-executes on a hit in either, i.e., if **MAX(SSBF[load.addr], $SSBF_{SM}$[load.addr]) > load.SVW**.

3.7. SVW vs. MCB/ALAT

We now revisit our comparison of SVW with MCB/ALAT in the context of the optimizations we have studied.

We have already seen that SVW can filter re-executions for the non-associative LQ. Could MCB/ALAT do the same? The answer is possibly, but probably not well. In order to begin tracking collisions, MCB/ALAT requires the load's address. Since the non-associative LQ optimization is concerned with memory ordering, it is sufficient to track loads from the time they execute which means that an address would be available for MCB/ALAT. Next, as store addresses become available, MCB/ALAT would need to check these address against all tracked load addresses and invalidate addresses, i.e., force eventual re-executions, on matches. In order to not invalidate older loads when they collide with younger store addresses, the MCB/ALAT would need to track load and store sequence numbers and perform associative ordering comparisons. At this point, the MCB/ALAT is incorporating some SVW functionality, i.e., the SSN definition of the window, but doing it in an associative, i.e., less effective way.

As for the scalable SQ and redundant load elimination optimizations, MCB/ALAT cannot help here at all. Because these optimizations have to do with speculative forwarding rather than execution ordering, collision tracking must begin before the address of the load is available.

4. Experimental Evaluation

We use timing simulation to evaluate SVW's impact on three of our load/store optimizations: the intra-thread non-associative LQ, the scalable SQ, and redundant load elimination. We do not evaluate SVW on the inter-thread component of the non-associative LQ because our simulation infrastructure does not execute shared-memory programs.

Benchmarks. Our benchmarks are the SPEC2000 integer suite. We compiled the benchmarks using the Digital OSF C compiler with optimization flags –O3. We run them to completion, on the training input sets using 5% peri-

odic sampling with 5% cache and branch predictor warm-up. Each sample contains 10M instructions.

Performance simulator. Our performance simulator executes the Alpha AXP user-level instruction set using the ISA definition 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 (or three) level on-chip memory system complete with buses and finite outstanding miss handling resources.

Because the non-associative LQ and scalable SQ target wide machines, while redundant load elimination targets narrower machines we use two different processor configurations. The common aspects of the configurations are the memory system, branch predictor, and pipeline structure. The fetch unit includes 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. 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 LQ/SQ processor configuration is an 8-way issue superscalar with a 512-entry reorder buffer, a 128-entry LQ, a 64-entry SQ, 200 issue queue entries, and 448 physical registers. It can issue 5 integer, 2 FP, 2 load, 2 store, and 1 branch per cycle. Our redundant load execution baseline configuration is 4-wide with a 128-entry ROB, 32-entry LQ, 16-entry SQ, 50 issue queue entries, and 160 physical registers. It can issue 3 integer, 1 FP, 1 load, 1 store and 1 branch per cycle. Both configurations use intelligent load speculation managed by a 64-entry store-set table [6]. Both have a single store retirement port; dual ports improve the performance of only one benchmark (*vortex*) by 6% on the 8-wide machine.

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 SQ and LQ and four to redundant load elimination. Where it is used, SVW adds an additional stage. Our baseline SVW configuration is 16-bit SSNs and a 512-entry (1KB total storage) SSBF. SSBF read/write bandwidths match load and store issue bandwidths, i.e., 2/2 or 1/1.

4.1. SVW's Impact on the (Intra-Thread) Non-Associative LQ

We begin by evaluating SVW's impact on the intra-thread non-associative LQ optimization. Figure 5 shows load re-execution rates (percent of retired loads) and speedups (percent IPC improvement) of four configurations. Our baseline configuration here is the 8-way superscalar with a 128-entry associative LQ and the ability to issue a single store per cycle, i.e., the LQ has a single associative port.

The first configuration is NALQ, in which the associative LQ port is replaced by re-execution; this change allows this configuration to execute two stores per cycle rather than one, previously store issue bandwidth was limited by the number of LQ ports. With a good natural filter—only loads that issued in the presence of older ambiguous stores are re-executed—the average "natural" re-execution rate for this optimization is 7.4%. Only three programs re-execute more than 10% of their loads, and only twolf re-executes 20%. These numbers are a little higher than those initially reported [5] because our window is a larger and our machine is wider. With these low re-execution rates the average gain from the additional store port are 0.3%. 9 of the programs suffer slight (less than 1%) slowdowns and *parser* (*par*) shows a 3.5% slowdown stemming from an 8.5% re-execution rate. Again, slowdowns are not directly corre-

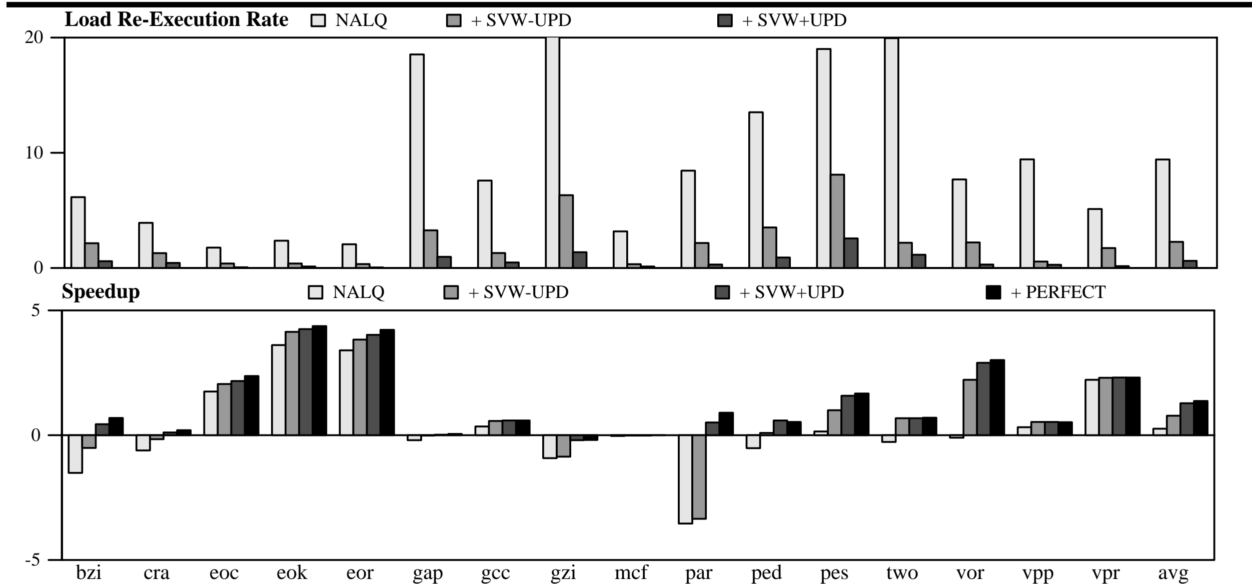

FIGURE 5. Non-associativeLQ re-execution rate (top) and performance (bottom)

lated to re-execution rate because the primary 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 to NALQ. The first SVW configuration, *SVW–UPD*, does not update a load's SVW to the SSN of the store from which the load forwards a value. The second configuration, *SVW+UPD*, does perform this simple optimization. Even without forward updates, SVW reduces the average load re-execution rate from 7.4% to 2.0% with a maximum of 8.1% (*perlbmk.diffmail*, *ped* in the figure). Most of the remaining re-executions can also be filtered using the "update SVW on forward" technique, which reduces re-executions further to 0.6% of all loads, with a maximum of 2.6% (again *perlbmk.diffmail*). This constitutes a 92% reduction in the number of re-executions. With the forwarding optimization—it's so simple that it doesn't make sense not to implement it—NALQ's performance improvement climbs to 1.3% with only one program (*gzip*) showing a slowdown of –0.2%.

NALQ+SVW's speedups are not impressive, but NALQ is not about performance. The point of these experiments—as it is with all the experiments that follow—is not to show how good each baseline technique is, but rather how close to ideal SVW allows that technique to be. The final experiment (*+PERFECT*) implements NALQ with perfect, zero-latency, infinite bandwidth re-execution. The average performance improvement of the ideal NALQ is 1.4%. SVW comes pretty close (1.3%) to achieving it.

4.2. SVW's Impact on the Scalable SQ

Figure 6 shows the results of similar experiments that measure SVW's impact on the scalable SQ optimization. Our baseline configuration here is again 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, load execution takes 4-cycles due to the associative SQ.

The first configuration presented relative to this baseline, SCSQ, is the scalable SQ design we sketched in Section

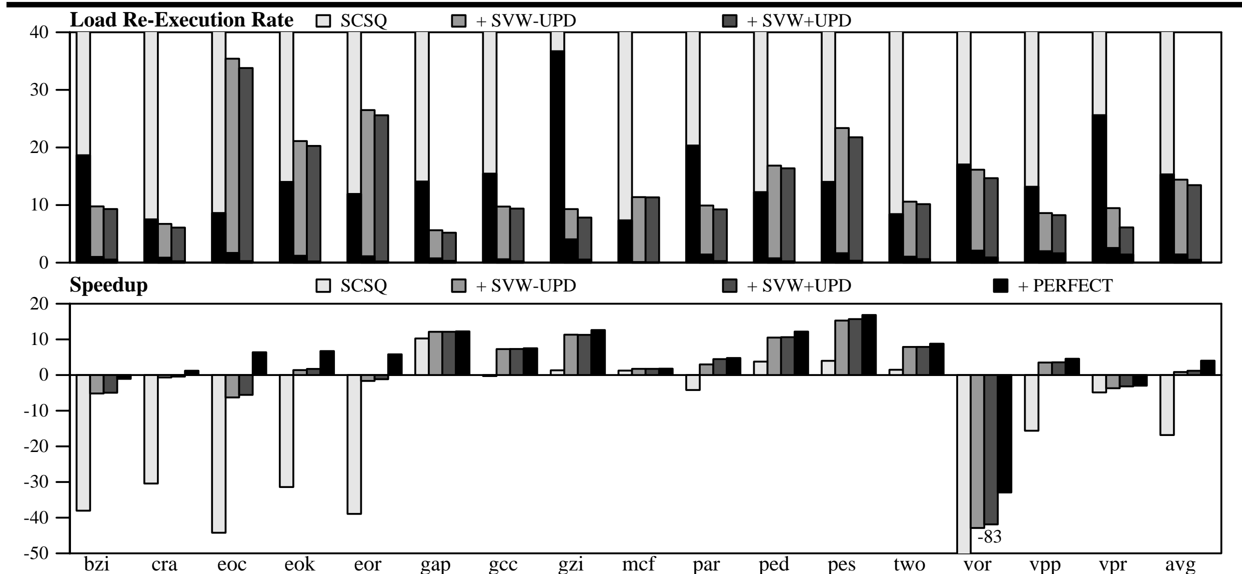

FIGURE 6. Scalable SQ re-execution rate (top) and performance (bottom)

2.2. 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 SQ of this size is required to match the access time of the cache banks. Each data cache bank is fronted by an 8-entry unordered, "best-effort" forwarding buffer which handles most of the easy forwarding scenarios. The load issue bandwidth of this configuration is also two loads per cycle, but only one of these may access the FSQ. Here, loads execute in two cycles.

Recall, the scalable SQ optimization has no natural re-execution filter and must re-execute all loads. In the graph, we break down this re-execution rate to re-executions of loads that access the FSQ (black portion at the bottom of the bar), and re-executions of loads that either use best-effort forwarding or do not forward at all (shaded portion at the top). Because all loads re-execute, this configuration—despite the 2 cycle reduction in load hit latency—achieves an average slowdown of 16%, the maximum slowdown is 83% (*vortex*). Programs that suffer the most are those with large numbers of loads followed closely by stores, but also programs with high baseline IPCs like *bzip2*, *eon*, and *vortex*. With only a single store retirement port, there is not enough bandwidth to retire all stores and re-execute all loads while still sustaining the natural throughput of the program.

In the next two configurations, *SVW–UPD* and *SVW+UPD*, we again add SVW, first without the "update on forward" optimization (*SVW–UPD*) and then with it (*SVW+UPD*). The average re-execution rates for these configurations are 14.5% and 13.4%, respectively, with maximum rates of 33.4% and 32.7% (both *eon.cook*). The "update on forward" optimization does not help much here because it only applies to loads that are steered to the FSQ. It cannot be performed on loads that use best effort forwarding, which doesn't maintain the invariants required to support it. It is not surprising that with an 86.6% reduction in re-execution, the average performance impact of the scalable SQ 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%.

Again, the 1.2% average improvement with SVW is close to the 4% improvement the scalable SQ can achieve even with perfect re-execution (our last configuration, *+PERFECT*). In fact, here we can see that re-executions are

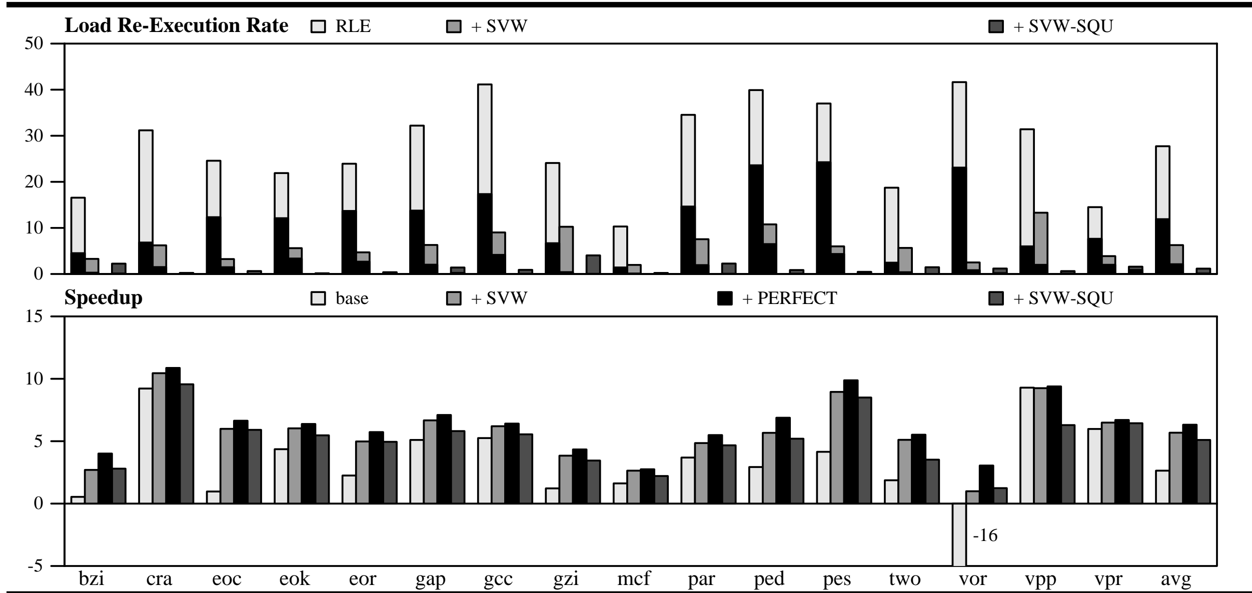

FIGURE 7. Redundant load eliminiation re-execution rate (top) and performance (bottom)

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 the 16-entry FSQ provides. Another way to quantify this result is that SVW brings the scalable SQ to within 86% of its ideal performance.

4.3. SVW's Impact on Redundant Load Elimination

Figure 8 shows a similar set of experiments that measure SVW's impact on redundant load execution. Here our baseline configuration is the 4-wide machine with no elimination. The first configuration relative to this baseline (*RLE*) adds a 512-entry 2-way set-associate load reuse table and a four stage re-execution pipeline. Recall, eliminated loads are not executed and so do not fill their corresponding LQ entries with addresses and values. For re-execution, these values must be read out of the main register file which has a 2-cycle access latency. We add a single dedicated read port to the register file for this purpose. The address is read first and the value is read in time to be compared with the reloaded value. In the baseline technique, all eliminated loads must be re-executed.

Since our configuration eliminates an average of 27.7% of the loads in the program, this is also the re-execution rate. The maximum rate is 41.6% (*vortex*). We break down re-executions according to whether the corresponding 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 average performance improvement corresponding to this elimination rate is 2.6%, with a peak of 9.2% (*crafty* and *vpr.place*). *Vortex* again posts a 16% slowdown (the only program to post any slowdown) and the reason for which is the same. Vortex has a combination of a high baseline 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.

Redundant load elimination only 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 must re-execute. We can change that by adding SVW to the re-execution pipeline, inserting the stage between register file

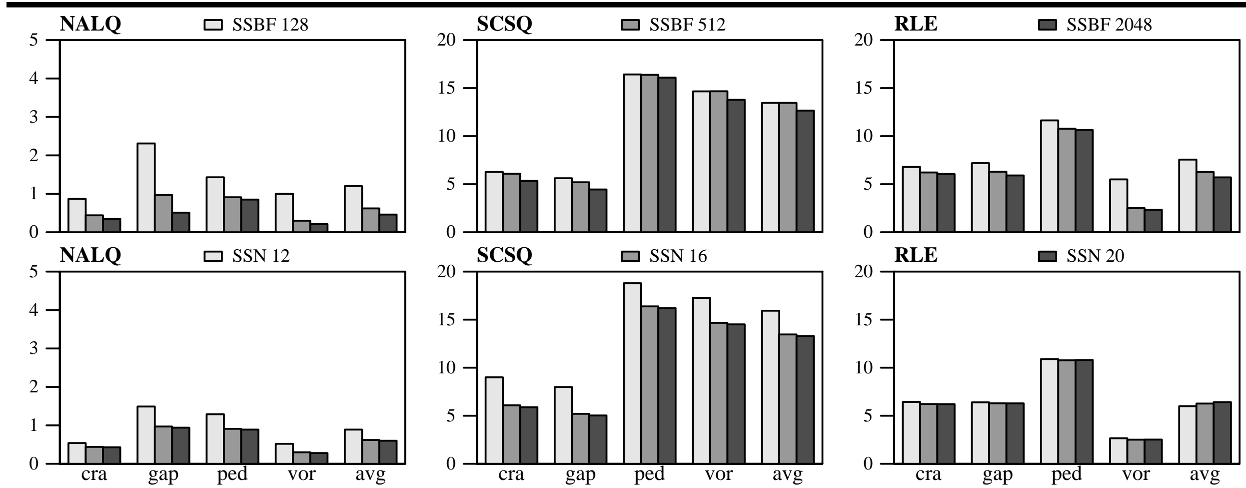

FIGURE 8. SVW re-execution rate sensitivity to SSBF size (top) and SSN width (bottom).

access and data cache access. Because eliminated loads do not execute and do not forward, they are not eligible for the "update on forward" SVW optimization. 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%. Here, SVW is 83% of the way from baseline elimination and elimination with perfect re-execution.

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. SVW is disabled for squash reuse because of a certain corner case—a forwarding store on the squashed path and the absence of that store on the subsequent correct path—which the SSBF cannot capture. Our final experiment, *SVW–SQU*, we simply disable squash reuse in the baseline elimination configuration. Although re-executions drop markedly, from 6.3% to 1.2%, performance drops slightly as well, from 5.7% to 5.1%. Getting rid of the last few re-executions does not justify forfeiting squash reuse.

4.4. SVW Configuration Sensitivity Analysis

The SVW configuration in all experiments thus far uses 16-bit SSNs and a 512-entry (1KB) SSBF. In this section, we measure the effects of varying these parameters.

SSBF size. The number of SSBF entries affects the number of filter false positives. In a larger SSBF, a load is less likely to alias with a recent store to a different address. The top row of graphs in Figure 8 shows re-execution rates for a subset of the benchmarks for each optimization and at three SSBF sizes: 128, our default 512, and 2K entries.

Certainly, none of the three optimizations we discuss requires an SSBF as large as 2K entries. The average SVW size of the three optimizations—i.e., the average distance in stores between the load and the oldest store in its SVW—is between 5 and 30 stores. Clearly, the chances of 2 *different* entries out of 30 aliasing in a 2K buffer are small. They are also small in a 512-entry buffer and a 128-entry buffer. The non-associative LQ optimizations is impacted by the smaller buffer at the same absolute rate as the other two. The impact looks bigger because the number of *a priori* SVW re-executions is smaller. The lesson is simple: a 1KB (512 entries) SSBF suffices, but so does a 256B SSBF.

SSN width. SSN width affects the frequency of SSN wrap-around which determines either the fraction of time in

which the SVW filter is unusable—for the non-associative LQ and scalable SQ—or the frequency with which the reuse table must be flushed—for redundant load elimination. The bottom row of graphs in Figure 8 shows re-execution rates for the same benchmark subset using a 512-entry SSBF but with 12-bit, 16-bit (our default), and 20-bit SSNs, corresponding to wrap-around intervals of 4K, 64K, and 1M stores, respectively.

SSN width, at least on the narrow side, impacts the LQ and SQ optimizations because both of these have SVWs bounded by the size of the SQ and deal with wrap-around by disabling the SSBF in the SQ-sized range leading up to zero. With a 64-entry SQ, our default 16-bit SSNs will wrap-around once every 64K stores; the SSBF will be unusable 64 out of 64K or about 0.1% of the time. At 20-bits, wrap-around occurs every 1M stores and the SSBF is unusable 0.006% of the time which for all practical purposes is indistinguishable from 0.1%. However, at 12-bits wrap-around occurs every 4K stores and the SSBF is unusable about 1.6% of the time. That is significant.

Notice, redundant load elimination which deals with wrap-around by clearing the integration table and preventing SVWs from crossing the wrap-around point seems impervious to SSN size, and even re-executes a little more when SSN increases. What is going on? Well, clearing the IT at more and more frequent intervals has the effect of reducing the number of eliminations. Since it is only eliminated loads which are re-executed, re-executions generally decrease along with them. The effects are very small, though, because in practice a redundant load has to be dynamically close to the original because otherwise physical register reclamation would have scavenged the original value.

5. Related Work

Re-execution and re-execution filtering. Dependence-free checking—via in-order pre-retirement re-execution—is the fundamental mechanism of the DIVA microarchitecture [2]. 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. [8] 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 [9, 21]. Recently, Cain and Lipasti [5] have revived this idea, but used filtering heuristics to reduce the number of re-executions and make the approach competitive with snooping. 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. SVW enhances their technique. A general re-execution filtering mechanism for software optimization is the Memory Conflict Buffer (MCB) [7] or Advanced Load Alias Table (ALAT) [10]. We have already discussed the relationship of these to SVW at some length.

Load Queue and Store Queue Optimizations. 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. [19] reduce SQ access bandwidth by guarding the SQ with a Bloom filter that conservatively encodes the addresses of in-flight stores; only loads whose addresses “hit” in this filter access the SQ. The same authors also show how LQ search bandwidth and LQ size can both be reduced using similar Bloom filter guarding. The design of Park etl. al. [16] achieves the same access bandwidth reductions, but using a store-load dependence predictor [6, 13, 22] 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. 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 SQ holds the most recent stores while a smaller, second-level SQ holds all in-flight stores. A Bloom Filter eliminates most searches to the second-level SQ. The scalable SQ approach we study here—splitting the SQ into a large, non-associative retirement structure and a small, associative (but low bandwidth) forwarding structure—was proposed by Roth [18] and by Baugh and Zilles [3].

It is instructive to compare SVW's use of Bloom Filtering with these previous uses. Previous techniques have used Bloom Filters to guard SQ access. Bloom filters of this kind are managed speculatively and out-of-order meaning that their contents are difficult to maintain precisely and that they are vulnerable to false positives from loads that match younger (i.e., non forwarding) stores. They are also accessed on the load execution 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, i.e., it is not vulnerable to collisions from younger stores. It is not accessed on the load execution critical path.

Redundant Load Elimination Optimizations. The redundant load elimination implementation we use is register integration [17], which captures both load-load redundancy and store-load forwarding. There are other implements, which primarily differ in their use of address matching or memory dependence speculation rather than register dependence information to detect sharing opportunities [11, 14, 15, 20]. SVW can be used to reduce re-executions in the two more general mechanisms [15, 20]. The two speculative memory bypassing implementations [11, 14] operate strictly within the instruction window and can detect false eliminations using intra-thread memory ordering hardware.

6. Conclusions

A high-bandwidth, low-latency load-store unit is a critical component of high ILP processing. Unfortunately, this same unit is also one of the most complex and non-scalable pieces in the execution core. Load/store optimizations simplify some aspect of the core load-store unit in exchange for the in-order pre-retirement re-execution of some or all of the loads in the program. Recent examples of load/store optimizations include a non-associative load queue (LQ), a speculatively accessed speculatively populated store queue (SQ), and the elimination (reuse) of redundant loads. Unfortunately, re-execution itself mitigates the benefits of these optimizations by contending for cache bandwidth with store retirement and by serializing load re-execution with subsequent store retirement. If a sufficient number of loads must be re-executed, the costs of re-execution outweigh the benefits of the optimization it supports.

Store Vulnerability Window (SVW) is a new mechanism that reduces the re-execution requirements of a given load/store optimization significantly, by an average of 85% across three load/store optimizations we have studied. This reduction eliminates cache port contention and eliminates many of the dynamic serialization events that contribute the bulk of re-execution's cost, and allows the optimizations to achieve nearly their full potential. In the case of the scalable SQ, it actually enables the optimization, which would not have been profitable without SVW. SVW is a simple scheme based on monotonic store sequence numbering and a novel application of Bloom Filtering. The cost of an effective SVW implementation is a 1KB buffer and an additional 2B field per LQ entry.

We are currently investigating the impact of SVW on other load optimizations and studying its energy trade-offs.

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

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

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

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

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

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

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

[11] 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.

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

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

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

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

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

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

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

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

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

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

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