Decoupled Store Completion/Silent Deterministic Replay: Enabling Scalable Data Memory for CPR/CFP Processors

Andrew Hilton and Amir Roth

Computer and Information Sciences Department, University of Pennsylvania

{adhilton, amir}@cis.upenn.edu

Abstract

CPR/CFP (Checkpoint Processing and Recovery/Continual Flow Pipeline) support an adaptive instruction window that scales to tolerate last-level cache misses. CPR/CFP scale the register file by aggressively reclaiming the destination registers of many in-flight instructions. However, an analogous mechanism does not exist for stores and loads. As the window expands, CPR/CFP processors must track all in-flight stores and loads to support forwarding and detect memory ordering violations.

The previously-described SVW (Store Vulnerability Window) and SQIP (Store Queue Index Prediction) schemes provide scalable, non-associative load and store queues, respectively. However, they don't work smoothly in a CPR/CFP context. SVW/SQIP rely on the ability to dynamically stall some loads until a specific older store writes to the cache. Enforcing this serialization in CPR/CFP is expensive if the load and store are in the same checkpoint.

We introduce two complementary procedures that implement this serialization efficiently. **Decoupled Store Completion (DSC)** allows stores to write to the cache before the enclosing checkpoint completes execution. **Silent Deterministic Replay (SDR)** supports mis-speculation recovery in the presence of DSC by replaying loads older than completed stores using values from the load queue. The combination of DSC and SDR enables an SVW/SQIP based CPR/CFP memory system that outperforms previous designs while occupying less area.

Categories and Subject Descriptors

C.1.1 [*Processor Architectures*]: Single Data Stream Architectures—Pipeline processors

General Terms

Design, Performance

Keywords

Checkpoint processors, load-store queues

ISCA '09, June 20–24, 2009, Austin, Texas, USA.

1. Introduction

Building an instruction window that exposes both instruction- and memory- level parallelism requires scaling four critical structures: register file, issue queue, load queue, and store queue. CPR (Checkpoint Processing and Recovery) [1] scales the register file by managing physical registers at a checkpoint granularity. CPR reclaims registers that are not named in checkpoints, overcoming the traditional constraint that every register-writing in-flight instruction have a physical register allocated to its destination. CFP (Continual Flow Pipelines) extends CPR to scale the register file and issue queue in the presence of last-level cache misses [23]. Miss-dependent instructions drain from the window and release their issue queue and register resources. When the miss returns, they are re-injected into the window, re-acquire these resources and execute.

Unfortunately, checkpoint-granularity management does not apply to stores and loads in a straightforward way. As the window grows, CPR/CFP must continue to track *all* in-flight stores and loads to support in-order store commit, store-load forwarding, and the detection of both intra-thread and inter-thread memory ordering violations. Existing CPR/CFP data memory system proposals rely on hierarchy—traditional associative primary load and store queues backed by secondary queues. The secondary queues in these designs suffer from one of two problems: they are either area and power inefficient [1] or they activate only in the shadow of a last-level cache miss and don't support a large window in the more general cases of data cache misses, store misses, or regions of otherwise low ILP [6].

Conceptually, the simplest and most efficient design uses flat non-associative load and store queues that support a large window under all conditions. As such, the previously-proposed Store Vulnerability Window (SVW) [3, 16] and Store Queue Index Prediction (SQIP) [21] are attractive. However, there is a subtle but important mismatch between these techniques and CPR/CFP. Specifically, both SQIP and SVW fundamentally rely on the ability to *dynamically serialize a particular store-load pair*, *i.e.*, to stall either load execution (SQIP) or re-execution (SVW) until the store completes and writes to the data cache. Unfortunately, CPR manages not only registers at a checkpoint granularity but also stores—stores cannot begin writing to the cache until the entire checkpoint finishes execution and the possibility of a squash can be ruled out. Dynamically serializing a store-load pair that occurs in the same checkpoint requires flushing and splitting the checkpoint into two during subsequent execution. This is slow and wastes checkpoints.

In this paper, we propose two complementary mechanisms that allow SVW/SQIP to operate effectively in a CPR/CFP environment by reconciling checkpoint-granularity register management with traditional instruction-granularity store completion. **DSC (Decoupled Store Completion)** allows stores to complete and write to

the cache *before* the enclosing checkpoint completes execution—although stores still complete in program order. **SDR (Silent Deterministic Replay)** supports mis-speculation recovery in the presence of DSC. In CPR/CFP, when an un-checkpointed instruction mis-speculates, the processor squashes to the next older checkpoint, and re-starts execution there. With DSC, it is possible that completed stores—and older loads over-written by them—would have to be flushed and re-executed. SDR re-constitutes the values of these loads from the load queue, avoiding both data cache access and potential write-after-read hazards with younger DSC stores.

DSC and SDR act synergistically with each other and with SVW and SQIP. SVW already uses a load queue that captures values, and so SDR comes essentially for free. SDR simplifies the SVW implementation by removing the need to checkpoint SVW's Bloom filter. SDR enables DSC which improves SVW/SQIP's performance and eliminates the need to speculatively serialize store-load pairs. Finally, DSC/SDR simplify the multiprocessor interface of CPR/CFP by making it identical to the interface of a traditional instruction-granularity processor.

Our work makes the following contributions:

- We identify a mismatch between SVW/SQIP and CPR/CFP. CPR/CFP effectively completes stores at checkpoint granularity whereas SVW/SQIP requires stores to complete at instruction granularity.
- We introduce DSC/SDR, complementary procedures which resolve this mismatch by supporting instruction-granularity store completion in a checkpoint substrate.
- We show that for CPR/CFP, a memory system using SVW/SQIP load and store queues and augmented with DSC/SDR outperforms existing designs and occupies less area.

2. Background: SVW/SQIP

SVW and SQIP were proposed in the context of instruction-granularity ROB-based processors. We review them here.

Load Re-Execution and SVW (Store Vulnerability Window). Conventional processors detect intra- and inter-thread memory ordering violations by "snooping" (*i.e.*, searching) a load queue that tracks the addresses of all in-flight loads in program order. The primary drawback of this approach is the non-scalability of associative search. An alternative approach is to re-execute loads in-order prior to commit, detecting a violation when the values obtained by (out-of-order) execution and (in-order) re-execution differ [3]. Re-execution is cheaper than associative search for large windows. It also does not signal spurious violations. Finally, it can detect violations in other forms of load speculation, not just store-load ordering speculation. Its primary drawback is that it consumes significant data cache bandwidth.

Store Vulnerability Window (SVW) is an address-based filter that reduces the data cache bandwidth requirements of load re-execution [16, 17]. The basic idea is that a load can skip re-execution if no store has written to its address in a sufficiently long time. SVW identifies dynamic stores using monotonically increasing sequence numbers (SSNs). A store's store queue position is the low order bits of its SSN. A small address-indexed table called the Store Sequence Bloom Filter (SSBF) contains the SSNs of the youngest store to write to each (hashed) address. An extended commit pipeline processes executed stores and loads in order. Stores write their SSNs at the SSBF index corresponding to their address. The global register SSN_{verify} tracks the SSN of the youngest store to have updated the SSBF. Loads read the SSBF using their address and determine whether they need to re-execute based on the SSN they find. The re-execution test is specific to the type of speculation the load undergoes.

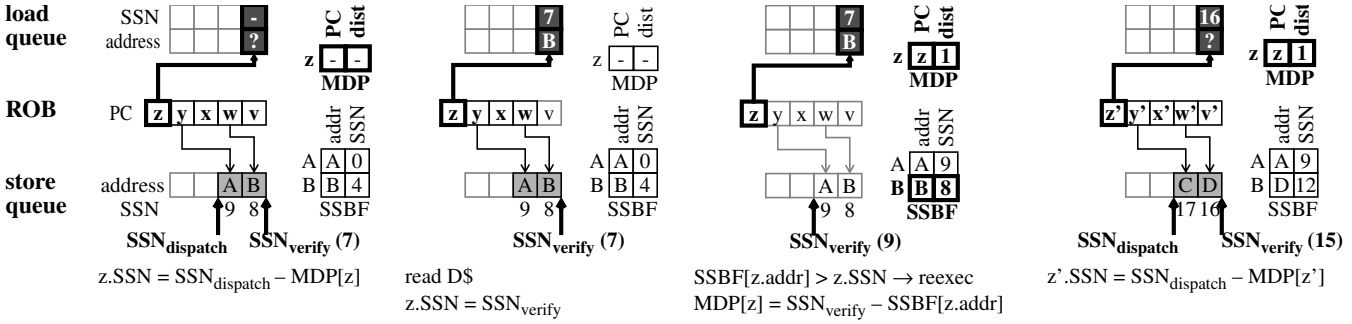
SQIP (Store Queue Index Prediction). Conventional processors perform store-load ordering speculation. A more aggressive form of speculation is forwarding speculation in which the processor speculates for every load the precise identity of the forwarding store. Forwarding speculation supports a scalable store queue which is both age-ordered and non-associative [21], avoiding the scalability problems of fully-associative designs and the set-overflow problems of address-indexed ones. Forwarding speculation can be performed with very high accuracy (99.9% for most programs) because only a small fraction of loads forward, and because most forwarding behavior is stable and predictable. Forwarding prediction schemes that represent store-load dependences as dynamic store distances [22, 24, 28] mesh with SVW and typically yield the highest prediction accuracies.

Working Example. Figure 1 shows a working example of SQIP and SVW. In our notation, data addresses are uppercase letters, instruction PCs are lowercase letters, SSNs and store distances are numbers, and data values are irrelevant. The figure uses three instructions: stores w and x to addresses B and A, followed by load z to address B. The figure shows four snapshots in time. Each snapshot shows a ROB (each entry contains a PC), a four entry load queue on top (each entry contains an address and the SSN of the predicted forwarding store), a four-entry store queue (each entry contains an address, the entry's SSN is implicit), two sets (corresponding to hashed addresses A and B) of a direct-mapped SSBF (each entry contains an address tag and and an SSN), and a one-entry memory dependence predictor (each entry contains a PC tag and a store distance).

The dynamic instruction stream is tracked by three explicit global pointers. The position of these pointers in the dynamic store stream is marked by SSNs. We have already introduced SSN_{verify}. Stores older than SSN_{verify} have updated the SSBF. They have also transitioned from the store "queue" to the store "buffer" and cannot be aborted. The corresponding instruction stream pointer is P_{verify}. Loads older than P_{verify} are verified and cannot be aborted as well. Global pointer $SSN_{store-complete}$ ($P_{store-complete}$) tracks the youngest store to write to the data cache. Global pointer $SSN_{dispatch}$ ($P_{dispatch}$) tracks the youngest dispatched store (instruction).

At T1, (Figure 1a) stores w and y have dispatched (they are older than $SSN_{dispatch}$) and executed. Load z dispatches and consults the memory dependence predictor. There is no matching entry so the load will only access the data cache. At T2 (Figure 1b) load z executes, calculates its address (B), and reads the cache. Load z is unaware of older store w to address B because it cannot search the store queue. Load z records the current value of SSN_{verify} (7)—it is "vulnerable" to all younger stores (*i.e.*, stores with higher SSNs). At T3 (Figure 1c), load z is verified; notice stores w and y have updated the SSBF. Load z reads the SSBF set corresponding to address B and finds a store to address B with SSN 8. As load z is only safe with respect to SSNs ≤ 7, it must re-execute. Re-execution shows it received the wrong value. Load z is squashed and the memory dependence predictor is trained to forward from the appropriate store. The distance to the forwarding store is computed as SSN_{verify}–SSBF[z.addr] (in this case 9–8=1). At T4 (Figure 1), load z' (a future instance of load z) predicts a dependence on y' by computing $SSN_{dispatch}$–MDP[z].

Multiprocessor Issues. Processors that verify intra-thread store-load ordering by snooping the load queue use the same mechanism to verify inter-thread store-load ordering. Load queue search is triggered by invalidation messages and (for directory protocols and some memory consistency models) local cache evictions as well. How does SVW track these events and "trip" loads that are exposed to them? SVW treats events that would require load queue snooping as "asynchronous stores" from the same thread. It tracks them using a second SSBF (SSBF-MT) which is organized at a



(a) T1: dispatch load z **(b) T2: execute load z** **(c) T3: verify load z** **(d) T4: dispatch load z'**

Figure 1. A working example of SVW/SQIP in a ROB processor

cache line granularity. Verifying loads read both the SSBF and the SSBF-MT and re-execute if they hit in either. When an asynchronous store event occurs, SVW writes SSN_{verify} +1 in the corresponding SSBF-MT entry. The value SSN_{verify} +1 trips all exposed loads that had not been verified when the event occurred.

A second issue concerns memory consistency models that disallow store buffers, *e.g.*, sequential consistency (SC). SVW can enforce SC by stalling P_{verify} at any load if $P_{verify} > P_{store-complete}$.

3. CPR/CFP Memory Systems

Previously proposed memory systems for CPR/CFP use hierarchical designs. Small associative load and store queues handle ordering and forwarding for the youngest instructions in the window. Older loads and stores spill to larger secondary structures.

HLQ (Hierarchical Load Queue). The secondary load queue is address-indexed and set-associative [6]. Set-overflow is handled either by squashing the incoming load's checkpoint or by stalling the load until the oldest checkpoint commits and load queue entries come free. The disadvantage of HLQ is the performance penalty of resolving set conflicts, and the cost of over-sizing the load queue to minimize set conflicts.

HSQ (Hierarchical Store Queue). There are two proposed designs for secondary store queues. The first is a large associative queue with an access latency that matches that of the L2 [1]. To avoid accessing the secondary queue on every load—and effectively increasing load latency to L2 latency—the secondary queue is guarded by an address-indexed Bloom filter (MTB). Load latency elongates to L2 latency only on an MTB "hit". The disadvantage of HSQ is the area and power cost of the secondary queue.

SRL (Store Redo Log). The follow-on design replaces the large associative store queue with two structures which activate on a last-level cache miss [6]. While the miss is pending, miss-independent stores drain into a small forwarding cache [11], supporting forwarding to miss-independent loads. In parallel, *all* stores drain into an age-ordered Store Redo Log (SRL). When the miss returns, the forwarding cache is flushed and the miss slice is re-injected into the window. Miss-dependent stores re-acquire first-level store queue entries for store-load forwarding within the slice. When they complete, they write their values to their pre-determined positions in the SRL. When the miss slice completes execution, dispatch at the tail of the window resumes and the SRL drains to the data cache in program order. An address indexed Bloom filter (LCF) supports limited forwarding from the SRL during the draining process. Loads that cannot forward using the LCF stall until draining completes. SRL has both performance and algorithmic complexity disadvantages. Performance-wise, SRL provides a large store queue only in the shadow of a last-level cache misses and not in the general case. Complexity-wise, although the SRL, LCF, and forwarding cache are physically simple, their associated management algorithms—which are required to detect forwarding violations—are complex.

Initial SVW/SQIP Performance Study. Figure 2 shows IPC speedup of HLQ/HSQ [1], HLQ/SRL [6]—both with 8-way set-associative load queues—and two SVW/SQIP memory systems over a baseline with conventional fully-associative load and store queues with 64 and 48 entries, respectively. The scalable load queues (HLQ, SVW) have 512 entries, the scalable store queues (HSQ, SRL, SQIP) have 256 entries. Table 1 describes the full configuration in detail.

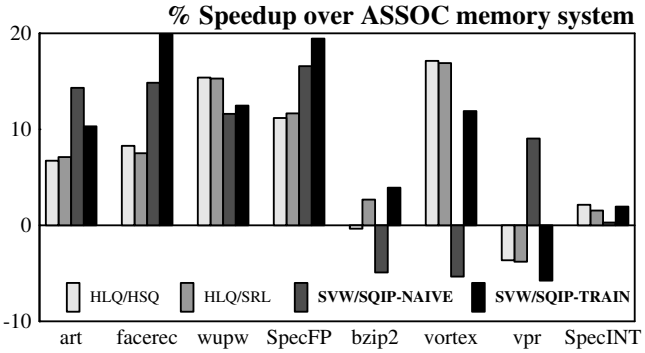
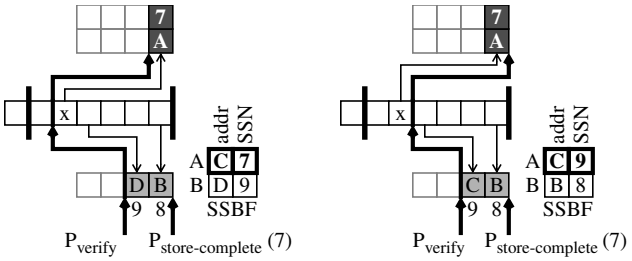


Figure 2. SVW/SQIP in a CPR/CFP processor

HLQ/HSQ and HLQ/SRL generally have similar performance, 12% speedups on SpecFP and 2% on SpecINT, with occasional small slow-downs (*e.g.*, *vpr*). The first-cut SVW/SQIP implementation is SVW/SQIP-NAIVE. In SpecFP, SVW/SQIP-NAIVE generally out-performs the existing designs because it provides uniform low-latency forwarding and conflict-free load tracking. However, in SpecINT it generally under-performs them. This low performance is caused not by forwarding mis-speculation, but by SVW's frequent need to serialize a load-store pair—to force the store to complete and write to the data cache before the load can verify.

The Use of Store-Load Serialization in SVW/SQIP. Even conventional non-SVW/SQIP processors occasionally need to serialize a store-load pair—the common cases here are a narrow store followed by a wide load to an overlapping address, or a fence instruction. However, the SVW/SQIP combination actively uses serialization to ensure correctness and forward progress.

SVW can verify most loads without re-execution but it does need to re-execute 1–2% of loads. The use of SQIP would appear to imply that load re-execution requires an empty store buffer (the portion of the store structure older than SSN_{verify}) because re-execution cannot associatively search the store buffer. In actuality,



(a) Verifiable load **(b) Verification "hole"**

Figure 3. Store-load serialization in SVW/SQIP

loads can frequently re-execute with a non-empty store buffer. There are three relevant cases. In the simplest case, the SSBF contains an entry with a matching address. If the entry's SSN is younger than $SSN_{store-complete}$, then the correct value is in the store buffer at the index indicated by that SSN. If the entry's SSN is $SSN_{store-complete}$ or older, the correct value is in the data cache. Either way, re-execution can proceed without draining the store buffer. Figure 3 shows the other two cases. In each case, load x is being verified—its address is A and its forwarding SSN is 7. The figure shows a direct-mapped SSBF with two sets. In Figure 3a, the SSBF does not contain an entry with a matching address so the relevant SSBF entry is the oldest one (in this case only one) in the corresponding set ($SSN_{ssbf-set}$). The fact that $SSN_{ssbf-set} \leq SSN_{store-complete}$ unambiguously implies that the store buffer does not contain an entry with a matching address, and re-execution can proceed simply by accessing the data cache. In Figure 3b, again there is no matching address but this time $SSN_{ssbf-set} > SSN_{store-complete}$. Here, re-execution cannot immediately proceed because there may be a conflicting store in the store buffer in the region between $SSN_{ssbf-set}$ and $SSN_{store-complete}$. In other words, the SSBF has a "hole"—there *may* be a store that has "aged" out of the SSBF but that is still in the store buffer. Without searching the store buffer there is no way to know.

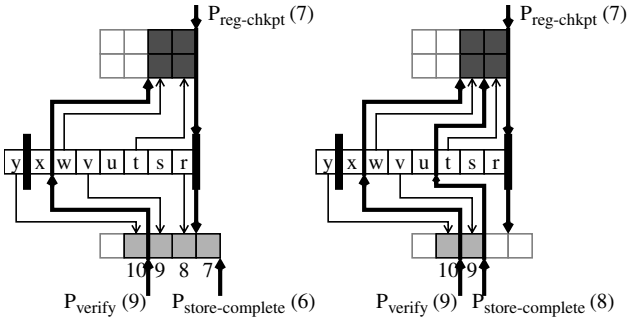
Handling Load-Store Serialization in CPR/CFP. In a ROB (*i.e.*, instruction-granularity) processor, SSBF holes close naturally. Eventually, enough stores will drain from the store buffer to the data cache and turn the scenario in Figure 3b into the one in Figure 3a. However, in CPR/CFP holes do not close naturally if the load and store reside in the same checkpoint. The load will not re-execute and verify until the store writes to the cache, the store will not write to the cache until the checkpoint commits, and the checkpoint will not commit until all instructions in it execute and all loads verify. Resolving this stalemate requires squashing the entire checkpoint and, on the re-execution, creating a second checkpoint immediately before the load—thereby placing the load and store in two different checkpoints—in order to avoid a recurrence of the same exact situation. This is SVW/SQIP-NAIVE's strategy.

One way to reduce this penalty is to remember the PCs of loads that experience holes in a small table and to create a checkpoint whenever a future instance of one of these PCs is renamed. This is the scheme used in SVW/SQIP-TRAIN and it does improve performance, by 3% on average and by as much as 20% in some programs (*e.g.*, *vortex*). However, on certain programs this strategy turns counter-productive. Benchmarks that consume many checkpoints due to poor branch prediction (*e.g.*, *vpr*) suffer from greedy checkpoint allocation. Benchmarks that need many registers (*e.g.*, *art*) also suffer, because checkpoints hold registers and delay their reclamation.

The relative performance of NAIVE and TRAIN suggests that robust performance requires a serialization mechanism that avoids both squashing *and* greedy checkpoint allocation.

4. DSC and SDR

The previous section shows that using SVW/SQIP effectively in a CPR/CFP context requires a mechanism that gracefully and cheaply handles store-load serialization. We introduce a pair of complementary procedures that accomplish this: DSC (Decoupled Store Completion) and SDR (Silent Deterministic Replay). Traditional CPR/CFP processors manage both registers and loads and stores at checkpoint granularity. The combination of DSC/SDR allows a CPR/CFP processor to continue to manage registers at checkpoint granularity—retaining the associated efficiency benefits and non-blocking support—while at the same time managing loads and stores at the more traditional and familiar instruction granularity. Instruction-granularity handling of stores includes graceful handling of store-load serialization.



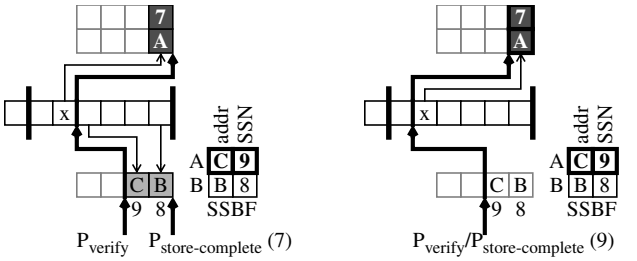
(a) CPR/CFP **(b) CPR/CFP+DSC/SDR**

Figure 4. DSC/SDR overview

4.1 Overview

The key to DSC/SDR is the reassociation of the notion of "commit", *i.e.*, the guarantee that an instruction will not be rolled back. CPR/CFP processors traditionally associate commit with the register-checkpoint pointer, $P_{reg-chkpt}$—the oldest register checkpoint and the oldest point in the program for which register state can be restored. This association is what gives CPR its "bulk commit" property. In DSC/SDR, we shift the notion of commit to the verification pointer, P_{verify}. Figure 4 illustrates this shift and its significance. The figure shows only PCs, not addresses and SSNs, so that we can focus on commit. In Figure 4a, commit is associated with $P_{reg-chkpt}$, and the processor must obey $P_{verify} \geq P_{reg-chkpt} \geq P_{store-complete}$; $P_{reg-chkpt} \geq P_{store-complete}$ is necessary because data cache writes cannot be undone. The fact that $P_{reg-chkpt}$ proceeds at checkpoint granularity forces store completion to trail the oldest checkpoint. In Figure 4b, we associate commit with P_{verify}. Now the processor must only obey $P_{verify} \geq P_{reg-chkpt}$ and $P_{verify} \geq P_{store-complete}$ individually—it does not need to order $P_{reg-chkpt}$ and $P_{store-complete}$ relative to each other. Here, $P_{store-complete}$ (8) has advanced past $P_{reg-chkpt}$ (7). This is essentially DSC (Decoupled Store Completion).

Of course, associating commit with P_{verify} means that "committed" instructions can physically be rolled back. In fact, according to our definition, committed instructions *will* be rolled back whenever load-verification detects a forwarding violation. This is where SDR (Silent Deterministic Replay) comes in. SDR makes the rollback of "committed" instructions transparent to the outside world. SDR relaxes the intuitive invariant that the oldest register checkpoint, $P_{reg-chkpt}$, represents architected register state. With DSC/SDR, $P_{reg-chkpt}$ can represent state *older* than architected register state. SDR incrementally re-constructs architected register state on demand.



(a) Verification "hole" **(b) DSC closes the "hole"**

Figure 5. DSC implements store-load serialization

4.2 DSC (Decoupled Store Completion)

Decoupled Store Completion (DSC) allows stores older than P_{verify} to begin writing to the data cache before the enclosing checkpoint finishes execution. Stores still write to the cache in program order and $P_{store-complete}$ cannot proceed past un-executed branches or unverified loads. Like traditional store completion, DSC is permanent and non-speculative. It does not require a cache that accepts speculative writes [7]. Completed stores never need to be discarded or flushed, and their store buffer entries are reclaimed immediately.

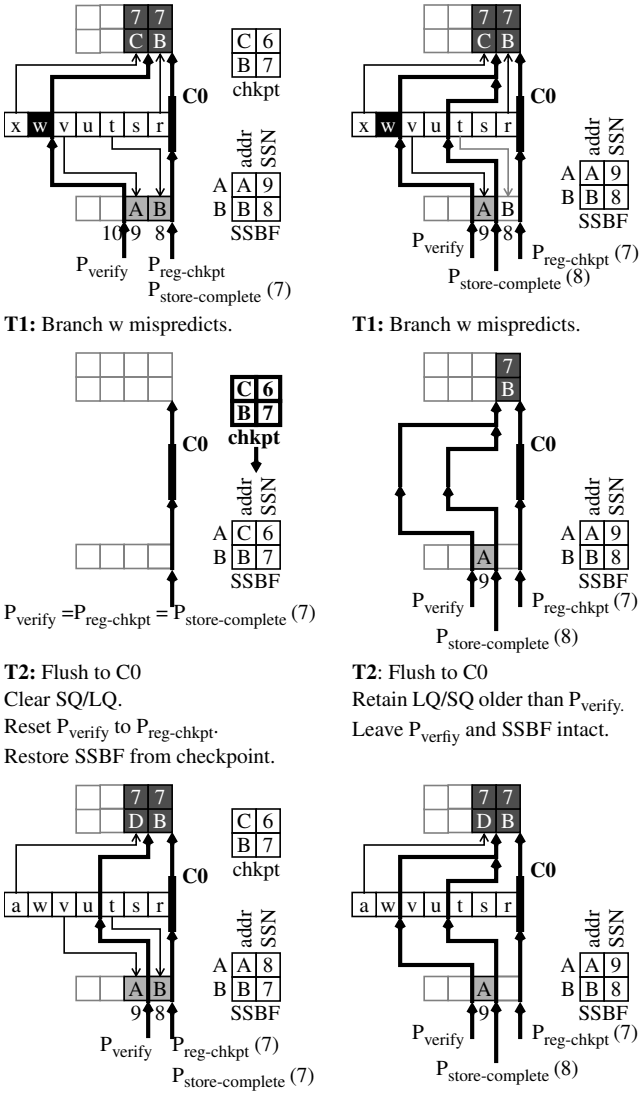
DSC's primary benefit is that it simplifies the implementation and reduces the cost of store-completion/load-execution serialization. DSC allows a store-load pair that resides in the same checkpoint to be serialized—without having to predict the need for serialization *a priori*, without squashing, and without consuming additional checkpoints. Figure 5a repeats the scenario from Figure 3b, but this time adds DSC. Now verification un-blocks as soon as $SSN_{store-complete}$ advances far enough to make the $SSN_{ssbf-set} \leq SSN_{store-complete}$ condition evaluate positively and allow the load to re-execute safely by reading the cache (Figure 5b).

Other Benefits. DSC improves performance by closing SSBF verification holes naturally, without having to squash or to allocate checkpoints greedily. In doing so, it enables the use of smaller, lower-associativity SSBFs and fewer checkpoints. It also reduces store buffer occupancy and improves data cache bandwidth utilization by smoothing store bursts.

4.3 SDR (Silent Deterministic Replay)

In a CPR/CFP processor, mis-speculation on an un-checkpointed instruction involves squashing the window past the mis-speculated instruction to the immediately older checkpoint. Instructions older than the mis-speculation are re-fetched and re-executed despite having correctly executed the first time. This "checkpointing overhead" [1] is illustrated in Figure 6a. Branch w mis-predicts (time T1, top). Recovery involves squashing to checkpoint C0, clearing the load and store queues, restoring the SSBF from a checkpoint, and discarding instructions r–w (T2, middle). Instructions r–w are subsequently re-fetched and re-executed (T3, bottom).

Associating commit with P_{verify} means that on an un-checkpointed mis-speculation, some "committed" instructions may have to be squashed and re-executed. Re-executing committed instructions is fine as long as it is done *deterministically* (*i.e.*, instructions produce the same values during re-execution as they did during initial execution) and *silently* (*i.e.*, the outward appearance is that execution only takes place once). The seemingly difficult part about this is replaying committed loads in the presence of DSC. DSC stores may have overwritten these loads in the cache causing write-after-read hazard during re-execution. Figure 6b shows this scenario. Again, branch w mis-predicts (T1, top). And again, load r has to be squashed and re-executed. However, younger store t has already over-written r in the data cache—both reference address B.



T1: Branch w mispredicts.

T1: Branch w mispredicts.

T2: Flush to C0
Clear SQ/LQ.
Reset P_{verify} to $P_{reg-chkpt}$.
Restore SSBF from checkpoint.

T2: Flush to C0
Retain LQ/SQ older than P_{verify}.
Leave P_{verfiy} and SSBF intact.

T3: re-execute all instructions older than P_{verify}.

T3: don't re-execute loads and stores older than P_{verify}.
Replay loads (t) from LQ.

(a) Traditional recovery **(b) SDR recovery**

Figure 6. SDR recovery in the presence of DSC

SDR overcomes this potential problem by exploiting the observation that loads that may be vulnerable to write-after-read hazards from DSC stores are all older than P_{verify}—the load queue values for these loads are known to be correct. SDR uses the portion of the load queue that is older than P_{verify} as a log and silently replays committed loads by copying the corresponding values from the load queue to newly allocated registers.

Unlike conventional recovery, SDR recovery (T2, middle) leaves P_{verify}, the SSBF, and the load and store queues untouched. Instructions r–w are re-fetched as usual (T3, bottom). However, load r is silently replayed from the load queue, side-stepping the write-after-read hazard with store t. Stores t and v are discarded at rename: v is already in the store queue and t is already in the data cache.

Branch	48 Kbyte 3-table PPM direction predictor. 2K-entry, 4-way set-associative BTB. 32-entry RAS
Pipeline	4-way superscalar with 17 stages: 3 fetch, 2 decode, 1 rename, 1 dispatch, 1 issue, 2 regread, 1 execute, 1 regwrite, 1 SVW, 3 replay, 1 commit. 13 cycle minimum branch misprediction penalty.
Execution	192/192 integer/FP physical registers, 32/32 integer/FP issue queue entries. 4-way issue with up to 4 integer, 2 FP, 2 loads, 1 store, and 1 branch per cycle. 3 Kbyte, 3-table PPM distance-based memory-dependence predictor.
Window	8 checkpoints. 512-entry slice buffer.
Memory	32 Kbyte, 8-way set-associative, 64 byte line, 3-cycle access instruction and data caches, with 8-entry victim buffers. 2 Mbyte, 16-way set-associative, 64-byte line, 10-cycle access L2 with 8-entry victim buffer. 8 8-entry stream buffers. 400 cycle memory latency to the first 16 bytes, 4 cycles to each additional 16 bytes. 128 outstanding misses.

	Load Queue	**Store Queue**
ASSOC	64-entry, fully-associative	48-entry fully-associative (3 cycle)
HLQ/HSQ	ASSOC + 512-entry, 8-way set-associative	ASSOC + 256-entry, fully-associative/1K-entry MTB (10 cycle)
HLQ/SRL	ASSOC + 512-entry, 8-way set-associative	ASSOC + 256-entry indexed/1K-entry LCF (3 cycle)
SVW/SQIP	512-entry indexed. 512-entry 4-way set-associative SSBF	256-entry indexed + 24-entry fully-associative (3 cycle)

Table 1. Simulated processor configurations

Other Benefits and Non-Benefits. As shown in Figure 6b, SDR removes the need to checkpoint and recover the SSBF itself—because loads older than P_{verify} are replayed deterministically they do not have to be re-verified.

DSC allows load queue entries to be reclaimed aggressively but SDR does *not* allow load queue entries to be similarly reclaimed. Load queue entries are not reclaimed until the checkpoint is fully verified because they may be needed to supply register values during SDR reocvery.

4.4 Multiprocessor Issues

We have already argued for the multiprocessor safety of SVW/SQIP (Section 2). The addition of DSC/SDR does not destroy this property. The simplest way to see this is using bi-simulation with a ROB processor. Consider the effect of an invalidation to the cache line of an "exposed" load. If the CPR processor has verified the load, then the ROB processor has committed it. The correct value for this load is the value it has at CPR verification (*i.e.*, the value the ROB processor commits). SDR guarantees this: because the load has been verified, it will always produce the same output, without any externally visible effects. If the load has not been verified when the invalidation arrives, then the conflict will be detected at verification and the load will be forced to re-execute.

DSC/SDR preserve coherence. They also simplify the implementation of different consistency models on CPR. For instance, they enable a simple, non-speculative implementation of SC. Without DSC/SDR, enforcing SC non-speculatively on CPR requires acquiring the permissions to all the stores in a checkpoint—and simultaneously tracking external conflicts with all loads—before committing that checkpoint and draining the stores to the cache. With DSC/SDR, SC is enforced in the same way as it is on a ROB-based processor with SVW, *i.e.*, by not allowing P_{verify} to advance past a load if it is ahead of $P_{store-complete}$. For weaker memory models, DSC/SDR allows a CPR processor to correctly implement fence instructions without creating checkpoints for them.

With or without DSC/SDR, CPR can be made to enforce SC more efficiently using speculation [2, 5, 27]. By presenting the memory system with the same instruction-granularity load/store interface of a traditional ROB processor, DSC/SDR allow consistency model optimizations developed in the ROB context [26] to be migrated seamlessly to the CPR context.

5. Evaluation

This paper primarily addresses single-thread performance and efficiency issues, so our evaluation uses the SPEC2000 benchmarks. The benchmarks are compiled for the Alpha AXP ISA at optimization level -O4. They execute to completion on their training inputs with 2% periodic sampling with 4 million instructions of warmup and 1 million instructions sampled per period. Our infrastructure cannot execute benchmarks *fma3d* and *sixtrack*.

Our timing simulator models CPR/CFP processors with non-blocking caches, stream-buffer prefetching, and realistic bus interconnect. It models both associative and non-associative, flat and hierarchical load and store queues. Table 1 describes our configurations in detail.

Distance-Based PPM-style Forwarding Predictor. Our use of SQIP mandates a forwarding predictor that delivers high accuracies at large window sizes. We use a distance-based forwarding predictor that resembles a tagged PPM (prediction by partial matching) branch predictor [10]. It consists of 3 tables each of which is indexed by a hash of the PC and an increasingly longer global path history postfix. A load uses the prediction of the matching entry from the table indexed by the longest history.

Both HSQ and SRL use a small, fully-associative queue in conjunction with a larger queue. SQIP/PPM can *optionally* use a small associative store queue to reduce the burden on the forwarding predictor by handling the common forwarding cases without prediction. The associative store queue holds the youngest stores in the window and is accessed in parallel with the large indexed store queue. Our default configuration uses a 24-entry associative helper store queue.

5.1 Comparative Performance

Figure 7 shows percent speedup over a baseline CPR/CFP processor with the ASSOC data memory configuration (fully-associative 64/48-entry load/store queues). This baseline performs like a ROB machine—the small load and store queues restrict the window—yet factors out the characteristics of the checkpoint substrate, allowing us to compare the data memory systems.

We show four designs—HLQ/HSQ [1] and HLQ/SRL [6] with 8-way set-associative HLQs, SVW/SQIP-TRAIN (Section 3), and SVW/SQIP+DSC/SDR. All configurations use the same 3 Kbyte PPM store-load dependence predictor: HSQ and SRL use it for load scheduling only, SQIP uses it for unified scheduling and forwarding. Using smaller scheduling predictors for HSQ and SRL reduces performance by up to 9% (1–2% on average). The main result of this experiment is that **the addition of DSC/SDR boosts the average performance of SVW/SQIP by 3% on SpecFP and by 5% on SpecINT. This boost allows SVW/SQIP to match or exceed the performance of either previous design on all but two benchmarks.** On SpecFP, SVW/SQIP+DSC/SDR produces speedups of 24% while HLQ/HSQ and HLQ/SRL only produce

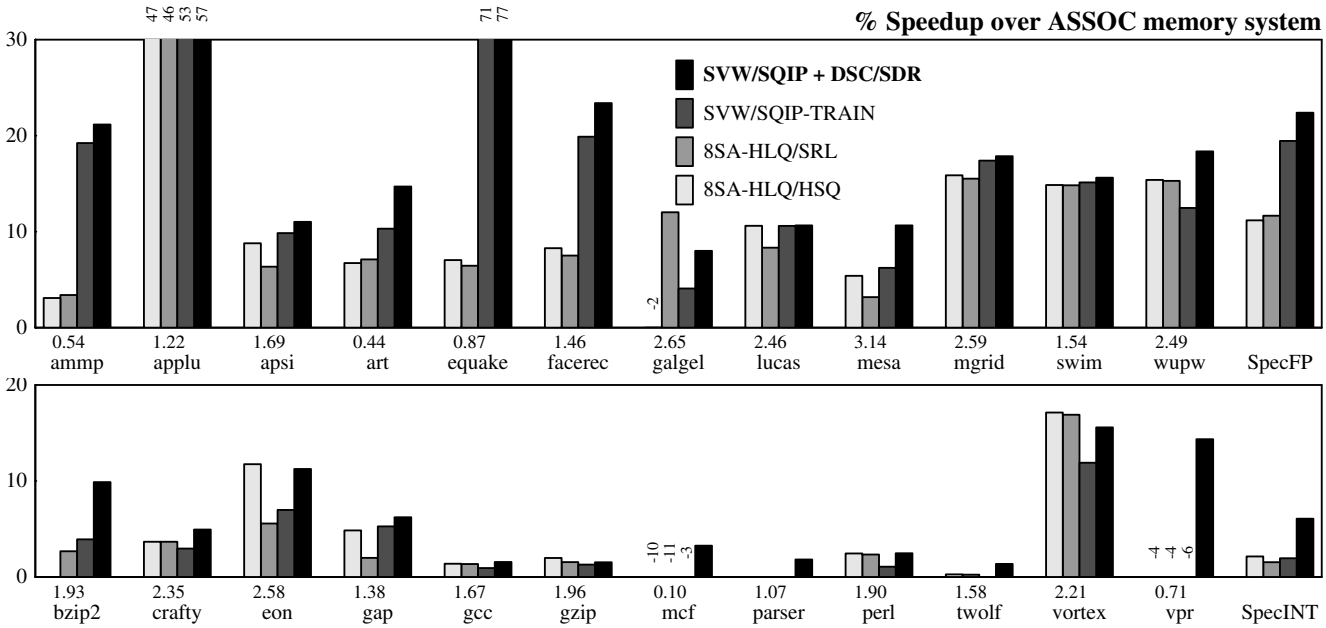


Figure 7. Comparative performance for four data memory system designs

12%. On SpecINT, it produces 6% speedups, compared to 2% for the others.

SVW/SQIP+DSC/SDR under-performs SRL on two programs. *galgel* has suffers from checkpoint overhead due to mis-predicted branches [1]. SRL avoids checkpoint overhead because its large store queue doesn't active during these periods, artificially restricting the window. *vortex* has difficult to predict forwarding patterns.

5.2 Performance Analysis

SVW/SQIP+DSC/SDR differs from previous schemes in three ways: the load queue (SVW+DSC/SDR vs. HLQ), the store queue (SQIP+PPM vs. HSQ vs. SRL), and the secondary benefits of DSC.

Load Queue Effects. The largest performance effects are associated with the load queue. The top graph in Figure 8 shows the performance of different load queue designs. To factor out store queue effects, we couple all load queues with an idealized, flat, 256-entry, 3-cycle associative store queue. We cannot couple all load queues with SQIP because only an SVW load queue can verify SQIP. From left, the bars correspond to 1K-entry (4-way and 8-way) set-associative HLQs, a 512-entry fully-associative HLQ, a 512-entry SVW load queue, and the same enhanced with DSC/SDR.

The performance differences between the set-associative and fully-associative load queues illustrate the large penalty of set-conflicts, which are typically resolved by stalling a load attempting to "enter" a full set. Stalling a load can reduce MLP and degrade performance. The effect can be direct—the load or a dependent misses the last-level cache—or indirect—the load holds resources which prevent other loads from entering the window. With an 8-way set-associative HLQ, *equake* has an MLP of 2.4, with 1.5 loads stalling due to set-conflicts under each miss, and a speedup of 16%. With a fully-associative load queue, it has MLP of 6.8 and a speedup of 78%.

Replacing fully-associative search with SVW reduces performance, on average by about 3%, but sometimes more significantly (*e.g.*, *vpr*). One effect is that SVW delays the detection of load violations from store execution to load verification. When memory dependence prediction accuracy is (relatively) low, these delays can become significant. Mis-speculations that occur under a last-level cache miss and won't be resolved until the miss returns can be especially harmful. A second effect is that without DSC, SVW must squash and restart entire checkpoints to resolve verification "holes". The primary result here are that **with these disadvantages, SVW provides performance that is competitive with realistic set-associative HLQs. With DSC/SDR, SVW actually out-performs fully-associative load queues.**

Store Queue Effects. The bottom graph of Figure 8 uses a 512-entry SVW+DSC/SDR load queue to compare the performance of different 256-entry store queues—HSQ, SRL, and SQIP/PPM both without and with a 24-entry fully-associative helper. We also compare with two idealized configurations—a 3-cycle HSQ and a 3-cycle ORACLE store queue. ORACLE outperforms HSQ because it has perfect load scheduling.

HSQ provides high forwarding accuracy as loads can associatively search the entire store queue. Loads that forward from the second-level store queue (or even loads that alias in the MTB) do so at a higher latency. However, the out-of-order window can easily hide this latency as evidenced by the fact that the 10-cycle HSQ generally has performance close to the (idealized) 3-cycle HSQ.

SRL avoids searching a large secondary queue. However, it expands the store queue only under a last-level cache miss whereas programs can still benefit from a large store queue even in the absence of last-level cache misses. SRL restricts the performance of programs (*e.g.*, *mesa* and *gap*) that benefit from larger windows by virtue of having many data cache misses—store or load—or otherwise low ILP. SRL was proposed as a low-complexity alternative to HSQ, but the complexity of SQIP/PPM is even lower. SQIP/PPM doesn't require accessing another structure in series with the large indexed store queue, whereas SRL requires serial Bloom Filter-store queue access. If store queue size is limited by access latency (*i.e.*, data cache latency), SQIP may support a larger store queue than SRL. Our simulations do not penalize SRL for Bloom-filter access latency.

The main result here is that **with the small helper associative store queue, SQIP/PPM matches the performance of the more complex SRL design, and only slightly under-performs the significantly larger HSQ**. The associative helper store queue makes

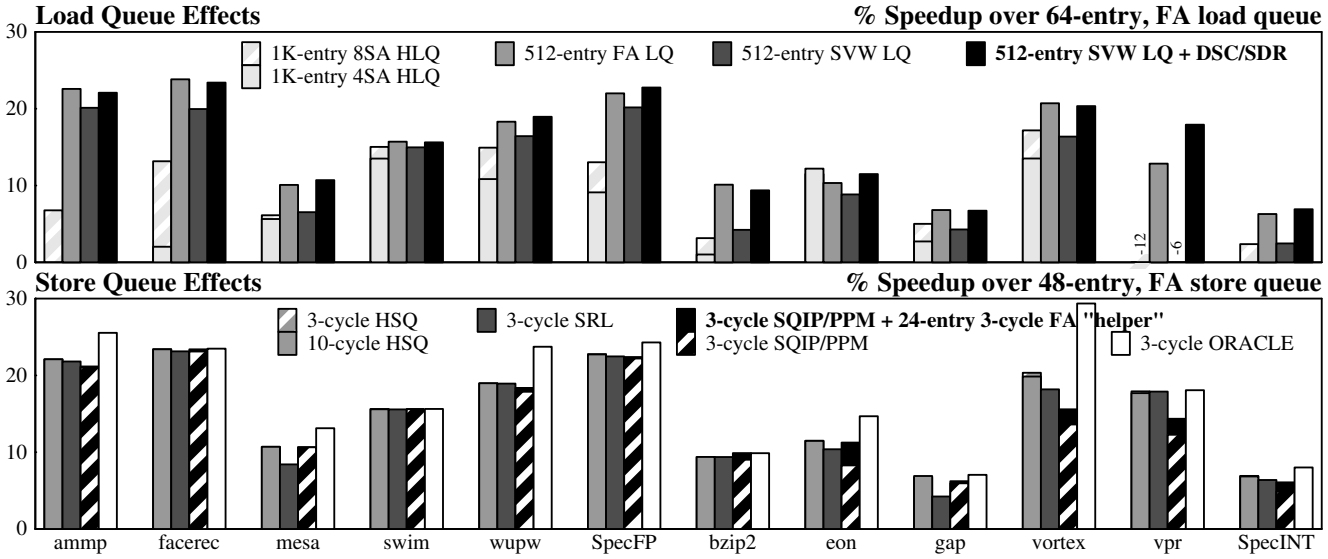


Figure 8. Load and store queue Effects

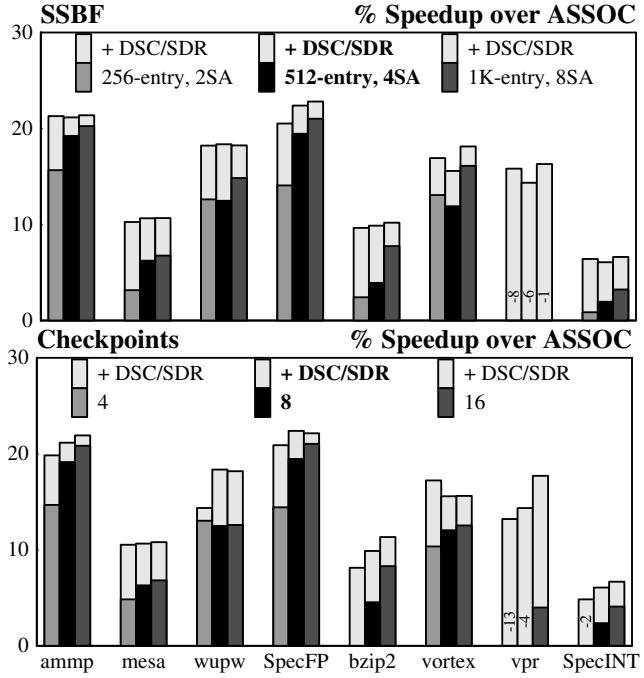


Figure 9. DSC benefit sensitivity analysis

a large difference, especially for SpecINT. We experimented with larger helper queues but the slight increase in performance does not justify the added area. SQIP/PPM performs within 3% of an ORACLE store queue because forwarding prediction accuracy exceeds 99.9% for all but two benchmarks.

DSC Sensitivity Analysis. DSC enables SVW/SQIP to work efficiently in a checkpoint substrate by streamlining instances of store-load serialization. This allows the use of small SSBFs and fewer checkpoints. Figure 9 explores the sensitivity of SVW/SQIP—both with and without DSC/SDR—to SSBF configuration (top) and checkpoint count (bottom).

Smaller, lower-associativity SSBFs produce more "holes". Without DSC, SVW/SQIP is quite sensitive to SSBF associativity. Our default SSBF has 512 entries and is 4-way set-associative, a configuration that produces a low re-execution rate, but does suffer from holes. DSC makes SVW/SQIP virtually insensitive to SSBF configuration. In fact, with DSC higher-associativity SSBFs can actually reduce performance. With DSC, there is no need to force periodic checkpoints in order to serialize store-load pairs allowing checkpoints to grow large. Mis-speculations within larger checkpoints can squash larger numbers of instructions.

With 8 register checkpoints, DSC provides significant benefit. Its benefit decreases as the number of checkpoints grows and individual checkpoints become cheaper. However, CPR/CFP register checkpoints are expensive—in addition to the map table cost [19], there is also the cost of reference counting hardware [18]. CPR/CFP checkpoints hold physical registers. If physical registers are a critical resource, additional checkpoints may cause dispatch delays (*e.g.*, *vortex*). A significant benefit of DSC/SDR is that they enable good performance with fewer checkpoints.

5.3 Performance-Area Trade-Offs

We use CACTI-4 [25] to estimate the area, in 45nm technology, of different memory system components and to conduct a performance-area analysis. Table 2 lists the area (in mm^2) of different memory components: load queues, store queues, as well as the ancillary structures SSBF, PPM (memory dependence predictor), MTB, and LCF (HSQ and SRL Bloom filters). SRL's 256-entry 4-way set-associative forwarding cache (0.069 mm^2) and SVW/SQIP-TRAIN's load-checkpoint table (0.017 mm^2) are not shown The table shows some intuitive relationships, *e.g.*, fully-associative structures are larger than set-associative and direct-mapped structures with the same number of entries. Other relationships deserve additional explanation. For instance, an SVW-style load queue is over twice as large as a set-associative load queue with the same number of entries. There are two reasons for this. First, each SVW load queue entry is bigger because it contains a full address *plus* value *and* SSN. Second, a conventional load queue only needs to allow two writes (load execution) and one read (store execution) per cycle, but an SVW load queue supports two writes (load execution) and two reads (load verification). A second read port can be avoided by exploiting the fact that in-order verification allows the load queue to be interleaved, but there is still

	Entries	64	128	256	512	1K
Load Queue	FA	0.092	0.187	0.302	0.625	1.325
	4SA	0.021	0.026	0.038	0.059	0.104
	8SA	0.026	0.034	0.044	0.064	0.107
	SVW	0.033	0.047	0.099	**0.138**	0.244

	Entries	24	48	128	256	512
Store Queue	FA	**0.093**	0.152	0.355	0.546	1.145
	SQIP	0.028	0.035	0.066	**0.102**	0.189

	Entries	64	128	256	512	1K
SSBF	4SA	0.024	0.030	0.043	**0.067**	0.121
SSBF+Chk	4SA	0.044	0.055	0.079	0.121	0.223
SSBF-MT	DM	0.016	**0.022**	0.036	0.061	0.110

	Entries	256	512	1K	1.5K	2K
PPM	4SA	0.012	0.020	0.028	**0.036**	0.044
MTB	DM	0.009	0.014	0.024	0.036	0.043
LCF	DM	0.012	0.021	0.036	0.046	0.065

Table 2. Data memory component area (mm^2).

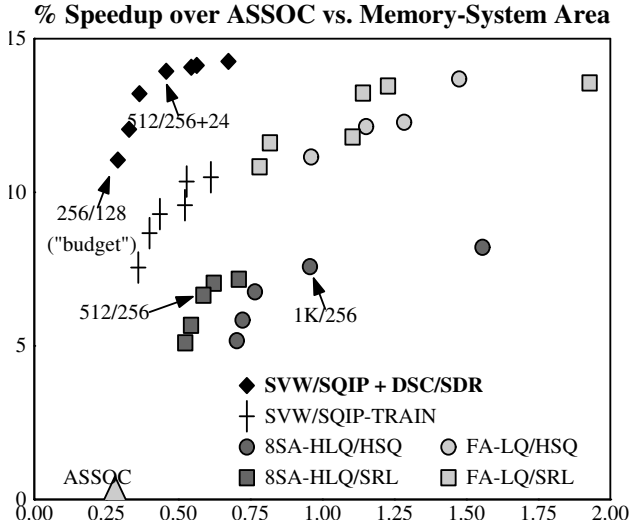


Figure 10. Performace-area trade-offs.

some area overhead. Fully-associative store queues are also larger than fully-associative load queues with the same number of entries. Each entry is larger having to hold both an address and a value, the store queue has two associative ports (for two loads executed per cycle) whereas the load queue has only one (for one store executed per cycle), and the store queue needs an additional read port in order to write stores to the cache. The area for a design is computed as the sum of the areas of its components. For instance, our proposed SVW/SQIP+DSC/SDR design includes the components in bold and occupies approximately 0.46 mm^2. For comparison, one Core2 processor occupies about 25 mm^2 in 45nm process.

Figure 10 plots the geometric mean speedup (for all programs) over the baseline ASSOC memory system against memory system area. There are seven different kinds of marks on the graph, each for a different class of design. Within each class, there is a curve that proceeds from low-area, low-performance configurations on the lower left, to high-area, high-performance ones on the upper right. Better designs are found on the upper left (*i.e.*, high-performance, low-area) part of the graph. We experimented with many configurations for each design class, but show only configurations that would lay on this curve—a configuration is not shown if another in its class gives better performance with lower area. The baseline ASSOC design (triangle) occupies 0.28 mm^2. We experimented with smaller queues, but saw sharp slowdowns.

As expected, the SVW/SQIP+DSC/SDR curve (black diamonds) occupies the upper left portion the graph. **With DSC/SDR, SVW/SQIP meets and even exceeds the performance of the most aggressive hierarchical designs, and does so using less area.** The hierarchical designs that achieve similar performance use fully-associative load queues (light gray squares and circles, respectively)—and these use significantly more area. SRL designs with set-associative load queues have area budgets close to those of SVW/SQIP, but also provide less performance.

One of the SVW/SQIP+DSC/SDR design points has nearly the same area (0.29 mm^2) as the baseline ASSOC system and outperforms it by 11%. This "budget" SVW/SQIP design point drops the 24-entry helper store queue. While it has the overhead for an SSBF and its 256-entry load queue occupies slightly more area than the baseline's 64 associative load queue, its 128-entry indexed store queue occupies significantly *less* area than the baseline's 48 entry associative store queue. The budget design illustrates the key to SVW/SQIP's superior performance-area tradeoff. Most of the performance advantage comes from replacing the set-associative load queue with a flat SVW load queue—plus DSC/SDR. In fact, some area is sacrificed in this transition. However, this area is easily recouped because SVW enables SQIP. While replacing a hierarchical store queue with a SQIP store queue does not gain much performance—in fact a little performance may be lost—the area gains are significant. An SVW load queue has a large performance advantage and a small area overhead. A SQIP store queue has a large area advantage and a small performance overhead. SVW and SQIP offset each other's overheads such that combination has large performance *and* area advantages.

6. Other Related Work

Un-ordered Load and Store Queues. Un-ordered late-binding load-store queues [20] scale by excluding un-executed loads and stores. Late-binding queues can also be address-banked to scale both capacity and bandwidth. Late-binding queues are a poor fit for CPR/CFP. CPR/CFP requires a large window primarily under a last-level cache miss when most store and load addresses are known and require queue entries. At the same time, CPR/CFP does not take advantage of the bandwidth scaling properties of late-binding queues.

Latency-Tolerant Processors. D-KIP [14] is a non-blocking design that uses a ROB and architectural register file multi-checkpointing. SVW/SQIP can work with D-KIP as well. D-KIP doesn't need DSC/SDR to implement store-load serialization because the out-of-order core operates at instruction granularity. DSC/SDR can still be used to simplify and streamline multi-processor operation. DSC/SDR can also simplify the multi-processor operations of latency-tolerant in-order processors like iCFP [8]. DSC/SDR are centralized schemes but can be adapted for use in distributed checkpoint-based architectures [13].

Early Retirement. Checkpoints created post-execution can be used to implement a form of "early retirement" that supports aggressive resource reclamation [9] and (to a lesser degree) load latency tolerance [4]. This style of architecture does not require scalable load and store queues.

Speculative Retirement. DSC/SDR are related to, but different from, speculative retirement [7, 15]. The latter speculatively drains instructions from the window in the shadow of store misses. DSC non-speculatively drains stores to the cache in order to efficiently implement store-load serialization in a CPR/CFP processor.

Deterministic Replay using Load Values. Bugnet uses checkpointing and determinstic replay from load value logs at the architecture level to reconstruct execution traces for multi-threaded program debugging [12].

7. Conclusions

CPR/CFP processors require in-flight memory-systems that track large numbers of loads and stores efficiently. SVW and SQIP support scalable load and store queues, respectively. However, they don't work efficiently in a CPR/CFP processor, largely because they rely on the ability to complete selected stores before the enclosing checkpoint finishes execution.

We present two complementary mechanisms that implement instruction-granularity store completion in a CPR/CFP processor. DSC (Decoupled Store Completion) allows stores to write to the cache before the enclosing checkpoint finishes execution. SDR (Silent Deterministic Replay) implements checkpoint granularity mis-speculation recovery in the presence of DSC. Together, DSC/SDR make the store interface of a checkpoint processor identical to that of a traditional ROB processor. This simplifies the design, reduces implementation cost, and improves performance. DSC/SDR make SVW/SQIP performance-competitive with previously-proposed designs. Combined with SVW/SQIP's area advantages, this results in significantly improved performance-area efficiency.

8. Acknowledgments

We thank the reviewers for their comments. This work was supported by NSF grant CCF-0541292 and by a grant from the Intel Research Council. Andrew Hilton was partially supported by a fellowship from the University of Pennsylvania Center for Teaching and Learning.

References

[1] H. Akkary, R. Rajwar, and S. Srinivasan. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. In *Proc. 36th Intl. Symp. on Microarchitecture*, pages 423–434, Dec. 2003.

[2] C. Blundell, M. Martin, and T. Wenisch. InvisiFence: Performance-Transparent Memory Ordering in Conventional Multiprocessors. In *Proc. 36th Intl. Symp. on Computer Architecture*, Jun. 2009.

[3] H. Cain and M. Lipasti. Memory Ordering: A Value Based Definition. In *Proc. 31st Intl. Symp. on Computer Architecture*, pages 90–101, Jun. 2004.

[4] L. Ceze, K. Strauss, J. Tuck, J. Renau, and J. Torrellas. CAVA: Hiding L2 Cache Misses with Checkpoint-Assisted Value Prediction. *IEEE Computer Architecture Letters*, 3(1), Dec. 2004.

[5] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk Enforcement of Sequential Consistency. In *Proc. 34th Intl. Symp. on Computer Architecture*, pages 278–289, Jun. 2007.

[6] A. Gandhi, H. Akkary, R. Rajwar, S. Srinivasan, and K. Lai. Scalable Load and Store Processing in Latency Tolerant Processors. In *Proc. 32nd Intl. Symp. on Computer Architecture*, pages 446–457, Jun. 2005.

[7] C. Gniady, B. Falsafi, and T. Vijaykumar. Is SC + ILP = RC? In *Proc. 26th Intl. Symp. on Computer Architecture*, pages 162–171, May 1999.

[8] A. Hilton, S. Nagarakatte, and A. Roth. iCFP: Tolerating All-Level Cache Misses in In-Order Pipelines. In *Proc. 15th Intl. Symp. on High Performance Computer Architecture*, pages 431–442, Feb. 2009.

[9] J. Martinez, J. Renau, M. Huang, M. Prvulovic, and J. Torrellas. Cherry: Checkpointed Early Resource Recycling in Out-of-Order Microprocessors. In *Proc. 35th Intl. Symp. on Microarchitecture*, Nov. 2002.

[10] P. Michaud. A PPM-like, Tag-Based Branch Predictor. *Journal of Instruction Level Parallelism*, 7(1):1–10, Apr. 2005.

[11] O. Mutlu, J. Stark, C. Wilkerson, and Y. Patt. Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors. In *Proc. 9th Intl. Symp. on High Performance Computer Architecture*, pages 129–140, Feb. 2003.

[12] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. In *Proc. 32nd Intl. Symp. on Computer Architecture*, pages 284–295, Jun. 2005.

[13] M. Pericas, A. Cristal, F. Cazorla, R. Gonzalez, D. Jimenez, and M. Valero. A Flexible Heterogeneous Multi-Core Architecture. In *Proc. 12th Intl. Conf. on Parallel Architectures and Compilation Techniques*, Sep. 2007.

[14] M. Pericas, R. Gonzalez, D. Jimenez, and M. Valero. A Decoupled KILO-Instruction Processor. In *Proc. 12th Intl. Symp. on High Performance Computer Architecture*, pages 53–64, Feb. 2006.

[15] P. Ranganathan, V. Pai, and S. Adve. Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap Between Memory Consistency Models. In *Proc. 7th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 199–210, Oct. 1996.

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

[17] A. Roth. Store Vulnerability Window (SVW): A Filter and Potential Replacement for Load Re-Execution. *Journal of Instruction Level Parallelism*, 8, 2006. (http://www.jilp.org/vol8/).

[18] A. Roth. Physical Register Reference Counting. *Computer Architecture Letters*, 7(1), Jan. 2008.

[19] E. Safi, P. Akl, A. Moshovos, A. Veneris, and A. Arapoyianni. On The Latency, Energy, and Area of Checkpointed, Superscalar Register Alias Tables. In *Proc. 2007 Intl. Symp. on Low-Power Electronics and Design*, Aug. 2007.

[20] S. Sethumadhavan, F. Roesner, J. Emer, D. Burger, and S. Keckler. Late-Binding: Enabling Unordered Load-Store Queues. In *Proc. 34th Int'l Symposium on Computer Architecture*, pages 347–357, Jun. 2007.

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

[22] T. Sha, M. Martin, and A. Roth. NoSQ: Store-Load Communication without a Store Queue. In *Proc. 39th Intl. Symp. on Microarchitecture*, pages 285–296, Dec. 2006.

[23] S. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual Flow Pipelines. In *Proc. 11th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 107–119, Oct. 2004.

[24] S. Subramaniam and G. Loh. Fire and Forget: Load/Store Scheduling with No Store Queue at All. In *Proc. 39th Intl. Symp. on Microarchitecture*, Dec. 2006.

[25] D. Tarjan, S. Thoziyoor, and N. Jouppi. CACTI 4.0. Technical Report HPL-2006-86, Hewlett-Packard Labs Technical Report, Jun. 2006.

[26] C. von Praun, H. Cain, J.-D. Choi, and K. Ryu. Conditional Memory Ordering. In *Proc. 33rd Intl. Symp. on Computer Architecture*, pages 141–152, Jun. 2006.

[27] T. Wenisch, A. Ailmaki, and B. Falsafi. Mechanisms for Store-wait–free Multiprocessors. In *Proc. 34th Intl. Symp. on Computer Architecture*, pages 266–277, Jun. 2007.

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