Store Vulnerability Window (SVW): A Filter and Potential Replacement for Load Re-Execution

Amir Roth AMIR@CIS.UPENN.EDU
University of Pennsylvania
Department of Computer and Information Science
Philadelphia, PA 19104 USA

Abstract

Load scheduling and execution are performance critical aspects of dynamically-scheduled processing. Several techniques employ speculation on loads with respect to older stores to improve some aspect of load processing. Speculative scheduling and speculative indexed store-load forwarding are two examples.

Speculative actions require verification. One simple mechanism that can verify any load speculation is in-order re-execution prior to commit. The drawback of load re-execution is data cache bandwidth consumption. If a given technique requires a sufficient fraction of the loads to re-execute, the resulting contention can severely compromise the intended benefit.

Store Vulnerability Window (SVW) is an address-based filtering mechanism that significantly reduces the number of loads that must re-execute to verify a given speculative technique. The high-level idea is that a load need not re-execute if the address it reads has not been written to in a long time. SVW realizes this idea using a store sequence numbering scheme and an adaptation of Bloom filtering. An SVW implementation with a 1KB filter can reduce re-executions by a factor of 200 and virtually eliminate the overhead of re-execution based verification. The same SVW implementation can be used as a complete replacement for re-execution with only 3% overhead.

1. Introduction

Load scheduling and execution are two of the most complex yet performance critical functions of a dynamically scheduled processor. For performance, loads should be scheduled to execute as early as possible but not earlier than older stores that write to the same address. Load execution itself should use high-bandwidth, low-latency structures.

These goals can be achieved using *speculation on loads with respect to older stores*. To approximate optimal scheduling, processors use speculation—typically guided by a predictor [6, 27]—to schedule loads intelligently in the presence of older stores from the same thread with unknown addresses. In a shared-memory program, this speculation implicitly extends to logically older stores from other threads whose addresses are unknown outside of their own thread. In conventional designs, the aspect of load execution that constrains latency and bandwidth is store-load forwarding which uses associative search of an age-ordered store queue to locate the youngest older-than-the-load store with a matching address. Recent proposals to reduce store-load forwarding latency and increase its bandwidth use speculation to reduce associative search frequency [2, 21], limit the number of entries that must be searched [2, 21], replace the age-indexed fully-associative queue with an address-indexed set-associative structure [8, 24], or maintain the age-ordered queue but replace associative search with indexed access [23].

Speculative actions require *verification*. Conventional processors verify scheduling speculation using an age ordered load queue that holds addresses of executed loads. When stores execute, they associatively search the load queue for younger loads with matching addresses. A match signals a "premature" load and triggers a recovery action, typically a flush of the load and all younger instructions. In a shared-memory multiprocessor, committed stores from other threads also trigger load queue searches. Associative load queue search has limitations. Associative search is slow. Address-based comparisons are also vulnerable to false positives caused by matches with silent stores [12]; these can be eliminated in the single-thread setting by augmenting the load queue with load data values, but cannot be eliminated in the shared-memory setting because data values of stores from other threads are not available for matching. Finally, associative load queue search is triggered by some store event (either execution or commit) which means it cannot verify speculation that takes place "beyond the window" [20]. In fact, if load queue search is triggered by store execution than it cannot verify techniques that speculate loads relative to older stores that have already executed [2, 21, 23].

An alternative form of verification that overcomes these limitations is *in-order load re-execution prior to commit* [1, 5, 9]. Prior to commit, loads that execute under some form of speculation re-execute by accessing the data cache again. Mis-speculation is detected—and recovery is initiated—when the re-executed value differs from the value obtained during original execution. Re-execution does not require associative search, is not confused by silent stores, and can verify any form of load speculation. Its primary disadvantages are that it consumes cache bandwidth and that it introduces a new critical loop [4]: a store may not commit until all previous loads have re-executed successfully. These limitations are not significant for techniques that apply speculation to—and as a result must re-execute—only a small subset of the loads. Speculative scheduling is one such technique; typically 10–20% of dynamic loads are scheduled speculatively. However, they can be quite significant for techniques that essentially make all loads speculative in one form or another [23]. For these, the performance cost of re-execution may seriously compromise or completely negate the primary gain, and even produce drastic slowdowns.

Store Vulnerability Window (SVW) is an address-based filtering mechanism that significantly reduces the number of loads that must re-execute to verify a given form of load speculation [22]. The high level idea behind SVW is that a given load need not re-execute if the address it reads has not been written to in a long time. The slightly more precise idea is that a given technique makes a given load speculative with respect to—in other words, vulnerable to—some dynamic range of older stores. The load should not have to re-execute if none of the stores from its store vulnerability window (SVW) writes to its address. The SVW implementation assigns dynamic stores monotonically increasing sequence numbers (SSNs). When a load executes, it is assigned the sequence number of the youngest older store to which it is not vulnerable (SSN_{nvul}); under most forms of speculation, this is the SSN of the youngest store to commit to the data cache. A

small address-indexed table called the Store Sequence Bloom Filter (SSBF) conservatively tracks the SSN of the youngest committed store to write to a given address. Before a load re-executes, it uses its own address to read the SSBF. If the entry it reads is less than or equal to its own vulnerability SSN (SSN_{nvul}), the load safely skips re-execution knowing that none of the older stores to which it is vulnerable wrote to the same address.

By reducing data cache bandwidth contention and removing dynamic load re-execution/store-commit serializations, SVW reduces the overhead of load re-execution. This is especially important for techniques that perform speculation of one kind or another on all loads and thus would require all loads to re-execute to guarantee correctness. Timing simulations on the SPEC2000 and MediaBench programs show that an SVW scheme with 16-bit SSNs and a 128-entry, 4-way set-associative Bloom Filter reduces re-executions by a factor of roughly 200 and virtually eliminates the overhead of re-execution based verification, even for techniques with *a priori* re-execution rates of 100%. The same SVW implementation can also completely replace re-execution—with a hit in the Bloom filter treated like a mis-speculation and triggering recovery—with only 3% overhead. The SVW implementation is simple, supports multiple forms of speculation simultaneously, and is shared-memory multiprocessor safe.

The next section presents background material on load re-execution and two load speculation techniques that exploit it: speculative scheduling and speculative indexed forwarding. Section 3 presents the Store Vulnerability Window (SVW) scheme and a simple representative implementation. Section 4 evaluates SVW in the context of speculative scheduling and forwarding. Section 5 concludes.

2. Background

SVW optimizes load re-execution, which itself is a verification mechanism for some form of load speculation. This section presents a simple load re-execution architecture and two load speculation techniques that use it for verification.

2.1. In-Order Load Re-Execution

The commit pipeline of a superscalar processor processes completed instructions in program order. The primary functions of the pipeline are to free resources for committing instructions, handle exceptions, and commit stores to the data cache. A basic commit pipeline consists of two nominal stages. The nominal DCACHE stage writes stores from the head of the store queue to the data cache or store buffer; this stage can occupy multiple physical stages as data cache access is typically a multi-cycle operation. Processors typically support only one data cache write per cycle because stores typically account for fewer than 20% of dynamic instructions[1]. The DCACHE stage is effectively idle for non-stores. The COMMIT stage flushes the pipeline on exceptions and retires non-excepting instructions, freeing their resources by incrementing the head pointers of the re-order buffer and the load and store queues as necessary.

To support load re-execution, the stages of a basic commit pipeline are augmented to perform explicit actions for re-executing loads. These additional actions are shown in bold in the pipeline action diagram in Table 1. The DCACHE stage uses addresses of re-executing loads—which are retrieved from the head of the load queue—to read the data cache and obtain new values that are guaranteed to be correct. The COMMIT stage compares the new values for re-executed loads to their original values and triggers a pipeline flush on a mismatch. Load re-execution logically shares the data cache port with store commit; stores and loads access this port in order relative to each other and so no explicit arbitration is necessary. Physically, load re-execution can share a data cache port with out-of-order load execution. For simplicity, Figure 1—which shows the structures and added or modified paths necessary to support load re-execution—shows re-execution as using a dedicated data cache port.

One of the stated limitations of Cain and Lipasti's load re-execution proposal [5] is that re-execution

1. Some processors use a wide data cache write port that can accommodate two stores to adjacent addresses [11].

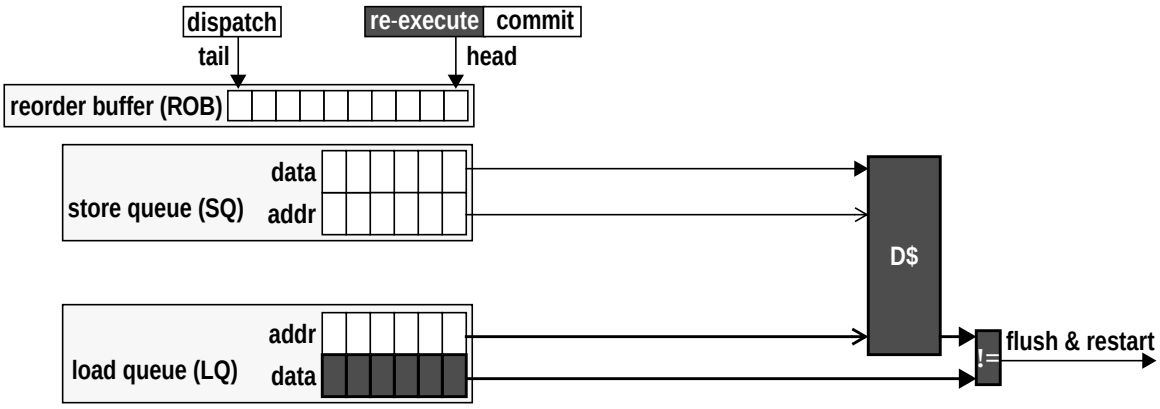

Figure 1. Commit pipeline with load re-execution (added/modified structures/paths in bold). Load re-execution adds a data field to the load queue. For simplicity, the figure shows re-execution as using a dedicated data cache port. In actuality, re-execution may share a port with store commit.

	DCACHE+	COMMIT
Store	D$[st.addr]=st.data	**SPCT[st.addr]=st.PC** commit
Load	**ld.reexec ?** **ld.ndata=D$[ld.addr]**	**(ld.reexec & ld.ndata != ld.data)** **? flush,** **train ld.PC/SPCT[ld.addr]** commit

Table 1. Action diagram for commit pipeline with load re-execution. Load re-execution adds load actions to the commit pipeline. Loads re-execute during the DCACHE stages and compare their re-executed values with their original executed values at COMMIT.

does not track the identities of stores that trigger flushes and can only train store-blind dependence predictors [11], rather than more refined store-load pair dependence predictors like store-sets [6]. This limitation can be overcome using the *store PC table (SPCT)*: a small 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. Figure 1 does not explicitly show the SPCT, which is only peripheral to re-execution.

Performance impact. The cost of load re-execution is the introduction of stalls (*i.e.*, bubbles) in the commit pipeline. If these stalls delay commit to the point of exerting back-pressure on the ROB, load and store queues, or register file, the result will be stalled dispatch and execution and reduced performance. Commit stalls manifest as a result of actual re-execution. As a result, the magnitude of their performance effect is proportional to the re-execution rate, the fraction of dynamic loads that re-execute.

The primary cause for commit pipeline stalls is simple contention for data cache bandwidth. A secondary cause is the fact that stores cannot commit until all older loads have successfully re-executed. In any commit pipeline with re-execution, a store cannot commit either in the same cycle as an older re-executing load or in the following cycle. If data cache access takes multiple cycles and if cache writes cannot be "canceled" up until the final cycle, stores cannot commit for a number of cycles equal to full data cache access latency following a load re-execution, effectively forming a critical loop [4]. Serialization stalls cannot be mitigated by a store buffer, which exposes stores externally and from which stores cannot be aborted. There are no stalls associated with load re-execution following a store or with consecutive dependent load re-executions. The former case is handled by buffering. The latter is handled by "dependence-free checking" [1] with the consumer load re-executing using the producing load's original output as its own input.

2.2. Speculative Load Scheduling

Dynamically-scheduled processors speculatively schedule loads out-of-order with respect to older stores. In single-threaded execution, a load is speculative with respect to older stores from its own thread whose addresses are not known at the time it executes. In multi-threaded shared-memory execution, a load is additionally speculative with respect to all stores from other threads that will commit before it does. With respect to stores from other threads, speculative load scheduling is a virtually unavoidable consequence of out-of-order execution. With respect to stores from the same thread, out-of-order execution does not necessarily imply speculative load scheduling; some early out-of-order processors like the Pentium Pro used a limited, non-speculative load scheduling policy in which loads executed in-order with respect to older store address calculations. However, speculative load scheduling is desirable for performance reasons. Most loads that execute in the presence of older stores with unknown addresses do not conflict with (*i.e.*, read addresses written by) those stores. Loads that do conflict with older un-executed stores do so repeatedly. To prevent pre-mature scheduling of these loads, they are delayed until all prior stores execute [11] or speculatively synchronized with the dynamic store with which they are most likely to conflict using some store-load dependence prediction mechanism, *e.g.*, Store Sets [6].

Verification using associative load queue search. Conventional processors detect load scheduling mis-speculation—pre-mature load scheduling—using a load queue that tracks the addresses of in-flight loads. Loads write their addresses into the load queue when they execute. When stores execute, they associatively search the load queue of their own thread for younger loads to the same address. A match signals a scheduling mis-speculation and triggers recovery. When stores commit, they indirectly—via the cache coherence protocol—trigger similar searches of the load queues of the other threads in the program. If the load queue tracks load output values in addition to addresses, spurious mis-speculations due to conflicting silent stores [12] from the same thread may be avoided. Spurious mis-speculations due to silent stores from other threads cannot be avoided because cache coherence protocols broadcast only the addresses—and the cache block addresses at that—of committing stores, not their data values.

Although some architectures require load queue search on load execution to ensure that loads to the same address execute in-order, load queue search is generally not on the load execution critical path. Nevertheless, load queue search is slow and power-hungry, searches are frequent, and spurious mis-speculations are costly.

Verification using load re-execution. To reduce spurious flushing, Gharachorloo et al. proposed using in-order load re-execution to verify speculative load scheduling in a shared memory environment [9]. They dismissed this approach because the bandwidth consumed by re-executing all loads outweighed the gains of reduced mis-speculation recovery. Cain and Lipasti made re-execution competitive by observing that only loads that are actually speculative must be re-executed and introducing two heuristics that safely filter non-speculative loads [5]. The scheduler can easily recognize and tag loads that are non-speculative with respect to older stores from the same thread. Loads that are non-speculative with respect to stores from other threads are those whose stay in the out-of-order window does not collide with a cache line invalidation. They show that only 2–15% of the loads are speculative with respect to older stores from the same thread and that an additional 20–40% are speculative with respect to stores from other threads [5].

2.3. Speculative Indexed Store-Load Forwarding

Although load scheduling mis-speculation detection is not on the load execution critical path, store-load forwarding—value communication from in-flight stores to younger loads—is. Conventional processors implement store-load forwarding using an age-ordered store queue. When stores execute, they write their addresses and data values into the store queue at a position corresponding to their age. When loads execute, in parallel with accessing the data cache, they associatively search the store queue for older stores with matching addresses. Loads obtain their values from the store queue entry of the youngest older store with a matching address (if any) or the data cache (if no older store has a matching address).

Associative search is power hungry and slow, and its power consumption and latency grow quickly with the number of entries searched. To allow store queues and instruction windows to scale to larger sizes and

higher bandwidths while keeping store-load forwarding latency low, researchers have proposed several novel store queue designs and access methods. Of particular interest here are proposed methods that use speculation to reduce store queue search frequency or the number of entries searched [2, 21] or to eliminate associative search altogether by replacing it with set-associative address-indexed [25, 24, 8] or direct mapped access [23]

This paper describes SVW filtered load re-execution in the context of (*i.e.*, as the verification component for) one of these designs: speculative indexed store-load forwarding [23]. Speculative indexed forwarding uses an age-ordered store queue. However, loads don't associatively search this store queue; instead they index it directly. A store-load dependence predictor like Store Sets [6] predicts for each load the store queue position of the dynamic store from which that load is most likely to forward. The load waits for that store to execute and then in parallel with data cache access indexes the store queue directly at this position. If the load's address matches that of the store, the load receives the store's value. Otherwise, it receives the value from the cache.

Speculative indexed forwarding can be thought of as an extension to intelligent load scheduling based on store-load synchronization. For scheduling, a store-load dependence predictor learns the identities of stores that forward values to loads *and* that execute out-of-order with respect to those loads. Loads wait for stores on which they potentially depend to execute, but otherwise access the store queue associatively as usual checking their address against those of all older in-flight stores. For forwarding, the predictor learns the identities of *all* forwarding stores—even those that execute in order with respect to the corresponding loads—and loads access the store queue directly, checking their address only against that of the predicted store. Speculative forwarding is effective because in most programs: i) 80% of the dynamic loads do not forward from older in-flight stores and these are guaranteed to execute correctly, ii) those loads that do forward do so in highly stable and predictable patterns [16].

Verification using associative load queue search. By virtue of the fact that they don't check all older stores, loads in speculative forwarding can be speculative with respect to some older stores that have already executed. To verify speculative forwarding using associative load queue search, load queue search is moved from store execution to store commit [18].

Verification using load re-execution. Under speculative scheduling, loads are speculative only with respect to older stores from the same thread whose addresses are not known at the time the load executes. Under speculative forwarding, loads are speculative with respect to *all* older stores because the forwarding predictor may fail to predict the store queue position of the correct forwarding store.

3. The SVW Re-Execution Filter

Store vulnerability window (SVW) is an address-based filter that significantly reduces the number of loads that must re-execute to verify a particular form of speculation. By reducing commit pipeline stalls due to data cache bandwidth contention and load re-execution/store-commit serialization, SVW virtually eliminates the overhead of re-execution, exposing the raw benefit of the underlying speculation. SVW is especially important for techniques like speculative forwarding which perform some form of speculation on all loads and which without SVW would not provide enough benefit to cover their own re-execution cost.

SVW is a general mechanism that can be tailored to verify many forms of load speculation. This section first presents SVW in the context of speculative scheduling and forwarding, both of which use the same SVW implementation. Section 3.4 presents SVW implementations for other forms of load speculation.

The performance cost of SVW is an additional stage in the commit pipeline which can exert a slight additional pressure on the out-of-order structures. The structural costs are the additions of: i) a short (*e.g.*, 16 bit) field to each load queue entry, and ii) a small (*e.g.*, 1KB) table, the store sequence Bloom Filter (SSBF). Figure 2 shows these added structures and paths.

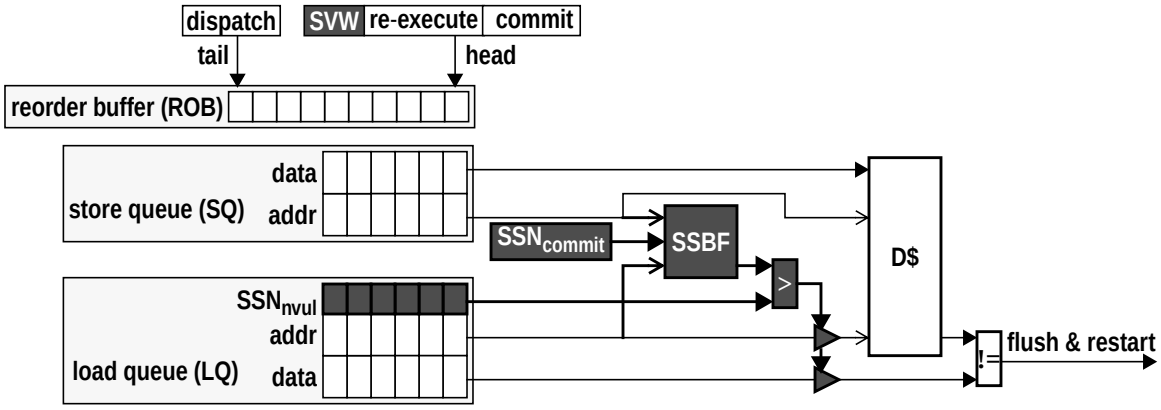

Figure 2. Commit pipeline with load re-execution and SVW filtering.
The added structures (shaded) are the global SSN_{commit} counter, the Store Sequence Bloom Filter (SSBF), and the additional load queue field SSN_{nvul}.

3.1. SVW for Speculative Scheduling and Speculative Indexed Forwarding

SVW exploits the high-level observation that a speculative load—*e.g.*, a load that issues in the presence of older stores with unknown addresses under speculative scheduling, or any load under speculative forwarding—should not have to re-execute if it reads an address that hasn't been written in a long time. The slightly more specific observation is that a particular technique makes a given load speculative with respect to some dynamic range of older stores; a load should not have to re-execute if none of the stores within this range wrote to the address it reads. To implement this general strategy, SVW uses: i) a scheme for naming dynamic store ranges, ii) a system for defining for each dynamic load the range of stores with respect to which it is speculative, and iii) a method for conservatively testing for address collisions within ranges.

SVW assigns each dynamic store a monotonically increasing sequence number, the *store sequence number (SSN)*. For now, assume that SSNs have infinite width; Section 3.3 discusses handling of SSN wrap-around. SSNs strictly require only monotonicity, but it is also convenient if they are dense. Density not only lengthens the wrap around period, it also allows SSNs to be largely implicit. The processor can explicitly represent a single global SSN, the SSN of the last committed store, $\mathbf{SSN_{commit}}$. The SSN of any in-flight store (**st.SSN**) can be computed using this global value and the store's relative-to-head position in the SQ. For convenience, the discussion often also refers to $\mathbf{SSN_{rename}}$, the SSN of the youngest store in the window. This is just $\mathbf{SSN_{commit} + SQ.OCCUPANCY}$.

SVW uses SSNs to name the dynamic range of stores to which each dynamic load is made vulnerable by a given form of speculation; this range of stores as a load's *store vulnerability window (SVW)*. Under all forms of speculation we have studied, a load is only vulnerable to a range of stores that is immediately older than itself; this allows the SVW of a given dynamic load to be defined as the SSN of a single older store. It is convenient to define a load's SVW as the SSN of the youngest older store to which the load is *not* vulnerable, this SSN is recorded in an additional field in the load queue, $\mathbf{ld.SSN_{nvul}}$. The operational definition of $\mathbf{ld.SSN_{nvul}}$ for both speculative scheduling and speculative forwarding is intuitive. Under both techniques, the range of stores with respect to which a load is speculative is the set of older stores that are in-flight when the load executes. If the load forwards from an older store, then it is vulnerable only to stores younger than this forwarding store, *i.e.*, $\mathbf{ld.SSN_{nvul} = forwarding\text{-}st.SSN}$. If the load does not forward, then it is vulnerable to all older in-flight stores, *i.e.*, $\mathbf{ld.SSN_{nvul} = SSN_{commit}}$.

Defining an SVW for each dynamic load is only half the equation. The other half is a small, address-indexed table in which each entry holds the SSN of the last committed store to write to any partially matching address. This is the *store sequence Bloom filter (SSBF)*, where the term Bloom filter is used to mean a filter that can only produce false positives [3]. SSBF access occupies a new stage in the commit pipeline (SVW); this stage immediately precedes data cache access (DCACHE). In this SVW stage, stores write their

	SVW	DCACHE+	COMMIT
Store	**SSBF[st.addr] = st.SSN**	D$[st.addr]=st.data	**SSN_{commit}++** SPCT[st.addr]=st.PC commit
Load	**ld.reexec &=** **SSBF[ld.addr] > ld.SSN_{nvul}**	ld.reexec ? ld.ndata = D$[ld.addr]	(ld.reexec & ld.ndata != ld.data) ? flush, train ld.PC/SPCT[ld.addr] commit

Table 2. Action diagram for commit pipeline with load re-execution and SVW filtering.
SVW prepends a single stage (SVW) to the commit pipeline. In this stage stores write their SSNs to the SSBF entry corresponding to their address. Loads read the SSBF entry corresponding to their adds and check it against their SSN_{nvul}. If the SSBF entry is greater, the load may collide with a store to which it is vulnerable and must re-execute.

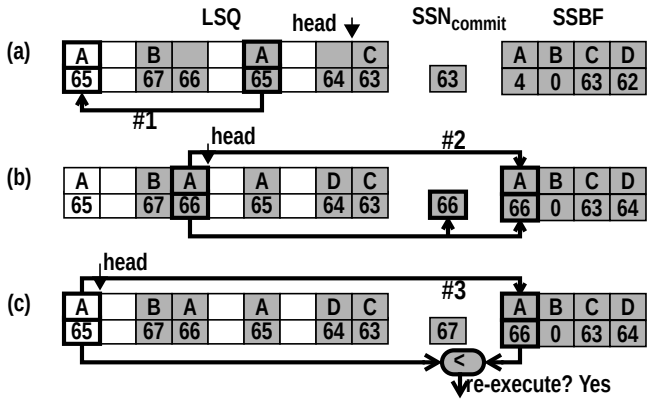

Figure 3. Working example of speculative load scheduling with SVW.
Each snapshot shows the load/store queue (LSQ) with stores shaded, the global counter SSN_{commit}, and the SSBF. Addresses are letters and SSNs are numbers. In this example, the load on the left of the LSQ executes before store 66. It forwards from store 65 and so is only vulnerable to stores younger than 65, *i.e.*, 66 and 67. The load rightly re-executes because it collides with store 66.

SSN into the SSBF entry corresponding to their address (mnemonically, **SSBF[st.addr] = st.SSN**). Speculative loads—*i.e.*, loads marked for re-execution—use their addresses to read the SSBF and evaluate a re-execution filter test by comparing the read entry to their own **ld.SSN_{nvul}**. Again, for speculative scheduling and forwarding the SVW filter test is intuitive. If **SSBF[ld.addr] > ld.SSN_{nvul}**, then the load may conflict with one of the stores to which it is vulnerable and must re-execute to detect possible mis-speculation. If **SSBF[ld.addr] ≤ ld.SSN_{nvul}**, then the load unambiguously does not conflict with any store in its vulnerability range and may skip data cache access. Table 2 shows the actions of the SVW stage in the commit pipeline.

Working example. Figure 3 shows the three SVW events in the life of one dynamic load. Each snapshot shows three structures. The load/store queue (LSQ) is on the left (in the example, the load and store queues are shown as a single interleaved queue for clarity, the example still works if they are separate). Stores are shaded. Older instructions are to the right; in each snapshot the LSQ head pointer advances to the left as older instructions commit. Values do not contribute to the example and so the figure shows only addresses (letters A, B, C, D) and SSNs (numbers). For stores (shaded entries), the SSN is the store's SSN; for loads, the SSN is the load's **SSN_{nvul}**. The second structure is the global counter **SSN_{commit}**. Finally, the SSBF shows the SSNs of the last retired stores to write to each of the four addresses. Notice, that the SSBF entry for the address C is 63, the SSN of the last committed store (which happens to write to C). The load of interest is the

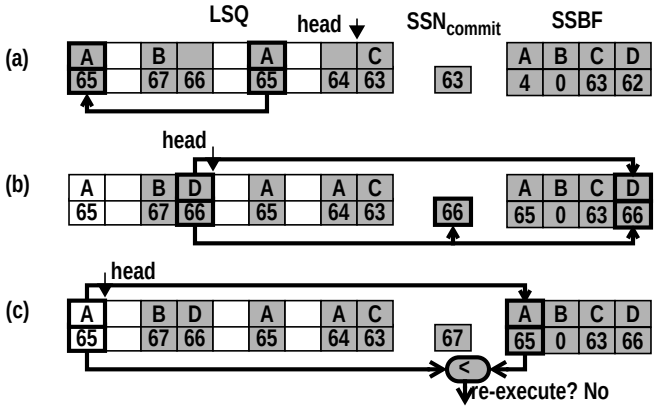

Figure 4. Second working example of speculative load scheduling with SVW.
In this example, store 66 writes to address D instead of address A and the load does not re-execute because it does not collide with any store younger than 65. Without SVW, the load would re-execute.

one on the left end of the LSQ, the one to address A. In each snapshot, the actively participating entries and structures are shown in bold outline.

In the first snapshot (Figure 3a), the load executes in the presence of older stores with unknown addresses 64 and 66, and is marked as speculative by the scheduler. The load forwards from store 65, which means it is not vulnerable to stores 65 and older. The load's $\mathbf{SSN_{nvul}}$ is set to 65 to reflect this (action #1).

In the second snapshot (figure 3b), store 66 writes to A, meaning that the load issued pre-maturely and must re-execute to detect this scheduling mis-speculation. When store 66 commits, it writes its SSN (66) into the SSBF entry for address A (action #2).

In the final snapshot (figure 3c), in the SVW stage of the commit 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 collides with store 66, to which it is vulnerable (action #3).

A Second Example. Figure 4 shows a alternative execution. In this example, store 66 (to which is the load is vulnerable) writes to address D and no longer collides with the load. Store 64 writes to address A and collides with the load, but the load is not vulnerable to this store as it is older than the store that forwards the value to the load. This time, when the load checks the SSBF, it finds the 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 would re-execute.

3.2. Multiprocessor Safety or SVW for Speculative Shared Memory Scheduling

To be practically useful, the SVW mechanism must be multiprocessor safe. Operationally, this means that for any load X, if a store on another thread that writes to X's address commits between the time that X executes and commits, then the SVW filter test should force load X to re-execute.

As described earlier, from the verification point of view, multiprocessor safety can be thought of as a variation of speculative scheduling within a single-thread. As in single-threaded speculative scheduling, the load is vulnerable from the time it executes to the time it commits. The differences in the shared-memory setting are: i) addresses of stores from other threads may only be available at the cache line rather than the word (or byte) granularity, ii) SSNs of stores from other threads may have no natural relationship to the SSNs of the current thread and, as a result, to the vulnerability ranges of loads in the current thread.

SVW updates the SSBF on stores from other threads, *i.e.*, coherence invalidations. To account for the fact that invalidations occur at a cache line granularity, SVW updates multiple SSBF entries per invalidation. The SSBF is divided into banks that equals the number of words in a cache line. For same-thread store commits, only one bank is write-enabled. For other-thread store commits (*i.e.*, invalidations) all banks are enabled. To account for the fact that SSNs from different threads may be unrelated, SVW pretends that a

store from another thread is an "asynchronous store" from the current thread and updates the SSBF with an SSN that is meaningful to the current thread, specifically $\mathbf{SSN_{commit}+1}$. To see that this choice of SSN is sufficient to guarantee safety, consider that a load can have an $\mathbf{SSN_{nvul}}$ of $\mathbf{SSN_{commit+1}}$ or greater under two conditions: i) either the load executes after the store from its own thread corresponding to $\mathbf{SSN_{commit+1}}$ commits in which case it also executes after the invalidation—which occurs somewhere between the commit of store $\mathbf{SSN_{commit}}$ and that of $\mathbf{SSN_{commit+1}}$—takes place, or ii) the load forwards from a store that itself is logically younger than the invalidating store.

To account for coherence protocols with non-silent cache line evictions (*e.g.*, token coherence [14]) in which a processor will not receive an invalidation message if the corresponding cache line is not resident, SVW must performs the above SSBF on any cache line eviction, even a local replacement.

3.3. Implementation Notes

This section addresses SVW implementation issues.

SSN wrap-around. In a real implementation, SSNs have finite width. Successful handling of SSN wrap-around means avoiding comparing SSBF entries (*i.e.*, retired store SSNs) that are slightly larger than zero with load SSN_{nvul}'s that are slightly less than zero. A naive interpretation of the re-execution filter test in this case would suggest that the load is safe because its $\mathbf{SSN_{vul}}$ is much larger (*i.e.*, younger) than the SSBF entry. However, in reality the SSBF entry may be slightly younger.

The following policy avoids such ambiguous comparisons. When $\mathbf{SSN_{rename}}$ wraps around to zero, the processor: i) drains the pipeline by waiting for all in-flight instructions to commit, ii) flash clears the SSBF, and iii) resumes 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 SSN_{nvul} 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 commits. As a result their $\mathbf{SSN_{nvul}}$ is zero. SVW's mechanism for handling SSN wrap-around is similar to the mechanism used by several processors to implement store barriers [11]. This similarity is not a coincidence. Store barriers prevent the interference of pre-barrier stores with post-barrier loads. The SVW barrier prevents pre-wraparound stores from interfering with post-wraparound loads.

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. 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 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 strict queue in the sense that it is tolerant of overflow (*i.e.*, it implicitly represents both in-flight and committed stores). This makes it incapable of maintaining temporal distance guarantees.

Speculative SSBF updates. The discussion so far treats 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 committed. However, physically forcing this atomicity actually exacerbates the load-to-younger-store serialization that re-execution filtering tries to avoid.

There are two possible approaches. The first allows stores to update the SSBF speculatively. This solution works for techniques whose SVW filter test is the inequality $\mathbf{SSBF[ld.addr] > ld.SSN_{nvul}}$; these include speculative forwarding and both single-threaded and multi-threaded speculative scheduling. Because SSNs increase monotonically and because high SSBF entries are interpreted conservatively, a wrong path store writing a high SSN into the SSBF only *potentially* results in a few superfluous re-executions, no necessary re-executions are 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.

The second approach is required for techniques with different SVW filter tests—Section 3.4 describes one such technique, speculative memory bypassing—and uses a short queue to buffer speculative updates to the SSBF. Empirically, on a 4-way issue machine with a 3-cycle data cache, a 4-entry queue induces less

than a 0.1% slowdown over an infinite sized queue.

Tagged, set-associative SSBF organizations. To reduce both SSBF capacity and the re-execution rate, the SSBF can be tagged and made set-associative. In an N-way set-associative SSBF, each set is managed like a FIFO, containing the SSNs of the last N stores to write to any address that maps to the set. The SVW filter test is modified as follows. The load compares its $\mathbf{SSN_{nvul}}$ with that of the youngest entry with a matching tag. If no tags match, the load compares its $\mathbf{SSN_{nvul}}$ with the SSN of the *oldest* entry in the set with the knowledge that the SSN of any store to a matching address must be smaller than any of the SSNs in the set. Note, making the SSBF set-associative reduces conflicts in much the same way that making a cache set-associative does. However, making the SSBF set-associative *and FIFO* results in even more powerful "active" conflict avoidance in the way that has no analogy for caches.

For tagging, the SSBF can use either physical or virtual addresses. Physically-tagged SSBFs are trivially multiprocessor safe. Virtually-tagged SSBFs can be made safe by: i) indexing the SSBF using the low-order untranslated address bits, or ii) creating fake entries in multiple sets on cache-line evictions.

SVW in a multithreaded processor. The fact that SSNs are only required to be monotonic—they don't have to be dense—enables a simple implementation of SVW on a multithreaded processor. All active threads share a single global $\mathbf{SSN_{commit}}$ counter and a single SSBF. The presence of other threads makes SSN appear sparse from the point of view of any single thread but preserves correct execution.

3.4. SVW for Other Load Speculation Techniques

SVW is a general mechanism that can be tailored to verify multiple forms of load speculation. The components of the SVW mechanism that are constant across speculations are the SSN-based naming scheme for dynamic stores and the contents of the SSBF, which logically track the SSNs of the most recent committed stores. Different speculative techniques make loads vulnerable to different ranges of stores and may use different definitions for $\mathbf{SSN_{nvul}}$. Different forms of speculation may also require different SVW filter tests. This section describes the SVW implementations of two additional speculative techniques, redundant load elimination and speculative memory bypassing. It also describes how a single SVW implementation can support multiple simultaneously active forms of speculation. The fact that the SSN numbering and the contents of the SSBF are constant makes this possible.

Speculative redundant load elimination. A given load X is redundant with an older load Y if X and Y read the same address and no store to that address interposes between them. Redundant loads are an artifact of static compilation, notably the limitations of static alias analysis, and the requirement that static optimizations be valid along all possible static paths. Speculative redundant load elimination uses a modification to register renaming to eliminate redundant loads at runtime, simultaneously removing those loads from out-of-order execution engine and the program's dataflow graph by pointing consumers of load X to the output of load Y. Redundant load elimination can be applied "outside-the-window", *i.e.*, after the original load has committed.

There are two kinds of load elimination implementations. Those that identify redundant loads using load-load dependence prediction [17] perform two kinds of speculation: i) that the redundant load reads the same address as the initial load, ii) and that no store to the same address interposes. Implementations that identify redundant loads using register dependence tracking [19] only perform the second kind of speculation. Re-execution is needed to verify this speculation—"outside-the-window" load elimination cannot be verified by associative LQ search even if that search is delayed until store commit because a conflicting store may commit before the load is eliminated—and a proper implementation of SVW can effectively filter these re-executions.

Under speculative scheduling and forwarding, a load is speculative with respect to older stores that are in-flight at the time it executes. In contrast, an eliminated redundant load is speculative with respect to all stores starting at the older load with which it is redundant. This difference requires a new definition of the SVW for redundant loads. Specifically, $\mathbf{eliminated\text{-}ld.SSN_{nvul} = original\text{-}ld.SSN_{rename}}$, where the value of $\mathbf{SSN_{rename}}$ at the time the original load is renamed denotes all stores older than the original load. This value is passed to the eliminated load by the structure that coordinates the elimination. Redundant load elimination is

an example of a technique that requires a different definition of the vulnerability window of a load.

Speculative memory bypassing. Like redundant load elimination, speculative memory bypassing uses a register renaming modification to eliminate loads. However, it does so by converting DEF-store-load-USE chains to DEF-USE chains, using the register map table to point USE to DEF's output [16, 19]. Opportunities for speculative memory bypassing exist because compilers cannot statically allocate all in-flight communication to registers due to separate compilation, a limited number of architectural registers, and the limitations of alias analysis.

Speculative memory bypassing is also a form of speculation of loads with respect to stores. A bypassed load is speculating that the bypassing store is the last store to write to its address. Like redundant load elimination, speculative memory bypassing can take place "outside the window" and as a result requires load re-execution for verification. Suitable SVW definitions for $\mathbf{SSN_{nvul}}$ and the re-execution filter test can be used to filter these re-executions. Here the window definition is $\mathbf{bypassed\text{-}ld.SSN_{nvul} = bypassing\text{-}st.SSN}$, and the re-execution filter test allows the load to skip re-execution if $\mathbf{SSBF[bypassed\text{-}ld.addr] == bypassed\text{-}ld.SSN_{nvul}}$. Speculative memory bypassing is an example of a technique that requires a different re-execution filter test.

SVW for multiple forms of load speculation simultaneously. A processor that uses multiple forms of load speculation can use a single SVW implementation with a single SSBF to filter re-executions for all of them. The key to seamless integration is to identify dynamic loads that are subject to multiple forms of speculation and to compose the corresponding SVW definitions and re-execution filter tests in a safe way. For instance, in a shared-memory multiprocessor an eliminated load is both vulnerable to all stores from the original load *and* to stores from other threads that commit before it does. The obviously safe thing to do is to treat the load as vulnerable to the union of these sets of stores by setting $\mathbf{ld.SSN_{nvul} = MIN(ld.SSN_{nvul\text{-}RLE}, ld.SSN_{nvul\text{-}MP})}$. As it turns out, however, simply setting $\mathbf{ld.SSN_{nvul} = ld.SSN_{nvul\text{-}RLE}}$ is also safe because if $\mathbf{ld.SSN_{nvul\text{-}MP} < ld.SSN_{nvul\text{-}RLE}}$ then it is the original load (not the redundant load) which is vulnerable to those additional stores. Similar composition is possible for speculative memory bypassed loads in a multiprocessor environment.

What forms of load speculation is SVW not appropriate for? Because it fundamentally tracks stores, SVW can filter re-executions for techniques that perform speculation on loads with respect to stores, and more specifically with respect to older stores (although it is not clear what advantage a certain technique may gain by speculating loads relative to younger stores). All the techniques we have seen—speculative scheduling both within a single thread and in a multiprocessor setting, speculative forwarding, load elimination and store-load bypassing—perform this kind of speculation. SVW cannot filter re-executions for techniques that speculate loads with respect to things other than stores. For instance, SVW cannot filter re-executions for load value prediction [13].

3.5. Discussion

This section examines SVW and the SSBF from three different viewpoints.

SSBF vs. a data cache. With its position in the commit pipeline immediately before data cache access, the SSBF can be thought of as a miniature data cache which filters re-execution accesses to the primary data cache. But if load re-execution accesses to the primary data cache are to be avoided, why are load re-execution accesses to a second cache that is accessed in series with the primary cache (i.e., the SSBF) acceptable? The answer is a simple one of latency, capacity, and bandwidth. The SSBF is much smaller than the data cache—typically 32 to 64 times smaller in fact—and is not accessed by out-of-order load execution. These factors make it much easier to provide high bandwidth single-cycle commit pipeline access to the SSBF than it is to provide similar access to the data cache. Because SSBF bandwidth is cheap, there are no commit pipeline stalls due to contention. Because SSBF access is single-cycle and loads and stores access the SSBF in the same stage, SSBF access does not form a "critical loop" [4].

The SSBF is smaller than the data cache for two reasons. First, each SSBF holds SSNs which are narrower than full 32- or 64-bit values. Second and more importantly, it theoretically only needs enough capacity to track in-flight stores. This advantage comes from the fact that SVW's safety test is an inequality ($\mathbf{ld.SSN_{nvul} < SSBF[ld.addr]}$) while that of a cache is necessarily an equality. This allows the SSBF to implic-

itly represent evicted entries and to continue to filter accesses to the data cache even when those entries are tested. A small value cache cannot do that and filters accesses to the primary cache less effectively [15].

SSBF vs. a store queue. In a tagged set-associative SSBF, each set behaves like a small queue. A tagged SSBF with a single set is therefore analogous to a conventional store queue. What is it about an SSBF that allows it to easily be made set-associative whereas turning a store queue into a set-associative structure requires substantially more effort [24]? The answer lies in the SSBF's usage. A store queue is difficult to make set-associative because it is intolerant of set overflow. If an entry is prematurely aged out of a store queue set, the result could be incorrect execution. At the same time, it is difficult to gracefully manage set overflow in a store queue, *e.g.*, by stalling dispatch. Set-associativity implies address-indexing and in the out-of-order execution engine, store addresses become available out-of-order. The SSBF is managed in-order and could gracefully deal with set overflow if it had to. However, there is no need as the SSBF is quite tolerant of set overflow. If an SSBF entry is prematurely aged out of a set, the only potential result is a superfluous replay.

SSBF vs. MCB/ALAT and SLT. The Memory Conflict Buffer (MCB) [7] and Intel Itanium's Advanced Load Alias Table (ALAT) [10] are architectural mechanisms that verify software speculation on loads with respect to stores. Software speculation splits conventional loads into a speculative load and a non-speculative check. The load deposits its address and a token—usually its output register name—in the MCB/ALAT. Subsequent stores search the MCB/ALAT for matching addresses and invalidate the corresponding tokens. The check searches the MCB/ALAT; if the token is still valid the check passes, otherwise it jumps to recovery code.

The Speculative Loads Table (SLT) is a microarchitectural mechanism that verifies speculative load scheduling in the presence of out-of-order store scheduling [18]. Speculative loads deposit their addresses, values, and the sequence numbers of their forwarding store (if the load forwards) in the SLT. Committing stores check the SLT for matching addresses. Loads with matching addresses, mismatching values, and forwarding sequence numbers smaller than the number of the committing store are marked as mis-speculated and trigger flushes.

Both MCB/ALAT and SLT—tables that track speculative load addresses—can be converted to filter load re-executions for hardware optimizations like the ones presented here. Speculative loads deposit their addresses at execution. Committing stores invalidate entries with matching addresses. Before re-execution, speculative loads check the table; if their entries are still valid, they can skip re-execution safely.

An SSBF—which accomplishes the same function by tracking non-speculative store SSNs rather than speculative store addresses—has several advantages over a converted MCB/ALAT/SLT. MCB/ALAT/SLT entries are allocated out-of-order and may need to be explicitly reclaimed on branch mis-predictions. SSBF entries are allocated in-order and are never explicitly reclaimed. In addition, a converted MCB/ALAT/SLT cannot be used to filter re-executions for hardware techniques that involve two dynamic instructions or operate "outside the window".

SSBF vs. other re-execution filtering time-stamps. The LEVO architecture [26] uses tagging to suppress instruction re-execution. However, LEVO is a quasi-dataflow machine with a static instruction and it uses tagging for all instructions simply to establish program order. SVW uses tagging to establish vulnerability windows for speculative loads.

4. Experimental Evaluation

The original SVW paper is primarily concerned with showing that SVW filtered load re-execution is a low-overhead verification mechanism that can flexibly support multiple load speculation techniques [22]. This paper presents several enhancements over the original proposal— tighter bounds on SVW windows, and SSBF tagging and associativity—that further reduce the re-execution rate. This evaluation shows that these improvements: i) allow SVW filtered load re-execution to approximate the performance of an ideal (zero latency, infinite bandwidth) verification mechanism, and ii) make SVW itself without re-execution a viable verification option.

Memory hierarchy	32KB, 32B line, 2-way set-associative, 3-cycle access, instruction and data caches. 128-entry, 4-way set-associative, 2-cycle access instruction and data TLBs. 2MB, 64B line, 8-way set-associative, 15-cycle access L2 cache. 200 cycle memory latency. 16B memory bus that operates at 1/4 core processor frequency. A maximum of 16 outstanding misses.
Pipeline	Front end stages: 1 predict, 3 fetch, 1 decode, 1 rename Out-of-order stages: 1 schedule, 2 register read, 1 execute, 1 writeback Commit stages: 1 setup (1 SVW), 3 data cache, 1 commit Minimum 10 cycle branch mis-prediction penalty
Front end	24Kb, hybrid 2bc-gskew, 8b history predictor, 2K-entry, 4-way BTB, 32-entry RAS 6-wide fetch can fetch past non-taken branches, 20 entry fetch queue 4-wide decode and rename
Execution core	4-wide out-of-order issue: 4 integer, 2 floating-point, 2 loads, and 1 store per cycle 128 entry reorder buffer, 48 entry load queue, 24 entry store queue, 50 entry issue queue MIPS R10000-style register renaming with 160 physical registers
Commit	Store-commit and load re-execution share 1 data cache port **16-bit SSNs** **128-entry, 4-way set-associative tagged SSBF (512B total), 1 write port, 2 read ports**
Speculative scheduling	1K-entry modified Store Sets predictor with 128-entry Store Alias Table (SAT)
Speculative forwarding	4K-entry modified Store Sets predictor with 128-entry Store Alias Table (SAT)

Table 3. Simulated processor configuration

To do this, we evaluate SVW in the context of two load speculation techniques: speculative load scheduling [6] and speculative indexed store-load forwarding [23]. As previously described in Section 2.2, these techniques are both logically and physically similar to each other. From the point of view of SVW, the salient difference between them is their raw re-execution rate. Unfiltered speculative scheduling re-executes about 30% of all loads whereas unfiltered speculative forwarding requires all loads to re-execute.

Methodology. The benchmarks are the SPEC2000 and MediaBench suites. The programs are compiled using the Digital OSF C compiler with optimization flags –O3. All programs run to completion, the SPEC programs on their training inputs at 2% periodic sampling with 5% cache and branch predictor warm-up and 10M instructions per sample; the MediaBench programs on their supplied input with no sampling. The timing simulator executes the Alpha AXP user-level instruction set using the ISA and system call modules from SimpleScalar 3.0. It models a superscalar processor with MIPS-style register renaming, out-of-order execution, aggressive branch prediction and a two level on-chip memory system. Table 3 details the simulated processor's configuration.

4.1. Re-Execution Rates and Performance

Figures 5 and 6 show load re-execution overhead—relative to a baseline with ideal, zero-latency, infinite-bandwidth load re-execution—for four re-execution/SVW configurations in the context of speculative scheduling and speculative indexed store-load forwarding, respectively.

Speculative load scheduling. Figure 5 shows relative execution times (bars) and load re-execution rates (rotated text immediately below the bars) for four speculative scheduling configurations. The baseline is speculative scheduling with ideal load re-execution; the IPC of this ideal baseline is shown in text above the corresponding benchmark name. The top graph shows SPECint2000, the middle graph shows SPECfp2000, and the bottom graph shows MediaBench. The bar group on the right hand side of each graph shows geometric mean relative execution time and arithmetic mean re-execution rates.

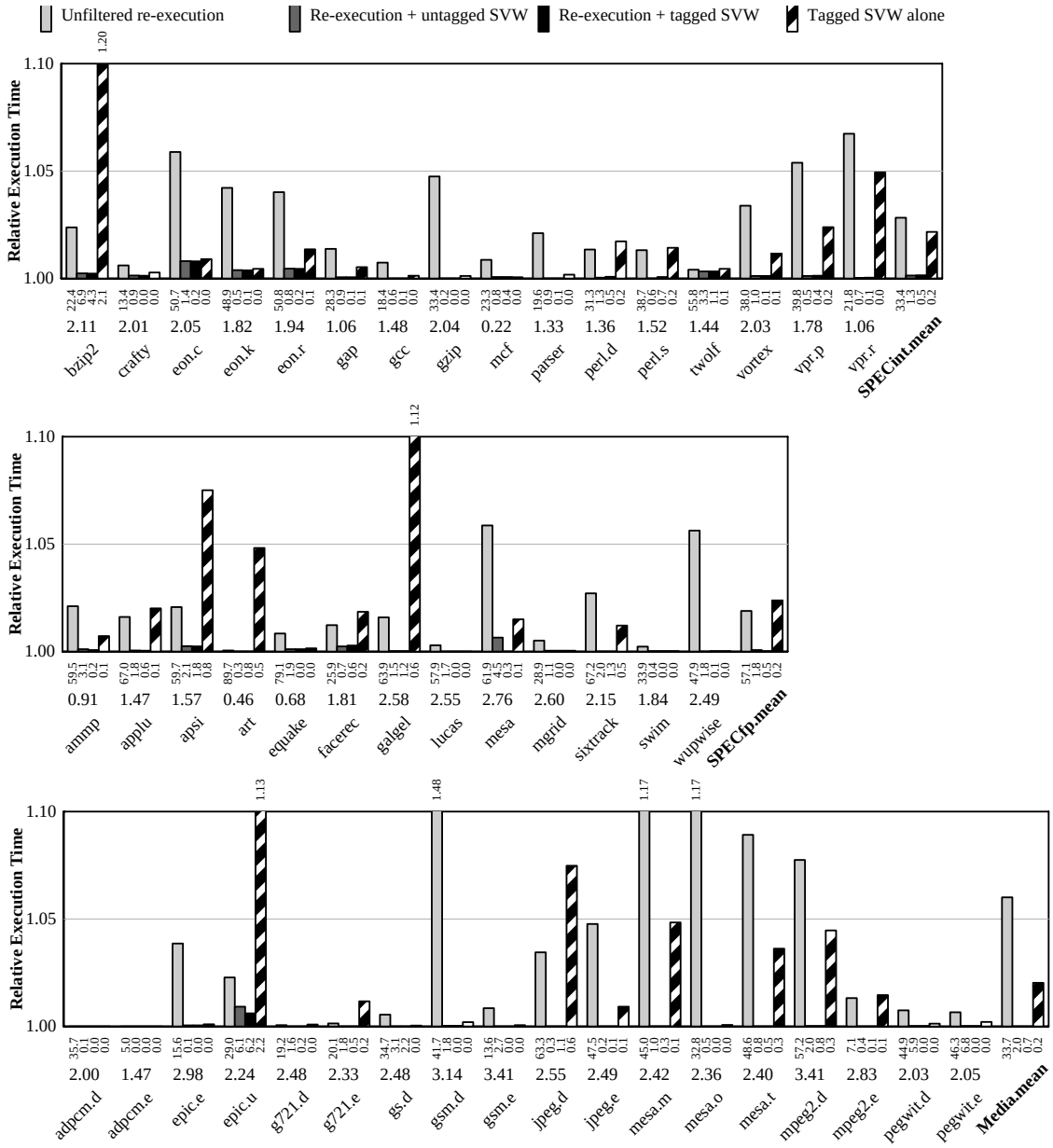

Figure 5. Re-execution rate and execution time overhead for speculative scheduling.
Results for SPECint2000 (top), SPECfp2000 (middle) and MediaBench (bottom). Execution times relative to ideal re-execution (bar) and re-execution rates (number below bar) for four experiments. From left in each group: i) unfiltered re-execution, ii) re-execution with SVW and an untagged SSBF, iii) re-execution with SVW and a tagged SSBF, and iv) SVW with a tagged SSBF alone.

In each bar group, the first configuration starting from the left uses unfiltered re-execution. With an average unfiltered re-execution rate of 33% for SPECint[1] and MediaBench and 57% for SPECfp, average slowdowns over ideal re-execution are only 3% for SPECint, 2% for SPECfp, and 6% for MediaBench. Note, execution time overhead is not necessarily proportional to load re-execution rate. Re-executing loads do not

1. The re-execution rates are higher than those Cain reported [5] because we use the SPCT to train a dependence predictor that synchronizes each load with a single older store. In this scheme, even synchronized loads can be speculative.

cause commit pipeline stalls if they do not occur in the immediate vicinity of stores or other re-executing loads. And even if a commit pipeline stall does occur, it may be absorbed by a wide commit burst the following cycle. Generally, execution overhead is high for programs with high re-execution rates *and* high store commit rates (*i.e.*, IPC's), *e.g.*, *gsm.decode* and *wupwise*.

The second configuration from the left uses re-execution with the originally described SVW scheme with a 512-entry direct-mapped untagged SSBF [22] which is indexed by quad (8-byte) aligned addresses. The average re-execution rate drops to 1.3% for SPECint, 1.8% for SPECfp, and 2.0% for MediaBench with execution time overhead of 0.2%, 0.1%, and 0.05%, respectively.

The third configuration replaces the 512-entry direct mapped untagged SSBF with a 128-entry, 4-way set-associative tagged SSBF which is also indexed by quad-aligned addresses. Although the tagged SSBF contains four times fewer entries than its untagged counterpart, it reduces the re-execution rate further to 0.5%, 0.5%, and 0.7% of committed loads, respectively. The tagged, set-associative SSBF reduces two forms of aliasing that afflict its untagged counterpart. The first form of aliasing is due to different quad-aligned addresses. This form is relatively rare. Empirically, under speculative scheduling, the average size of the vulnerability window of a load is less than 8 stores. In a group of 8 stores, the probability that two stores to different quad-aligned addresses alias in a 512-entry SSBF is less than 1%. The second kind of aliasing is due to non-overlapping sub-quad stores within the same quad-aligned address. Consider the instruction sequence **store4 A**, **store4 A+4**, **load4 A**, where **A** is a quad aligned address. The load forwards from the first store and so is vulnerable to the second younger store. With an untagged direct-mapped SSBF, the load re-executes because its SSBF lookup yields the SSN of the younger (second) store. With a tagged, set-associative SSBF, the load matches the tag and data size of the younger store to determine that there is no actual address overlap. A similar match determines that the load does overlap with the older store; however, the load is not vulnerable to this store and so does not re-execute. Sub-quad aliasing is common in programs with significant numbers of sub-quad stores, *e.g.*, *bzip2* and *pegwit.decode*.

The fourth and final configuration (shown in diagonal stripes) exploits the low re-execution rate provided by the tagged SSBF to eliminate re-execution altogether. In this setup, the processor interprets a positive re-execution filter test—a test that indicates that the load may conflict with a store to which it is vulnerable and must re-execute—as a mis-speculation and initiates recovery without ever checking whether the load's value is actually wrong. Remarkably, this configuration is only 2% slower on average (in each of the three suites) than ideal re-execution and it is faster than unfiltered re-execution. In other words, SVW without re-execution outperforms re-execution without SVW! This result is accompanied by the equally surprising result that eliminating re-execution actually lowers the SVW Bloom filter hit rate, to an average of 0.2% for all suites.

With a tagged, 4-way set-associative SSBF, most of the remaining re-executions are due to silent stores [12]. With re-execution, silent stores induce positive SVW filter tests. For instance, a load that executes in the presence of an older un-executed silent store to the same address does not initially forward from that store and therefore is vulnerable to it. The SVW filter test properly forces the load to re-execute, but re-execution does not detect a mis-speculation—re-execution compares values and the store is silent—and the StoreSets predictor is not be trained to synchronize the load with the silent store. The result is that the same sequence of events repeats on the next dynamic execution of the load. When re-execution is eliminated, a positive SVW filter test is treated as a mis-speculation which in turn trains the StoreSets predictor. Subsequent dynamic executions of the load synchronize with and forward from the silent store, and as a result are not be vulnerable to it and do not re-execute.

Similar effects are observed when SVW is used without load re-execution in conjunction with other forms of load speculation. In speculative forwarding, the forwarding predictor learns to forward from silent stores. In redundant load elimination, the redundancy prediction/tracking mechanism learns not to eliminate loads where a conflicting silent store occurs between the original load and the redundant load. In speculative memory bypassing, the prediction/tracking mechanism learns to bypass loads from silent stores.

A verification scheme consisting of SVW in isolation generally outperforms a scheme consisting of unfiltered re-execution. The notable exceptions are *bzip2* and *epic.unepic*, both of which have SSBF hit rates of 2.2% which result in pipeline flushes once every 200 instructions or so. Both programs suffer from sub-quad aliasing, specifically byte-aliasing. Both benchmarks contain loops that perform byte writes and reads

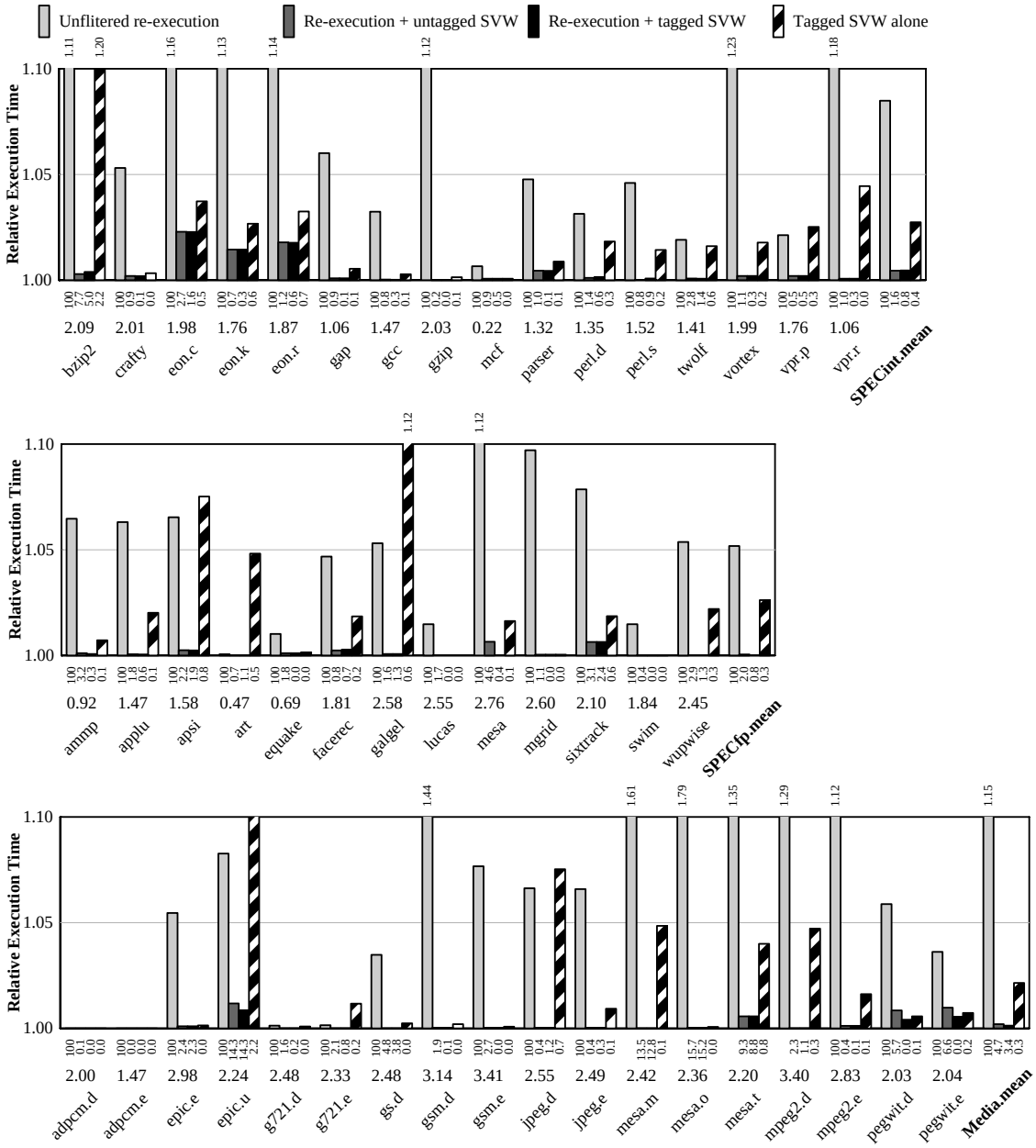

Figure 6. Re-execution rate and execution time overhead for speculative forwarding.

that (may) depend on writes from any of the previous 8 iterations. In a 4-way set-associative tagged SSBF, a sequence of 4 byte writes overflows a set and causes the re-execution—or mis-speculation recovery—of any load that depends on a byte store that is more than 4 stores in the past. As shown in Figure 6—at least for *epic.unepic*—increasing SSBF associativity to 8 eliminates this pathology.

Speculative indexed store-load forwarding. Figure 6 shows the results of similar experiments in the context of speculative indexed forwarding. Despite an *a priori* re-execution rate of 100%, SVW again virtually eliminates the overhead of load re-execution. A verification mechanism consisting of SVW in isolation is only 3% slower on average than an idealized re-execution verification scheme.

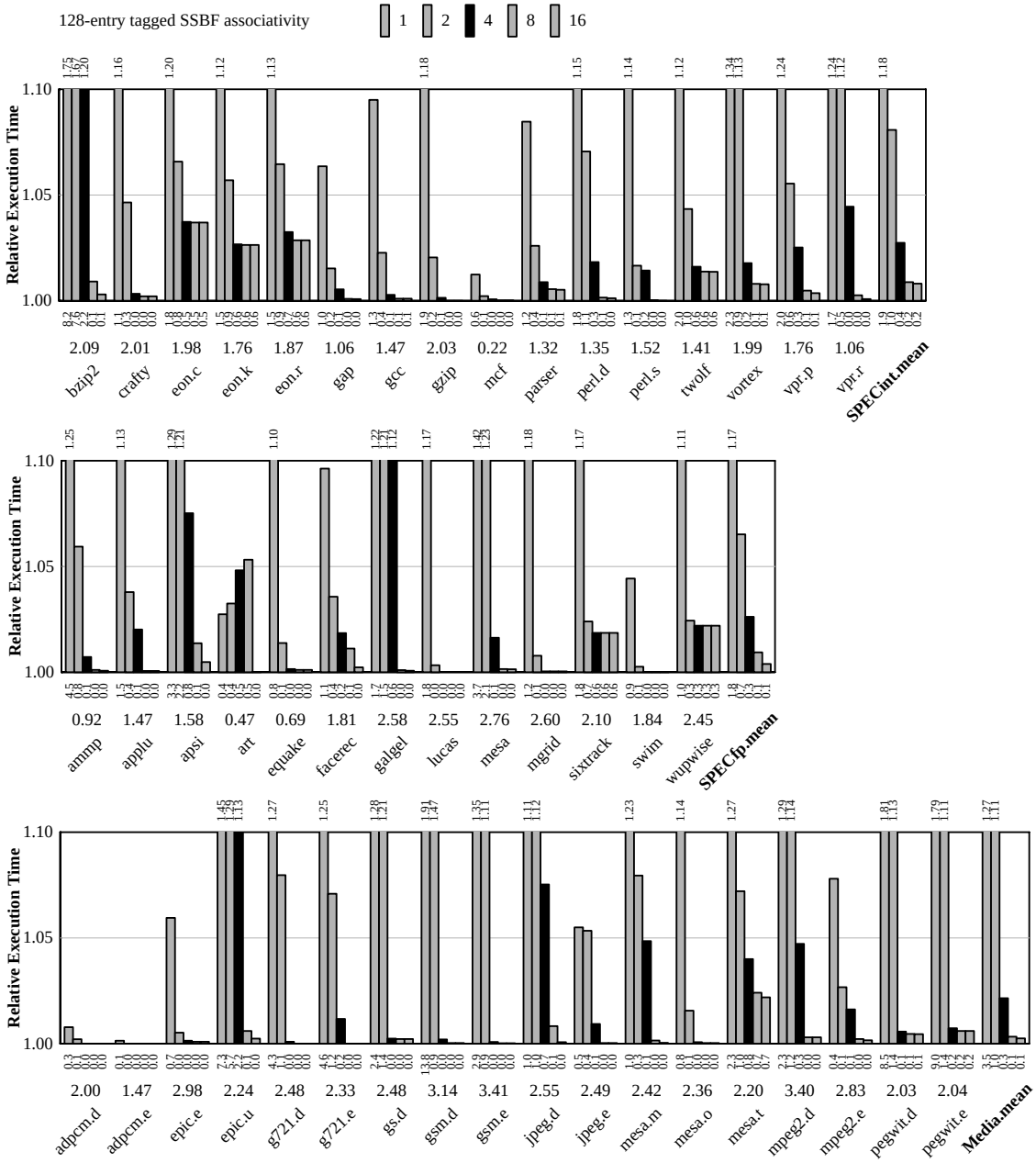

Figure 7. "Re-execution" rates and execution time overhead for tagged SSBFs of varying associativity. All experiments show speculative forwarding with SVW using a 128-entry tagged SSBF, but no re-execution. From left, associativities are: 1, 2, 4 (the default), 8, and 16.

4.2. SSBF Configuration Sensitivity Analysis

As the instruction window grows, more (and more varied) store-load interactions become possible and techniques that speculate on loads relative to older stores invariably require larger and potentially more sophisticated predictors to drive them. In contrast, SVW itself is non-speculative and—unlike the techniques that it verifies—is not subject to the same scalability concerns. The way in which increasing window size adversely affects SVW is by increasing the sizes of vulnerability windows, making the SSBF more prone to

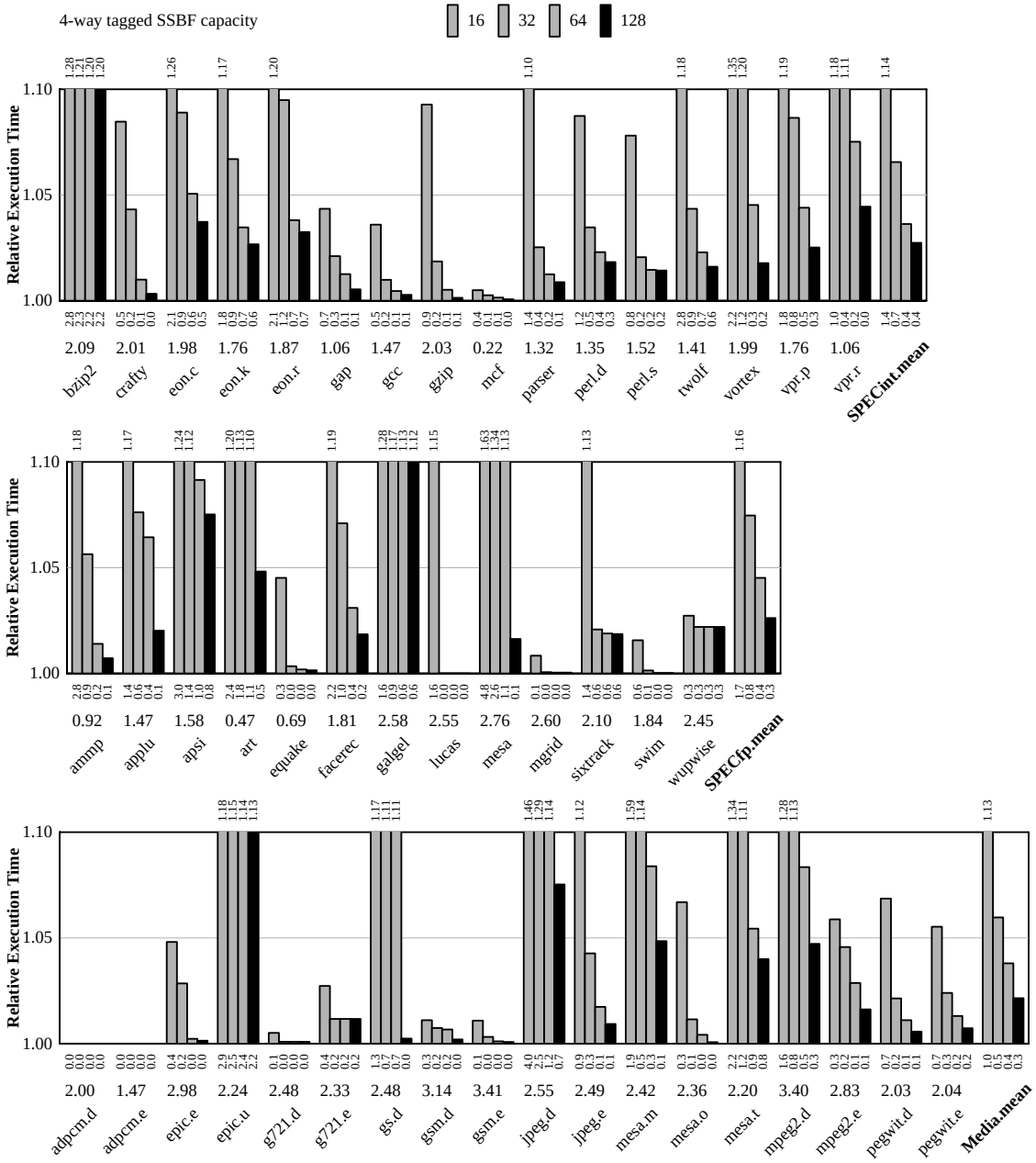

Figure 8. "Re-execution" rates and execution time overheads for SSBFs of varying capacity. All experiments show speculative forwarding with SVW using a 128-entry tagged SSBF, but no re-execution. From left, associativities are: 1, 2, 4 (the default), 8, and 16.

aliasing. Of course, a FIFO set-associative tagged SSBF proactively "de-aliases" mitigating this effect.

Scaling a given load speculation technique to a large window is challenging to do in a realistic way, because effective window sizes are often limited by branch prediction. In this section, we study SVW scalability in a different way: by varying the SSBF configuration for a given speculative technique and a fixed size window of 128 instructions. To amplify the negative performance impact of re-execution, the baseline speculation technique is speculative forwarding (which makes 100% of the loads speculative), we also use SVW in isolation, *i.e.*, without re-execution.

Tagged SSBF associativity. Figure 7 shows the relative execution time and SSBF hit rate (reported mis-speculation rate, there is no actual re-execution here) of SVW with a tagged 128-entry SSBF in isolation. Results are shown for associativities of 1, 2, 4 (our default), 8, and 16.

Previously, we saw that 4-way associative SSBFs are vulnerable to byte-aliasing. Byte aliasing is common in programs with loops that perform byte stores, as many compression and media programs do, *e.g.*, *epic.unepic*, *jpeg.decode*, *mesa.mipmap*, and *mpeg2.decode*. For these programs, an 8-way set-associative SSBF is needed if SVW is used as the sole verification mechanism. Generally speaking, a 4-way set-associative SSBF is necessary, both to minimize "random" aliasing between different quad-aligned addresses and to avoid word-aliasing within a single quad-aligned address.

An interesting effect is visible in *art* (SPECfp) in which increasing associativity from 1 to 8 actually results in slight increases in mis-speculation reporting (*i.e.*, positive SVW test) rates and overhead. Both plummet to zero for a 16-way set-associative SSBF. Here there is a set of 16 stores to 1KB-aligned addresses. In a direct mapped SSBF, pairs of these stores conflict in a one-entry "set". In a 2-way set-associative SSBF, groups of four stores conflict in two-entry sets, slightly increasing the positive test rate. In a 4-way SSBF, groups of eight stores conflict in four-entry sets, further increasing the positive test rate. In an 8-way SSBF, all sixteen stores conflict in one set. These conflicts are resolved by a 16-way SSBF. This behavior is obviously pathological and is not observed in any other program (at least not to the point where it affects overall behavior).

Tagged SSBF capacity. Figure 8 shows the overhead and reported mis-speculation rate of SVW with a tagged 4-way set associative SSBF of varying capacities. From left, capacities are 16, 32, 64 and 128 entries (our default).

As this experiment shows, SVW is less sensitive to reduced capacity than it is to reduced associativity. Reduced capacity increases natural aliasing somewhat, but a FIFO tagged SSBF has active de-aliasing. Reduced associativity reduces natural aliasing somewhat but also directly interferes with the SSBF's ability to de-alias. Even with a 16-entry SSBF, re-execution rates are reduced to an average of under 2% in each of the three suites.

Scalability summary. These experiments suggest (but does not directly show) that achieving a target re-execution rate for a given technique acting in a large window (*e.g.*, 1024 instructions) can be easily accomplished using an SSBF of 4 or 8 way associativity and a capacity of about 512 entries (4KB total storage) and likely with a less provisioned design.

5. Conclusions

Load scheduling and execution are performance critical yet complex functions. To overcome the complexity of these functions while achieving high performance, conventional and proposed techniques exploit speculation. More specifically, they perform speculation on loads with respect to older stores. One well known technique is speculative (intelligent) load scheduling which speculates that a load does not read the same address as that which will be written by an older, still un-executed store. Another recently proposed technique speculates the identity of the potentially forwarding store. This speculation is used to replace non-scalable associative store queue search with simpler, faster indexed access. Any out of order load execution can be viewed as speculative with respect to logically older but still uncommitted stores in other threads of a multithreaded shared-memory program.

Load speculation requires verification. One conceptually simple mechanism that can verify any form of load speculation is in-order re-execution prior to commit. Here, mis-speculation is detected if the re-executed value differs from the value obtained during the initial out-of-order execution. The primary drawback of re-execution is its excessive consumption of data cache bandwidth, a scarce and expensive resource. If a given technique makes a sufficient number of loads speculative—all speculative loads must re-execute in-order to guarantee correctness—the resulting cache bandwidth contention can nullify the benefit the technique aims to provide.

Store Vulnerability Window (SVW) is an address-based filtering mechanism that reduces the re-execu-

tion requirements of a given load speculation technique significantly. The idea behind SVW is that a given technique makes a load speculative with respect to some range of dynamic stores that is older than the load itself. The load should not re-execute if it does not read the same address as any store within this vulnerability range. SVW formalizes this concept using a store sequence numbering scheme to define per-load store vulnerability windows and an adapted Bloom filter to track the sequence numbers of the most recent committed stores on a per-address basis.

A simple SVW implementation consisting of 16-bit sequence numbers and a 128-entry Bloom filter with 1KB total storage reduces re-executions by a factor of 200. This reduction effectively eliminates the overhead of re-execution-based verification even for techniques with *a priori* re-execution rates of 100%. It also allows SVW to act as a replacement for re-execution, with Bloom filter hits interpreted as mis-speculations. An SVW-only verification scheme is only 3% slower on average than idealized—instant latency, infinite bandwidth—verification.

Acknowledgments

The author thanks the anonymous reviewers for their comments and suggestions, Vlad Petric for the idea of using SVW as a replacement for re-execution, Milo Martin for the idea of reducing the re-execution rate by making the SVW Bloom filter a tagged set-associative FIFO structure, and Tingting Sha for generating the data for this paper. This work was supported by NSF CAREER award CCR-0238203 and NSF CPA grant CCF-0541292.

References

[1] T. Austin. “DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design.” In *Proc. 32nd International Symposium on Microarchitecture*, pages 196–207, Nov. 1999.

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

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

[4] E. Borch, E. Tune, S. Manne, and J. Emer. “Loose Loops Sink Chips.” In *Proc. 8th International Symposium on High Performance Computer Architecture*, Jan. 2002.

[5] H. Cain and M. Lipasti. “Memory Ordering: A Value Based Definition.” In *Proc. 31st International Symposium on Computer Architecture*, pages 90–101, Jun. 2004.

[6] G. Chrysos and J. Emer. “Memory Dependence Prediction using Store Sets.” In *Proc. 25th International Symposium on Computer Architecture*, pages 142–153, Jun. 1998.

[7] D. Gallagher, W. Chen, S. Mahlke, J. Gyllenhaal, and W. Hwu. “Dynamic Memory Disambiguation Using the Memory Conflict Buffer.” In *Proc. 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–193, Oct. 1994.

[8] A. Garg, M. Rashid, and M. Huang. “Slackened Memory Dependence Enforcement: Combining Opportunistic Forwarding with Decoupled Verification.” In *Proc. 33rd International Symposium on Computer Architecture*, Jun. 2006.

[9] K. Gharachorloo, A. Gupta, and J. Hennessy. “Two Techniques to Enhance the Performance of Memory Consistency Models.” In *Proc. of the International Conference on Parallel Processing*, pages 355–364, Aug. 1991.

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

[11] R. Kessler. “The Alpha 21264 Microprocessor.” *IEEE Micro*, 19(2), Mar./Apr. 1999.

[12] K. Lepak and M. Lipasti. “On The Value Locality of Store Instructions.” In *Proc. 27th International Symposium on Computer Architecture*, pages 182–191, Jun. 2000.

[13] M. Lipasti, C. Wilkerson, and J. Shen. "Value Locality and Load Value Prediction." In *Proc. 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, Oct. 1996.

[14] M. Martin, M. Hill, and D. Wood. "Token Coherence: Decoupling Performance and Correctness." In *Proc. 30th International Symposium on Computer Architecture*, pages 182–193, Jun. 2003.

[15] A. Meixner and D. Sorin. "Dynamic Verification of Memory Consistency in Cache-Coherent Multithreaded Computer Architectures." In *Proc. 2006 International Conference on Dependable Systems and Networks*, pages 73–82, Jun. 2006.

[16] A. Moshovos and G. Sohi. "Streamlining Inter-Operation Communication via Data Dependence Prediction." In *Proc. 30th International Symposium on Microarchitecture*, pages 235–245, Dec. 1997.

[17] A. Moshovos and G. Sohi. "Read-After-Read Memory Dependence Prediction." In *Proc. 32nd International Symposium on Microarchitecture*, pages 177–185, Nov. 1999.

[18] S. Onder and R. Gupta. "Dynamic Memory Disambiguation in the Presence of Out-of-Order Store Issuing." In *Proc. 32nd International Symposium on Microarchitecture*, pages 170–176, Nov. 1999.

[19] V. Petric, A. Bracy, and A. Roth. "Three Extensions to Register Integration." In *Proc. 35th International Symposium on Microarchitecture*, pages 37–47, Nov. 2002.

[20] V. Petric, T. Sha, and A. Roth. "RENO: A Rename-Based Instruction Optimizer." In *Proc. 32nd International Symposium on Computer Architecture*, pages 98–109, Jun. 2005.

[21] 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.

[22] A. Roth. "Store Vulnerability Window (SVW): Re-Execution Filtering for Enhanced Load Optimization." In *Proc. 32nd International Symposium on Computer Architecture*, pages 458–468, Jun. 2005.

[23] T. Sha, M. Martin, and A. Roth. "Scalable Store-Load Forwarding via Store Queue Index Prediction." In *Proc. 38th International Symposium on Microarchitecture*, pages 159–170, Nov. 2005.

[24] S. Stone, K. Woley, and M. Frank. "Address-Indexed Memory Disambiguation and Store-to-Load Forwarding." In *Proc. 38th International Symposium on Microarchitecture*, pages 171–182, Nov. 2005.

[25] E. Torres, P. Ibanez, V. Vinals, and J. Llaberia. "Store Buffer Design in First-Level Multibanked Data Caches." In *Proc. 32nd International Symposium on Computer Architecture*, pages 469–480, Jun. 2005.

[26] A. Uht, D. Morano, A. Khalafi, and D. Kaeli. "Levo - A Scalable Processor with High IPC." *Journal of Instruction Level Parallelism*, 5, Aug. 2003. (http://www.jilp.org/vol5/).

[27] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan. "Speculation Techniques for Improving Load-Related Instruction Scheduling." In *Proc. 26th Annual International Symposium on Computer Architecture*, pages 42–53, May 1999.