NoSQ: Store-Load Communication without a Store Queue

Tingting Sha, Milo M. K. Martin, and Amir Roth
Department of Computer and Information Science, University of Pennsylvania
{shatingt, milom, amir} @cis.upenn.edu

Abstract

This paper presents NoSQ (short for No Store Queue), a microarchitecture that performs store-load communication without a store queue and without executing stores in the out-of-order engine. NoSQ implements store-load communication using speculative memory bypassing (SMB), the dynamic short-circuiting of DEF-store-load-USE chains to DEF-USE chains. Whereas previous proposals used SMB as an opportunistic complement to conventional store queue-based forwarding, NoSQ uses SMB as a store queue replacement.

NoSQ relies on two supporting mechanisms. The first is an advanced store-load bypassing predictor that for a given dynamic load can predict whether that load will bypass and the identity of the communicating store. The second is an efficient verification mechanism for both bypassed and non-bpyassed loads using in-order load re-execution with an SMB-aware store vulnerability window (SVW) filter.

The primary benefit of NoSQ is a simple, fast datapath that does not contain store-load forwarding hardware; all loads get their values either from the data cache or from the register file. Experiments show that this simpler design—despite being more speculative—slightly outperforms a conventional store-queue based design on most benchmarks (by 2% on average).

1. Introduction

Conventional dynamically-scheduled processors implement value communication between loads and older in-flight stores using an age-ordered store queue as an intermediary. When a store executes, it writes its value into the store queue at a position determined by its age. When a load executes, it both accesses the cache and associatively searches the store queue for older stores to the same address. On a match, it forwards the value from the youngest matching store; otherwise, it uses the value from the data cache.

One drawback of the conventional approach is the non-scalability of associative search, i.e., search latency grows quickly with structure size. Associative search constrains the scalability of the store queue, which in turn constrains the scalability of the entire instruction window. To address this challenge, recent work has proposed to reduce both search frequency and the number of entries that must be searched [2, 5, 12, 15, 18, 20], to replace the fully-associative age-indexed store queue with a set-associative address-indexed forwarding structure [6, 21, 24], or to maintain the age-ordered structure but replace associative search with speculative indexed access [19, 22]. This paper presents ***NoSQ*** (short for ***No Store Queue*** and pronounced like "mosque"), a microarchitecture that implements in-flight store-load communication without a store queue or any other intermediary structure. By avoiding such structures, the core datapath can be simpler, smaller, and faster.

The NoSQ microarchitecture uses ***speculative memory bypassing (SMB)*** [10, 11, 13, 14, 25] as a replacement for conventional store-queue based forwarding. SMB is a previously-proposed technique that dynamically transforms in-window store-load communication into register communication. It uses an extension to register renaming to short-circuit DEF-store-load-USE chains into lower-latency DEF-USE chains. Prior proposals used SMB opportunistically as a lower-latency complement to conventional store queue-based forwarding.

This paper proposes using SMB for *all* in-window store-load forwarding. A decode-stage predictor classifies loads as either bypassing or non-bypassing. All bypassing loads—loads that in a conventional design forward from an older in-flight store—skip out-of-order execution altogether. Instructions dependent on bypassed loads use SMB's renaming extension to obtain their values directly from the register file. Non-bypassing loads are injected into the out-of-order core as usual, execute when their register inputs are ready, and obtain their value from the data cache. Because NoSQ uses SMB for all in-window communication, there is no reason to maintain a store queue (or other analogous structure) to act as a forwarding intermediary, or to execute stores out-of-order to update this structure. Instead, store execution occurs in the in-order back-end commit pipeline.

NoSQ relies on two supporting mechanisms. The first is a ***store-load bypassing predictor*** that can predict both whether a given load will bypass and the identity of the bypassing store. Speculative memory bypassing, which short-circuits loads and stores without comparing their addresses, requires more robust prediction than speculative forwarding, which performs an initial address check [19]. This paper presents a bypassing predictor design that achieves prediction accuracies of greater than 99.8% on all SPEC2000 and MediaBench programs, using only 10KB of storage to do so. The predictor represents store-load dependences as the dynamic distances (in stores) from the store to the load, and is explicitly path-sensitive.

The second component is a lightweight mechanism for verifying the correct execution of both bypassed and non-bypassed loads while preserving SMB's datapath simplification benefits. NoSQ verifies all loads using in-order load re-execution [3] combined with a store vulnerability window (SVW) filtering mechanism [16, 17] extended to support SMB. With SVW, most bypassed loads skip re-execution (and the corresponding cache access) and commit without having accessed the cache even once.

Without a conventional store queue, NoSQ requires mechanisms to supply store addresses and data to the commit pipeline. NoSQ extends the commit pipeline with additional stages for reading store base addresses and data values from the register file and for calculating effective addresses. The register ports and address generation units that previously performed these functions in the out-of-order core are simply shifted to the back-end. NoSQ also uses these stages, ports, and adders to calculate the addresses of bypassed loads for verification

The 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)
0-7695-2732-9/06 $20.00 © 2006 IEEE

purposes. Adding a little more bandwidth to these stages—one additional register read port and one additional address generation unit—allows them to re-calculate the addresses of the remaining, non-bypassing loads. By calculating all load addresses in the back-end pipeline, the traditional load queue itself can be eliminated as well.

The primary contributions of this paper are:

- A design for a simple core datapath that does not include a store queue or other store-load forwarding structure. All loads obtain their values either from the cache or the register file. Stores and SMB loads are not dispatched to the out-of-order core. An extended commit pipeline calculates store and SMB-load addresses, commits stores to the data cache, and re-executes a small fraction of the loads to verify that every load receives the correct value.
- A new implementation of speculative memory bypassing (SMB) that is capable of handling all in-flight memory communication duties. This implementation includes a distance-based store-load bypassing predictor that achieves greater than 99.8% accuracy using only 10KB of storage, an integrated verification/training mechanism that allows most bypassed loads to skip cache access completely, and support for partial word bypassing.

Being more speculative and with a longer commit pipeline, NoSQ is not intended to outperform (in terms of IPC) a conventional design with an integrated store queue. Its goal is to match conventional performance while removing timing-critical and non-scalable structures from the processor's out-of-order engine. Nonetheless, experiments show that the lower-latency memory communication provided by speculative memory bypassing and the reduced contention for out-of-order resources more than offset the performance impact of the infrequent store-load bypassing mis-predictions. As a result, on average NoSQ outperforms an idealized conventional out-of-order superscalar design by 2%.

2. Background

This section reviews two previously-proposed techniques upon which NoSQ builds: speculative memory bypassing (SMB) [10, 11, 13, 14, 25] and in-order load re-execution [3, 7] with store vulnerability window (SVW) filtering [16, 17].

Throughout the paper, we identify dynamic stores using ***store sequence numbers (SSNs)*** [16], which form the basis of the SVW scheme and are more convenient than store queue indices because they also represent committed stores. All dynamic stores are assigned monotonically increasing SSNs at rename. A global counter, SSN_{rename}, tracks the SSN of the most recently renamed/dispatched store. A second counter, SSN_{commit}, tracks the SSN of the most recently committed store. $SSN_{rename} - SSN_{commit}$ is equal to the occupancy of the store queue. SSNs are easily convertible to store queue indices: the store queue index of an in-flight store is the low-order bits of the store's SSN. In the rare situations in which SSNs wrap around, the processor drains its pipeline and clears all hardware structures that hold SSNs.

2.1. Speculative Memory Bypassing (SMB)

Value communication through memory from instruction DEF to instruction USE takes place through a store-load pair, DEF-store-load-USE. Speculative memory bypassing (SMB) [10, 11, 13, 14, 25] optimizes in-window memory communication. SMB "short-circuits" the store-load pair in a DEF-store-load-USE chain by directly connecting the DEF to the USE using the register map table. Initial proposals used SMB only for its store-load communication latency reduction benefits; bypassed loads still executed in the out-of-order engine for verification [10, 11, 25]. Subsequent proposals used SMB to amplify execution core bandwidth and capacity as well, by allowing bypassed loads to skip the out-of-order engine and using in-order load re-execution for verification [13, 14].

A Store-Sets based SMB design. Table 1 shows the pipeline action diagram for an SMB implementation that leverages a modified StoreSets store-load dependence predictor [4] and that performs SMB verification by executing bypassed loads out-of-order. SMB-specific modifications are in bold. Originally designed for load scheduling, StoreSets is a two stage predictor. A decode-stage table called the StoreSet ID Table (SSIT) maps load PCs to communicating store PCs. A rename-stage table called the Last Fetched Store Table (LFST) maps each store PC to the SSN of its most recent dynamic instance[1]. This SMB implementation extends StoreSets by (i) adding to each SSIT entry an additional confidence counter that tracks the stability of the communicating store-load pair and (ii) extending the LFST to track not only the SSN of the most recent dynamic instance of each store PC, but also its input data physical register tag (*dtag*).

Collapsing a DEF-store-load-USE chain to a DEF-USE chain is a multi-step process. The first connection (DEF-store) is established when the store is renamed: DEF's output location which is also the store's data input physical register (*st.dtag*) is noted in the store's LFST entry. The second connection (DEF-load) takes place when the load is decoded and renamed. At decode, the load uses its own PC to pick up the PC of the communicating store from the SSIT. At rename it uses the forwarding store PC to pick up that store's data input physical register tag (*dtag*) from the LFST. To perform the actual "short-circuiting" operation, the load's output logical register is mapped to *dtag* rather than to a newly allocated physical. The third and final step takes place naturally when USE is renamed: conventional RAT (register alias table) lookup points USE's input to the load's output which is actually DEF's output (*dtag*). Effectively, the store passes *dtag* from DEF to load via the LFST, which then passes it to USE via the RAT.

In this example, the processor verifies SMB by executing the load itself in the out-of-order engine. When the load executes, it compares its value to the value in the short-circuited register. If the values do not match, recovery is initiated. The load also writes its address into the load queue, to allow its address to be checked by older stores that have yet to execute.

---

[1] As proposed, StoreSets uses the SSIT to map store and load PCs to Store Set IDs, and the LFST to map StoreSet IDs to store queue indices. This modified scheme achieves the same effect.

0-7695-2732-9/06 $20.00 © 2006 IEEE

	DECODE	RENAME	WAIT	EXECUTE

Store		st.dtag=RAT[st.dreg]
		st.atag=RAT[st.areg]
		LFST[st.PC].SSN=SSN_{rename}++
		LFST[st.PC].dtag=st.dtag
			st.dtag
			st.atag
				SQ[st.SSN].addr=RF[st.atag]+st.ofs
				SQ[st.SSN].data=RF[st.dtag]
				search LQ for older loads
				flush on ld.addr match

Load	$ld.PC_{fwd}$=SSIT[ld.PC].PC
	$ld.conf_{fwd}$=SSIT[ld.PC].conf
		ld.atag=RAT[ld.areg]
		$ld.SSN_{fwd}$=LFST[$ld.PC_{fwd}$].SSN
			ld.atag
			$ld.SSN_{fwd}$
				ld.addr=RF[ld.atag]+ld.ofs
				LQ[ld.lqpos]=ld.addr

		High-confidence forwarding? Speculative memory bypassing		
		$ld.PC_{fwd}$ & $ld.conf_{fwd}$ & $ld.SSN_{fwd}$ > SSN_{commit}? **RAT[ld.dreg]=ld.dtag=LFST[$ld.PC_{fwd}$].dtag**		read cache, search SQ for younger stores **flush if data != RF[ld.dtag]**
		Low-confidence? Conventional load		
		$!ld.PC_{fwd} \mid !ld.conf_{fwd}$? RAT[ld.dreg]=ld.dtag		read cache, search SQ for younger stores RF[ld.dtag] = data

Table 1. In-order decode/rename and out-of-order wait/execute pipeline action diagram for Store-Sets based SMB. Verification is performed by executing the bypassed load in the out-of-order engine.

2.2. Filtered In-order Load Re-execution

Conventional dynamically scheduled processors issue loads speculatively in the presence of older un-executed stores. They verify this speculation by buffering load addresses in a load queue. When stores execute, they search the load queue for younger executed loads with matching addresses. Matches signal mis-speculation and trigger recovery.

In-order load re-execution prior to commit. To avoid associative load queue search, load speculation can alternatively be verified by in-order load re-execution prior to commit [3, 7]; mis-speculation is detected when a load's re-executed value does not equal its (initial) executed value. As shown in Table 2, re-execution can be implemented within an existing in-order back-end pipeline; the actions corresponding to re-execution are in bold. The setup stage is extended to read load addresses and data values from the load queue. For the time being, ignore the SVW stage. The data cache stage is augmented to also re-execute speculative loads. The commit stage is extended to compare the newly loaded correct values (*ndata*) of loads marked for re-execution with the original value (*data*) and flush the pipeline on a mismatch.

Store Vulnerability Window (SVW) re-execution filtering. Under conventional load speculation, only loads that actually issued in the presence of older un-executed stores—typically 10–20% of all loads [3, 7]—are speculative, and only these loads must re-execute. Because the re-execution rate is low, load re-execution can share a data cache port with store commit without performance loss due to contention. However, more aggressive forms of load speculation—e.g., speculative indexed forwarding [19]—perform speculation of one kind or another on all loads and would seemingly require re-executing all loads. Re-executing all loads would in turn require an additional data cache port or would otherwise induce overheads that overwhelm the benefit of the speculation itself.

Store Vulnerability Window (SVW) [16] is an address-based filter that dramatically reduces the re-execution rate for any form of speculation on loads with respect to older stores. SVW is based on the observation that a load should not have to re-execute if no store wrote to a matching address in a sufficiently long time. SVW implements this basic idea using a small address-indexed table called the Store Sequence Bloom Filter (SSBF) tracks the SSN of the youngest committed store to write to each (hashed) address. When a load executes, it remembers the SSN of the youngest store to which it is not vulnerable (SSN_{nvul}); if the load forwards SSN_{nvul} is the SSN of the forwarding store, otherwise this is SSN_{commit} at the time of execution. Prior to commit, the load then accesses the SSBF and re-executes only if the last store to write to its address is younger than its SSN_{nvul}. Table 2 shows the SVW actions in the in-order back-end pipeline in bold. These actions are restricted to the new SVW stage which precedes the data cache access (store write, load re-execute) stage.

The original SVW proposal described the SSBF as untagged and direct mapped and achieved re-execution rate reduction factors of 20–50. To reduce re-executions further, the SSBF can be tagged and made set-associative, with each set managed in FIFO fashion. A tagged SSBF (T-SSBF) can reduce re-execution rates by factors of 100–200 with less total storage than its untagged counterpart [17].

	SETUP	SVW	DCACHE	COMMIT

Store	st.addr=SQ[head].addr
	st.data=SQ[head].data
		T-SSBF[st.addr]=st.SSN	D$[st.addr]=st.data	**SSN_{commit}++**
				commit

Load	**ld.addr=LQ[head].addr** **ld.data=LQ[head].data** **$ld.SSN_{nvul}$=LQ[head].SSN_{nvul}**	**ld.reexec &=** **T-SSBF[ld.addr] > $ld.SSN_{nvul}$**	**ld.reexec ?** **ld.ndata = D$[ld.addr]**	**(ld.reexec & ld.ndata != ld.data)** **? flush** **:** commit

Table 2. In-order back-end pipeline action diagram for load re-execution with SVW filtering.

The 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)
0-7695-2732-9/06 $20.00 © 2006 IEEE

	DECODE	RENAME	WAIT	EXECUTE
Store		**ROB[tail].dtag=RAT[st.dreg]** **ROB[tail].atag=RAT[st.areg]** **ROB[tail].ofs=st.ofs** SRQ[SSN$_{rename}$++].dtag = RAT[st.dreg]	**nothing!**	**nothing!**
Load	**ld.dist$_{byp}$=predict[ld.PC].dist** **ld.conf$_{byp}$=predict[ld.PC].conf**	**ROB[tail].SSN$_{nvul}$ = ld.SSN$_{byp}$ = SSN$_{rename}$–ld.dist$_{byp}$** **ROB[tail].atag=RAT[ld.areg]** **ROB[tail].ofs=ld.ofs**		
		High-confidence bypassing? speculative memory bypassing		
		ld.SSN$_{byp}$ > SSN$_{commit}$ & ld.conf$_{byp}$? **RAT[ld.dreg]=ROB[tail].dtag=SRQ[ld.SSN$_{byp}$].dtag**	**nothing!**	**nothing!**
		Non-bypassing (or low confidence bypassing)? simple cache access (with delay)		

		ld.SSNbyp $\leq$ SSNcommit | !ld.conf$_{byp}$?
		RAT[ld.dreg]=ROB[tail].dtag=ld.dtag
		ld.atag=RAT[ld.areg]
			ld.atag
			(SSN$_{byp}$ $\leq$SSN$_{commit}$)
				D$[ld.addr]

Table 3. In-order decode/rename and out-of-order wait/execute pipeline action diagrams. NoSQ does not dispatch stores to the out-of-order execution core, and uses speculative memory bypassing for all in-flight memory communication.

3. The NoSQ Microarchitecture

NoSQ uses speculative memory bypassing (SMB) to replace the conventional store-load forwarding path in an out-of-order processor. Unlike previous proposals that performed opportunistic SMB [10, 11, 13, 14, 25], NoSQ uses exclusive SMB to handle all store-load communication. In NoSQ, loads are classified as either non-bypassing or bypassing. Non-bypassing loads dispatch to the out-of-order engine where they wait for their address register input and read the data cache. Bypassing loads skip out-of-order execution; map-table "short-circuiting" directly connects their consumers to the predicted bypassing stores' producers. As the store queue is no longer used as an intermediary for store-load communication, stores skip the out-of-order engine as well. As in a traditional pipeline, stores write their values to the data cache at commit.

Exclusive SMB requires a highly-accurate store-load bypassing predictor (Section 3.3) and mechanisms for bypassing "difficult" but common cases like partial-word communication (Section 3.5). Eliminating the store queue requires extending the back-end pipeline to execute stores (Section 3.4). Before detailing these modifications, Sections 3.1 and 3.2 describe the basic structure and operation of NoSQ.

3.1. Store Load Dependences as Dynamic Store Distances

SVW uses store sequence numbers (SSNs) to identify both in-flight and committed stores. This naming scheme forms the basis for NoSQ, which represents store-load dependences as ***dynamic store distances*** [26]. Dynamic store distances are both compact and convenient. At rename, a predicted distance to a bypassing store can be easily converted to a dynamic store instance by simple subtraction, ld.SSN$_{byp}$ = SSN$_{rename}$–ld.dist$_{byp}$. At commit, the distance to the store the load should have bypassed from can be computed as ld.dist$_{byp}$ = SSN$_{commit}$–T-SSBF[ld.addr]; the T-SSBF holds the SSNs of the most recently committed store to each (hashed) address.

Store-load dependence schemes based on store distances can be both more efficient and more powerful than schemes based on store PCs. Store distances can be converted to SSNs (i.e., dynamic stores) and vice versa using simple subtraction from global counters. In contrast, store PCs and SSNs can be converted to each other only through a level of indirection. At rename, converting a store PC to an SSN requires a table that maps each (hashed) store PC to its most recent dynamic instance, e.g., a Last Fetched Store Table (LFST) [4] or a Store Address Table (SAT) [19]. For high accuracy, this table must be repaired during branch misprediction recovery. Converting an SSN to a store PC at commit also requires an additional table, e.g., an SPCT [16]. Store distance-based schemes can also easily represent dependences of loads on what is not the most recent instance of a static store, as in the loop body `X[i]=A*X[i-2]`. Store PC-based schemes, which use a table to map each store PC only to its most recent dynamic instance, cannot easily represent these communication patterns that occur frequently in some benchmarks [10, 19, 25].

3.2. Basic Structure and Operation

Table 3 shows the basic operation of NoSQ at the processor's front-end and out-of-order stages. At decode, all loads access the store-load bypassing predictor to obtain a predicted bypassing distance, ld.dist$_{byp}$. Loads that "miss" in the predictor or whose predicted bypassing store has already committed—this is determined at rename by comparing ld.SSN$_{byp}$ to SSN$_{commit}$—are predicted as non-bypassing. These non-bypassing loads are dispatched to the out-of-order engine where they wait for their base address register to become available. On issue, they perform a simple data cache access. Bypassing loads—loads that "hit" in the predictor and whose SSN$_{byp}$ > SSN$_{commit}$—are not dispatched into the out-of-order engine. Instead, their output register mapping is set to the physical register corresponding to the predicted bypassing store's data input. This register is retrieved from the ***store register queue (SRQ)*** using the low-order bits of ld.SSN$_{byp}$. The store register queue parallels a traditional store queue in structure, but unlike a traditional store queue is not a datapath element. It contains only physical register numbers (not addresses and values) and it is accessed only at rename, not at execute.

As Table 3 also shows, NoSQ handles especially difficult bypassing cases—e.g., bypassing from a narrow store to a wide load—by dispatching the load and delaying its issue until the corresponding store commits. The next section provides more details on the implementation of delay.

3.3. Store-Load Bypassing Predictor

The goal of a store-load bypassing predictor is to map each dynamic load to the dynamic in-flight store (if any) from which it *will* forward. Bypassing prediction, especially the kind performed by NoSQ, is more challenging than other forms of store-load dependence prediction. Dependence-predictors used for load scheduling (i.e., to reduce squashes due to premature load execution) [1, 4, 8, 23, 26] must capture only loads that execute out-of-order with respect to older stores on which they depend; they can also represent dependences conservatively because predicting a dependence where none exists only results in a little additional delay. Dependence predictors used for speculative indexed forwarding [19] must capture all in-flight store-load dependences, but can also be conservative. In speculative indexed forwarding, a load forwards from its predicted store only if their addresses match.

In contrast, NoSQ's predictor must be more precise. SMB passes values from a store's data producer (DEF) to the load's consumers (USE) without the benefit of an address check [10, 11, 13, 14, 25] and so conservative prediction of a dependence where none actually exists is unacceptable from a performance standpoint. Because NoSQ uses SMB exclusively rather than opportunistically, its bypassing predictor must generate a dependence prediction for every load; it does not have the option of generating no prediction when its confidence is low. NoSQ's bypassing predictor also replaces/subsumes any store-load dependence predictor in the baseline microarchitecture.

NoSQ's bypassing predictor design builds upon various existing designs [4, 10, 11, 25, 26]. The basic predictor organization is a load-PC indexed, set-associative, cache. Each predictor entry contains a (partial) tag and distance field. When the commit stage detects a bypassing mis-prediction—(i) a non-bypassing load should have bypassed, (ii) a bypassing load should have accessed the cache instead, or (iii) a bypassing load bypassed from the wrong dynamic store—it allocates an entry for that load in the predictor table (if necessary) and updates its distance field.

Explicit path-sensitivity and hybridization. To capture path-dependent bypassing patterns, NoSQ's bypassing predictor uses explicit path information in its indexing function. Like branch predictors with explicit path history in their indexing functions, NoSQ's bypassing predictor uses a hybrid structure to reduce both storage requirements and training times, exploiting the fact that many loads have path-independent bypassing patterns.

NoSQ's bypassing predictor uses two parallel tables: a path-insensitive table indexed by load PC, and a path-sensitive table indexed by a hash (XOR) of load PC and some number of path history bits. To capture both flow-sensitive (i.e., conditional branch) and context-sensitive (i.e., call-site) bypassing patterns, the path history contains both branch directions (1 bit per branch) and call PCs (2 bits per call). Loads access both tables in parallel. If a matching entry is found in both tables, the path-sensitive prediction is used. On a mis-prediction, entries are created in both tables. For loads with path-independent bypassing behavior, this training policy results in the one-time creation of an entry in the path-sensitive table. This entry will eventually be overwritten by a legitimate entry for a load with path-sensitive behavior. For loads with path-dependent bypassing behavior, the steady state contents of the predictor will be an entry in the path-sensitive table for each observed path and an entry in the path-insensitive table that corresponds to the most recent mis-predicted bypassing distance.

Although the NoSQ predictor is explicitly path sensitive, Store-PC based dependence predictors (e.g., StoreSets) do have some measure of implicit path sensitivity. For example, when there is no instance of the predicted static store along the current path, the load would be predicted not to forward; if there are instances of two different predicted static stores, the load is predicted to forward from the younger instance. In contrast, without using explicit path information, a distance-based dependence predictor is totally path-insensitive.

Delay. NoSQ uses SMB for all in-flight store-load communication. However, some store-load communication cannot be handled by bypassing. Specifically, SMB cannot perform partial-store (i.e., narrow-store/wide-load) communication because it cannot combine values from multiple sources. Other communication patterns may pathologically elude the predictor, e.g., path dependent communication patterns whose differentiating signature is longer than the predictor's history or path-independent/data-dependent patterns. To avoid bypassing mis-prediction in these cases, NoSQ effectively converts the would-be bypassing load to a non-bypassing load by dispatching it to the out-of-order engine and delaying it until the uncertain store commits, at which time the load safely retrieves its value from the data cache. A similar approach was used to reduce squashes in speculative indexed forwarding [19].

NoSQ implements delay by attaching a short confidence counter to each predictor entry. A prediction with a sub-threshold confidence results in the load waiting for the store corresponding to SSN_{byp} to commit rather than bypassing from that store. The confidence counters are initialized at an above-threshold value and updated at commit. Counters are decremented if a path-sensitive prediction was available but a bypassing mis-prediction resulted anyway—a condition that captures the three scenarios described above—and incremented otherwise. Because the baseline predictor is path sensitive, the delay prediction is also path sensitive.

3.4. Commit Pipeline and Resulting Core Simplifications

NoSQ's use of SMB to handle all in-flight store-load communication enables several simplifications and enhancements to the out-of-order execution engine. With store addresses and data values not needed in the out-of-order core to support forwarding, the store queue can be eliminated and all stores can skip out-of-order execution. A traditional store queue buffers store addresses and data values not only for store-load forwarding but also for store commit. Eliminating the store queue requires extending the back-end pipeline to perform store address generation and to retrieve store data from the register file. This additional support also enables eliminating the load queue. This section describes these modifications and the modifications to SVW needed to support SMB. Table 4 summarizes these modifications.

	SETUP	REGFILE	SVW	DCACHE	COMMIT
Store	**st.atag=ROB[head].atag** **st.dtag=ROB[head].dtag** **st.ofs=ROB[head].ofs**	**st.baddr=RF[st.atag]**	**st.addr=st.baddr+st.ofs** T-SSBF[st.addr].SSN=st.SSN T-SSBF[st.addr].tag=st.addr	**st.data=RF[st.dtag]** D$[st.addr]=st.data **DTLB[st.addr]**	commit
Load	**ld.atag=ROB[head].atag** **ld.dtag=ROB[head].dtag** **ld.ofs=ROB[head].ofs** **ld.SSN$_{nvul}$=ROB[head].SSN$_{nvul}$**	**ld.baddr=RF[ld.atag]**	**ld.addr=ld.baddr+ld.ofs** **ent=T-SSBF[ld.addr]** **ld. reexec= ld.bypassed** **? ((ent.tag != ld.addr) |** **(ent.SSN != ld.SSN$_{nvul}$))** : ent.SSN > ld.SSN$_{nvul}$	ld.reexec ? ld.ndata=D$[ld.addr] **DTLB[ld.addr]** **ld.data=RF[ld.dtag]**	(ld.reexec & ld.ndata != ld.data) ? flush, train : commit

Table 4. NoSQ in-order back-end pipeline action diagram. NoSQ performs address generation for all stores and loads at the SVW stage and translates addresses for all stores and loads that must re-execute at the data cache stage. Without a load queue or a store queue, load and store base addresses and data values are read from the register file.

SVW for SMB. NoSQ uses SVW to filter re-executions for both bypassed and non-bypassed loads. Both types of loads use the same filter, but they use different filter tests. Non-bypassing loads perform an inequality test, skipping re-execution if ld.SSN$_{nvul}$ $\leq$ T-SSBF[ld.addr]. In contrast, bypassing loads perform an equality test, skipping re-execution if ld.SSN$_{nvul}$=T-SSBF[ld.addr] (recall, SSN$_{nvul}$ is SSN$_{byp}$ for bypassing loads). This equality test actually requires the use of a tagged SSBF (a T-SSBF). Load speculation techniques that use only an inequality test (e.g., speculative scheduling or speculative indexed forwarding) may use an untagged SSBF (because SVW inequality tests are safe with respect to SSBF aliasing). Equality tests, however, are unsafe in the presence of aliasing, necessitating tags.

Using SVW to filter re-executions for bypassed loads actually reduces data cache read bandwidth consumption; because bypassed loads do not read the data cache in the out-of-order core—they are not dispatched to the out-of-order core—and most do not read the data cache for verification.

Eliminating the store queue and the out-of-order execution of all stores. Eliminating the store queue simplifies the latency-critical load execution path in the out-of-order engine. Allowing stores to skip out-of-order execution frees up issue queue entries and issue slots for use by other instructions.

To exploit these advantages, NoSQ effectively moves store execution from the out-of-order core to the in-order back-end. After rename, stores are not injected into the out-of-order core. Instead they are marked as completed and simply wait to commit. In the back-end commit pipeline, prior to the SVW and data cache write stages, stores access the register file to retrieve their data and base address values and perform address calculation. Essentially, instead of using the store queue as an intermediary buffer for their address and data, stores generate these "just in time" prior to their actual use.

Delaying these actions to commit requires buffering store sizes, data and base register tags, and address displacements until commit. These fields can (logically) be stored in the re-order buffer. For Alpha, these fields sum to 34 bits (2 for size, 8 for each register tag, and 16 for displacement). This is "new" storage, but it is written only at rename, it is read only at commit, and it exists outside the latency-critical execution core.

The back-end pipeline uses dedicated register read file ports to obtain store data and base address, an adder for address generation, and a TLB port for address translation. However, these are not additional structures. They are existing structures simply moved from the out-of-order core to the in-order back end. Eliminating the store queue does not require additional register file, address generation, or TLB bandwidth. This design favors a virtually-tagged T-SSBF. A physically tagged T-SSBF would require elongating the pipeline further to allow store addresses to be translated before T-SSBF access. With a virtually-tagged T-SSBF, store address translation can occur in parallel with the initial cycles of store data cache access. A virtually-tagged T-SSBF can be made multiprocessor safe [17].

The obvious performance cost of eliminating the store queue is an extension of the back-end pipeline which may increase pressure on core structures like the store queue and register file. However, this is not a significant concern for NoSQ. NoSQ eliminates the store queue. SMB reduces register file pressure by allowing multiple instructions—specifically, the DEF and the load in a DEF-store-load-USE chain—to share a single physical register[1]; pressure is reduced when the lifetimes of the DEF and the load naturally overlap.

Generating addresses of bypassed loads. In a microarchitecture with SVW-filtered re-execution, load data values are rarely needed in the back-end and so these can obtained from the register file, potentially using the store data register read port. However, load re-execution does require load addresses in the back-end. For non-bypassing loads, these can be obtained from the load queue. In NoSQ, bypassing loads do not execute out-of-order and do not update the load queue, requiring additional mechanisms.

In NoSQ, load input base address and output data register tags and address displacements are recorded in the ROB—these fields also hold the corresponding information for stores—and are available in the back-end. With this information, NoSQ uses the store address register read port and address generation unit to generate the addresses of bypassing loads in the back-end. Because relatively few loads bypass (typically only 10%), a single register read port and a single adder provide sufficient bandwidth to generate the addresses

---

[1] Physical register sharing requires modifications to the register allocation/de-allocation logic. Specifically, the physical registers must be explicitly reference counted to properly determine when it is safe reallocate a register.

for all stores and for bypassing loads. Address translation bandwidth for bypassed loads that must re-execute—i.e., those that “miss” in the T-SSBF—is provided by the store TLB port.

Eliminating the load queue. NoSQ’s extended back-end pipeline and support for transporting load address and data register names to the back-end makes eliminating the traditional load queue a practical option as well. This change requires an additional register read port and address generation unit in the back-end to provide sufficient bandwidth to calculate addresses for all loads, i.e., to *re*-calculate addresses for non-bypassing loads. This design option eliminates yet another structure from the processor at the cost of performing an additional register read and an additional address calculation for each non-bypassing load. Note that with a virtually tagged T-SSBF, an additional address translation is needed only for bypassing loads that ultimately re-execute.

Summary: out-of-order core/in-order back-end designs. Figure 1 shows three organizations for an out-of-order core and in-order back-end pipeline. Each diagram shows paths for a single load and a single store.

The top diagram shows a microarchitecture with an associative store queue and non-associative load queue, i.e., load re-execution with SVW filtering. All loads and stores execute out-of-order and the back-end pipeline obtains store and load addresses and values from the store and load queues.

The middle diagram shows NoSQ. The store queue is eliminated and stores execute in-order in the back-end pipeline using base register and offset information from the ROB. Notice, the number of register ports and address generation units is unchanged. Verification of bypassed loads shares these structures with store commit.

The bottom diagram shows NoSQ with load queue elimination. The cost of eliminating the load queue is re-calculating addresses for non-bypassed loads in the in-order back-end. To provide sufficient bandwidth for doing so, an additional register read port and address generation unit are needed. With this modest additional bandwidth, the performance of NoSQ with and without a load queue is identical. Table 4 shows NoSQ’s actions in the back-end pipeline with load queue elimination.

3.5. Partial-word Bypassing

For SMB to effectively perform as the only in-flight store-load communication mechanism, it must correctly handle all common cases, including many partial-word communication instances. Performing SMB on partial-word operations requires additional mechanisms, because a partial-word store-load pair may implicitly perform mask, shift, and sign/zero extend operations on the value passed from DEF to USE in a DEF-store-load-USE chain. On a 64-bit architecture, a partial-word store implicitly truncates (or masks) an 8-byte register value to *n* (1, 2 or 4) bytes when storing it to memory. A partial-word load implicitly zero/sign extends an *m* (1, 2, or 4) byte value to 8-bytes. Finally, a partial-word store-load pair may, from the point of view of the USE, perform a shift on DEF’s value, e.g., when a narrow load reads the upper half of the word written by a wide store. On the Alpha architecture (which is our experimental platform), there is yet another possible transformation. The *lds* and *sts* instructions convert from/to an in-memory 32-bit IEEE754 single-precision floating-point format to an in-register 64-bit representation. For SMB to successfully perform partial-word bypassing, it must mimic these transformations.

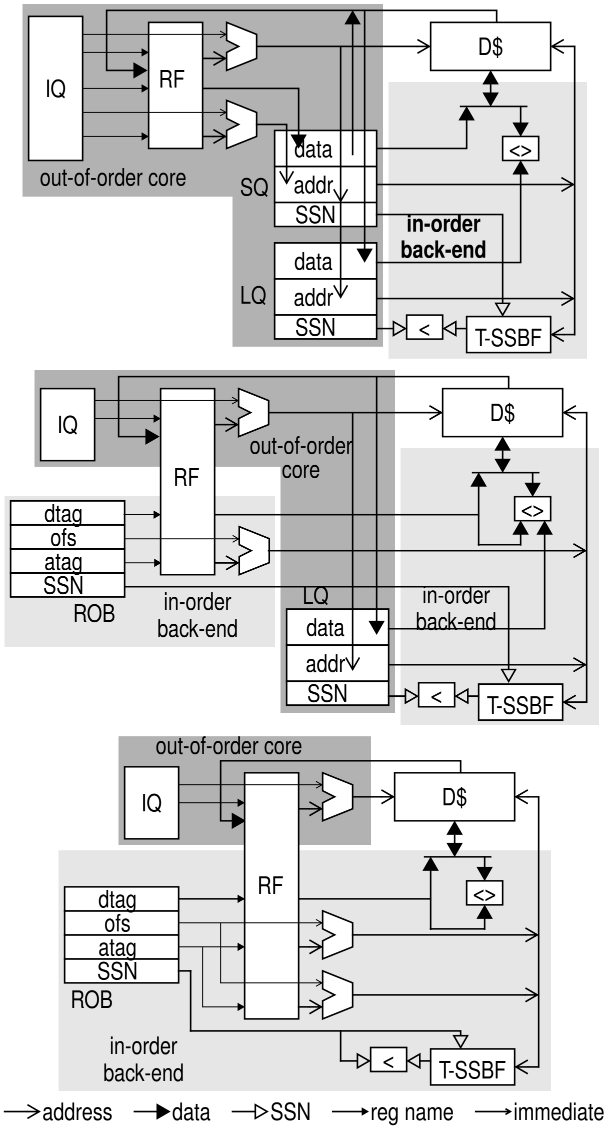

Figure 1. Three out-of-order core/in-order back-end divisions. Top: a conventional division with an associative store queue and non-associative load queue. **Middle:** NoSQ with a non-associative load queue. **Bottom:** NoSQ with no load queue.

To perform partial-word SMB, NoSQ injects a speculative ***shift & mask instruction*** into the out-of-order engine in place of the original load. This instruction reads the store’s data input register, performs the necessary transformation, and writes the value to the bypassed load’s output register. Based on the size and type of the store and load involved (the store size and type is recorded in the store register queue), NoSQ can non-speculatively determine (i) which bytes to mask, (ii) whether to zero-extend or sign-extend the value, and (iii) whether to apply the floating point transformation. However, NoSQ cannot determine a shift amount without the load and store addresses, and so shift amounts must be learned and predicted. To do this, NoSQ (i) extends each entry in the bypassing predictor with a shift amount and (ii) extends each entry in the T-SSBF with the store's size and low-order address bits. At commit, this additional information is combined with the size and low-order bits of a load's address to both learn shift amounts and, equivalently, to verify (without replay) that the predicted shift amount was correct.

4. Experimental Evaluation

The performance goal of NoSQ is to match a design with conventional forwarding using a fully-associative store queue. Because NoSQ has a longer back-end pipeline and can suffer from squashes due to store-load forwarding mis-speculation, it can under-perform (in terms of IPC) a conventional design. Alternatively, as NoSQ uses speculative memory bypassing, it can modestly exceed the performance of a conventional design. Experiments show that on average, NoSQ outperforms a conventional design, because the impact of its infrequent mis-predictions is offset by the latency benefits of SMB and by the additional benefits of eliminating the store queue and out-of-order execution of stores. As a secondary benefit, NoSQ reduces overall data cache accesses by 9% on average. The following experiments characterize the store-load communication behavior of our benchmarks, evaluate the accuracy of the bypassing predictor, and measure NoSQ's performance.

4.1. Methodology

We evaluate NoSQ using timing simulation on the SPEC2000 and MediaBench benchmark suites. The SPEC programs run on their training input sets using 2% periodic sampling with 2% cache and branch predictor warm-up and 10M instructions per sample. The Media programs run unsampled on their provided inputs. All programs run to completion.

The detailed simulator executes the Alpha AXP user-level ISA using the ISA definition and system call modules supplied by SimpleScalar 3.0. It models a 4-way fetch/issue/commit superscalar processor with a 128-entry reorder buffer, 48-entry non-associative load queue, 40-entry issue queue, and 160 physical registers. It models 64KB, 2-way set-associative instruction and data caches, 128-entry, 4-way set-associative TLBs, and a 1MB, 8-way set-associative 10 cycle-access L2. Memory latency is 150 cycles and the memory bus is 16 bytes wide and clocked at one quarter processor frequency. The front end can predict two branches per cycle and fetch past one. It uses an 12k-entry hybrid gShare/bimodal predictor, a 2k-entry, 4-way set-associative target buffer, and a 32-entry RAS. The front-end and execution pipelines total 11 stages: 1 predict, 3 fetch, 1 decode, 1 rename, 1 schedule, 2 register read, 1 execute, and 1 complete. Data cache latency is 3 cycles, so the load pipeline is 15 stages. The scheduler can issue up to 4 instructions per cycle: 4 simple integer, 2 complex integer/FP, 1 branch, 1 load and 1 store. Load speculation is verified by SVW filtered re-execution. The SVW configuration uses 20 bit SSNs and a 128-entry, 4-way associative T-SSBF with 2 read ports and 1 write port. Each T-SSBF entry is 8 bytes: 20-bit SSN, 3-bit offset, 3-bit store data size, and 38-bit tag; total T-SSBF size is 1KB.

	in-window store-load communication (% committed loads)		bypassing mis-predictions (per 10k loads)	
	total	partial-word	no delay	delay (% loads delayed)
adpcm.d	0.0	0.0	0.2	0.2 (0.0)
adpcm.e	0.0	0.0	0.2	0.2 (0.0)
epic.e	8.4	1.9	5.3	1.0 (0.3)
epic.d	17.0	5.0	8.9	5.3 (2.7)
g721.d	6.3	4.7	0.0	0.0 (0.0)
g721.e	6.9	5.8	40.9	0.7 (0.4)
gs.d	12.3	8.0	56.8	4.5 (3.3)
gsm.d	1.4	0.3	2.1	2.3 (0.2)
gsm.e	1.1	0.5	0.4	0.1 (0.0)
jpeg.d	1.1	0.2	2.2	1.9 (1.6)
jpeg.e	10.8	0.2	8.0	3.3 (1.8)
mesa.m	42.7	18.6	84.5	7.9 (5.2)
mesa.o	48.0	19.0	76.3	7.7 (5.8)
mesa.t	32.3	15.4	51.1	7.0 (4.5)
mpeg2.d	24.3	0.4	2.0	0.8 (0.4)
mpeg2.e	4.4	0.6	0.7	0.3 (0.1)
pegwit.d	6.4	6.3	6.2	2.4 (1.1)
pegwit.e	5.6	4.7	7.1	2.5 (1.2)
Media.avg	**12.7**	**5.1**	**21.6**	**2.0 (1.6)**
bzip2	8.8	5.9	24.6	3.8 (5.3)
crafty	2.8	1.9	17.5	5.7 (3.1)
eon.c	20.4	3.2	61.2	10.8 (4.3)
eon.k	15.4	1.7	56.6	13.9 (6.2)
eon.r	17.3	2.5	71.4	14.0 (6.1)
gap	8.1	0.2	4.5	1.3 (1.5)
gcc	7.7	1.4	17.4	10.4 (6.3)
gzip	15.0	8.7	7.3	2.5 (1.3)
mcf	0.9	0.1	27.7	5.0 (2.7)
parser	8.2	2.6	22.4	8.4 (4.2)
perl.d	9.9	1.9	4.5	2.1 (1.3)
perl.s	11.5	2.7	4.9	2.4 (1.5)
twolf	6.3	5.0	21.4	4.9 (2.5)
vortex	17.9	4.7	12.1	2.9 (1.7)
vpr.p	6.3	4.5	55.0	7.9 (4.6)
vpr.r	17.0	5.6	34.1	12.8 (5.2)
Int.avg	**10.8**	**3.3**	**17.5**	**4.5 (3.6)**
ammp	4.1	0.1	4.4	2.0 (0.8)
applu	4.9	0.0	0.1	0.1 (0.1)
apsi	3.8	0.5	4.7	0.3 (1.3)
art	1.4	0.4	0.1	0.1 (0.0)
equake	3.2	0.1	0.7	0.1 (0.1)
facerec	0.8	0.6	0.2	0.1 (0.3)
galgel	0.5	0.0	0.5	0.2 (0.1)
lucas	0.0	0.0	0.0	0.0 (0.0)
mesa	12.1	1.7	2.2	0.2 (3.0)
mgrid	1.2	0.0	0.1	0.0 (0.0)
sixtrack	9.4	1.0	59.2	10.7 (4.2)
swim	2.9	0.0	0.3	0.1 (0.1)
wupwise	5.5	0.8	1.8	0.2 (0.1)
FP.avg	**3.8**	**0.4**	**3.0**	**0.7 (0.8)**

Table 5. Communication behavior and prediction accuracy.

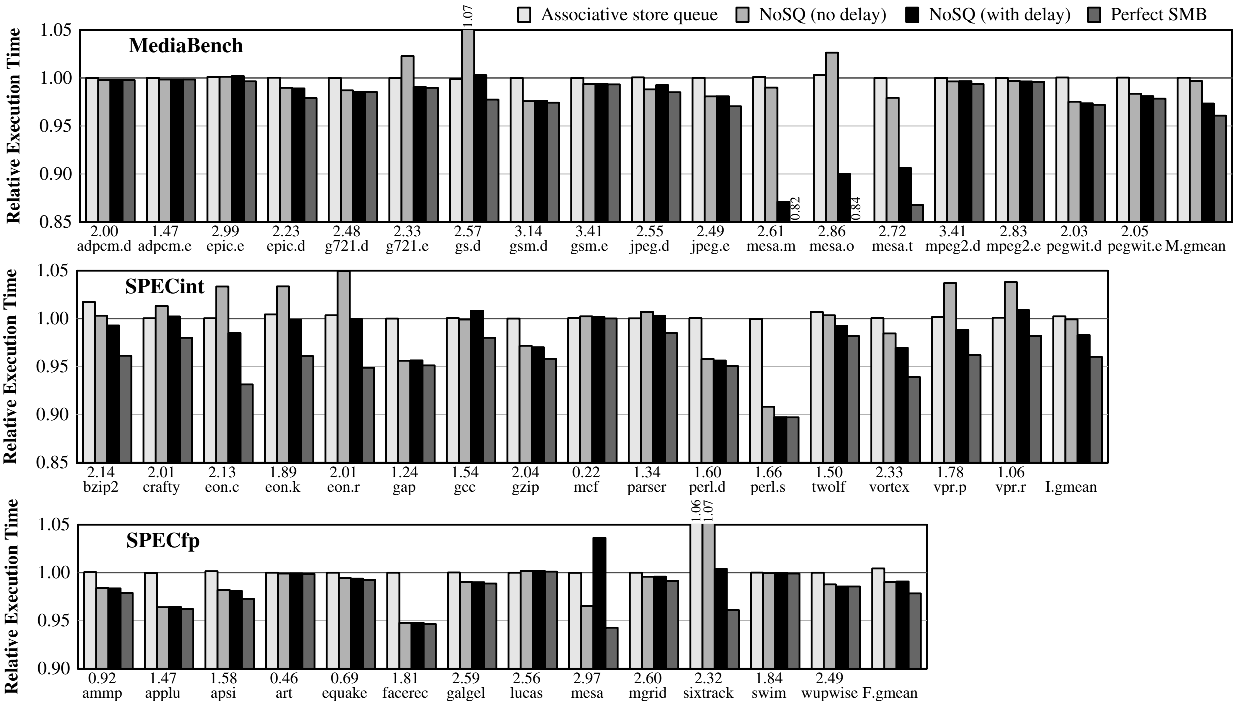

Figure 2. NoSQ performance on machine with 128-instruction window. Execution times relative to a conventional processor with perfect load scheduling: (i) microarchitecture with associative store queue, (ii) NoSQ without delay, (iii) NoSQ with delay, and (iv) an idealized NoSQ configuration.

The baseline has a 6 stage back-end pipeline: 1 setup, 1 SVW, 3 data cache, 1 commit. It has a 24-entry associative store queue and a 4k-entry StoreSets load scheduling predictor. NoSQ has an 8 stage back-end pipeline: 1 setup, 2 register read, 1 agen/SVW, 3 data cache, 1 commit. The bypassing predictor uses two 1K-entry, 4-way set-associative tables: one indexed by load PC, and one indexed by an XOR hash of load PC and 8 bits of branch/call history. Each entry contains a 6-bit distance field (corresponding to 64 in-flight stores), a 3-bit shift amount, a 2-bit store size, a 7-bit confidence counter, and a 22-bit tag, for a total of 5 bytes; the entire predictor is 10KB. Again, from a performance standpoint there is no difference whether NoSQ includes a load queue or not.

4.2. Communication Patterns and Prediction Accuracy

Table 5 shows the store-load communication behavior of the benchmarks. The left side shows the percentage of committed loads that—in a 128 instruction window with no limit on the number stores—experience store-load communication of any kind (*total*) and partial-word communication in particular (*partial-word*). Partial-word communication includes any situation in which either the load or store is less than eight bytes wide. The majority of loads do not communicate with older stores: 100% in some benchmarks and on average 87% in MediaBench, 89% in SPECint, and 96% in SPECfp. However, a few benchmarks have a high degree of store-communication (up to 48%). Whereas full-word communication is the common case, partial-word communication is common in many benchmarks (e.g., SPECint's *gzip* and *vpr;* MediaBench's *gs.d*, *pegwit*, and *mesa*) motivating NoSQ's support for partial-word bypassing using shift and mask instructions.

The right side shows the accuracy of NoSQ's predictor (in mis-predictions per 10,000 loads) for two different configurations. The first (*no delay*) shows raw prediction accuracy; here NoSQ does not delay loads with difficult bypassing behaviors. Even in this configuration, no benchmark has a mis-prediction rate above 1% (100 in 10,000 loads) and only 15 of 47 benchmarks have mis-prediction rates above 0.2% (20 in 10,000) loads. Average mis-prediction rate is 0.2% in MediaBench and SPECint and 0.03% in SPECfp. In the second configuration (*delay*), NoSQ uses delay to handle difficult bypassing behavior, e.g., partial-store bypassing. Next to prediction accuracy, in parentheses, the table lists percentage of all committed loads delayed by NoSQ. The addition of delay reduces mis-predictions to 0.02%, 0.05%, and 0.01% in MediaBench, SPECint, and SPECfp, respectively. No benchmark has a mis-prediction rate higher than 0.2%. To achieve this reduction, NoSQ delays an average of 1.6%, 3.6%, and 0.8% of loads, respectively. Delay prevents mis-predictions caused by partial-store communication in *g721.e* (two 1-byte stores to a 2-byte load) and reduces squashing due to hard-to-predict loads in *eon, vpr, sixtrack*, and MediaBench's *mesa*.

4.3. Performance

Figure 2 shows execution times of four configurations, relative to a microarchitecture with an associative SQ and perfect load scheduling. The IPC of this ideal baseline is printed above each benchmark name. The first bar in each group corresponds to a processor with an associative SQ and realistic StoreSets load scheduling. This experiment affirms that StoreSets is an effective load scheduling predictor—the performance difference between realistic and idealized scheduling is

The 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)
0-7695-2732-9/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY

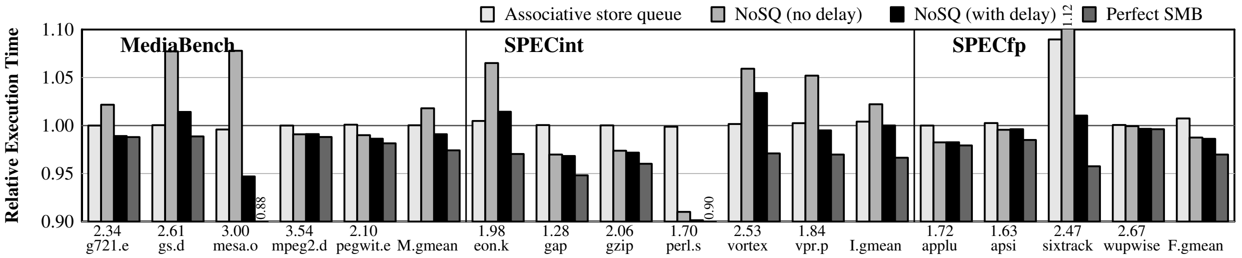

Figure 3. NoSQ performance on machine with 256-instruction window. Execution times relative to a baseline with associative store queue and perfect scheduling: (i) microarchitecture with associative store queue, (ii) NoSQ without delay, (iii) NoSQ with delay, and (iv) an idealized NoSQ configuration.

negligible in every benchmark except *bzip2* (2%) and *sixtrack* (6%)—and establishes a realistic baseline for NoSQ.

The second bar shows NoSQ with a realistic predictor and no delay, i.e., without any load scheduler in the out-of-order core. Overall, NoSQ slightly outperforms the realistic conventional configuration: by 0.3%, 0.3% and 1.4% for MediaBench, SPECint and SPECfp, respectively. In 22 of 47 benchmarks, NoSQ improves performance by more than 1%. These improvements are largely due to SMB's latency reducing effects. Although NoSQ also improves performance in three other ways—(i) eliminating store queue capacity dispatch stalls, (ii) reducing contention for issue bandwidth, and (iii) reducing contention for issue queue entries—the baseline processor is well-balanced meaning these resources are not a bottleneck. In 15 programs, NoSQ's performance is within 1% of a conventional design's. Finally, in 10 of 47 benchmarks, squashes due to bypassing mis-predictions result in more than a 1% slowdown for NoSQ.

The third bar shows NoSQ with delay. Delay both reduces the number of benchmarks with slowdowns and improves average performance. Adding delay reduces the number of benchmarks with more than a 1% slowdown to only 1 of 47 benchmarks (SPECfp's *mesa*). On average, it improves NoSQ's performance by 2.4% and 1.7% for MediaBench and SPECint where partial-store and other difficult communication behaviors are more frequent. However, over-delay can reduce NoSQ's performance, and it does for three benchmarks (*jpeg.d, gcc* and *mesa*).

The fourth bar shows an idealized NoSQ configuration with a perfect bypassing predictor and idealized support for partial-word bypassing. This configuration outperforms a conventional configuration by only 3.7% on average, jibing with previous assertions that—relative to a baseline with intelligent load scheduling—speculative memory bypassing is not a compelling performance technique [9]. NoSQ captures about half the benefit of that ideal case.

4.4. Performance Scalability

Figure 3 shows NoSQ's performance on a machine with a 256-instruction window. All window resources are doubled and the branch predictor size is quadrupled; however, NoSQ's bypassing predictor is not enlarged. The figure shows data for selected benchmarks, but includes suite-wide means.

A larger window increases store-load communication rates provides more opportunity for SMB and its performance benefits, as evidenced by the relatively improved performance of the idealized SMB configuration. However, a larger window also increases the frequency of difficult communication patterns as well as increasing the probability that a window-resident path-dependent communication instance will have a path signature that is longer than the one supported by the predictor. As a result, bypassing mis-predictions increase—delay does compensate for these somewhat, but in a large window delay is also expensive—and the performance of (realistic) NoSQ suffers. On average, NoSQ's improvement drops to 1% from 2%. Experiments show that NoSQ's performance can be largely restored by a larger bypassing predictor with a longer branch history.

4.5. Data Cache Read Bandwidth Consumption

In NoSQ, most bypassed loads never access the data cache. If the number of bypassed loads is large, cache read bandwidth reduction can be significant. Alternatively, if the number of bypassed loads is small and SVW cannot successfully filter re-executions for non-bypassed loads—causing them to access the cache twice—NoSQ can increase cache read bandwidth consumption.

Figure 4 shows data cache reads for NoSQ, normalized to those of the associative store queue baseline. Each bar shows out-of-order engine reads (bottom) and back-end re-execution reads (top). Due to the re-execution filtering effectiveness of the T-SSBF—only 0.7% of loads re-execute—NoSQ reduces data cache reads at a rate proportional to the frequency of bypassing: about 4% for SPECfp and over 10% for MediaBench and SPECint on average, although several programs see

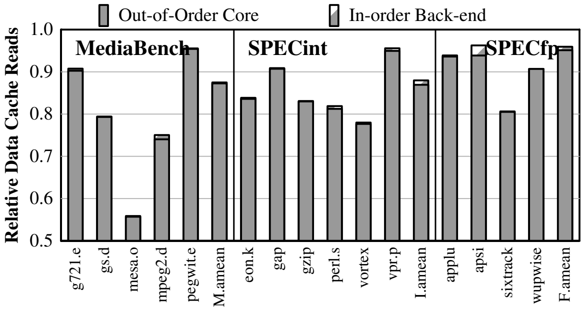

Figure 4. Data cache reads. Number of data cache reads for NoSQ with delay, relative to a baseline with associative store queue and load re-execution/SVW verification.

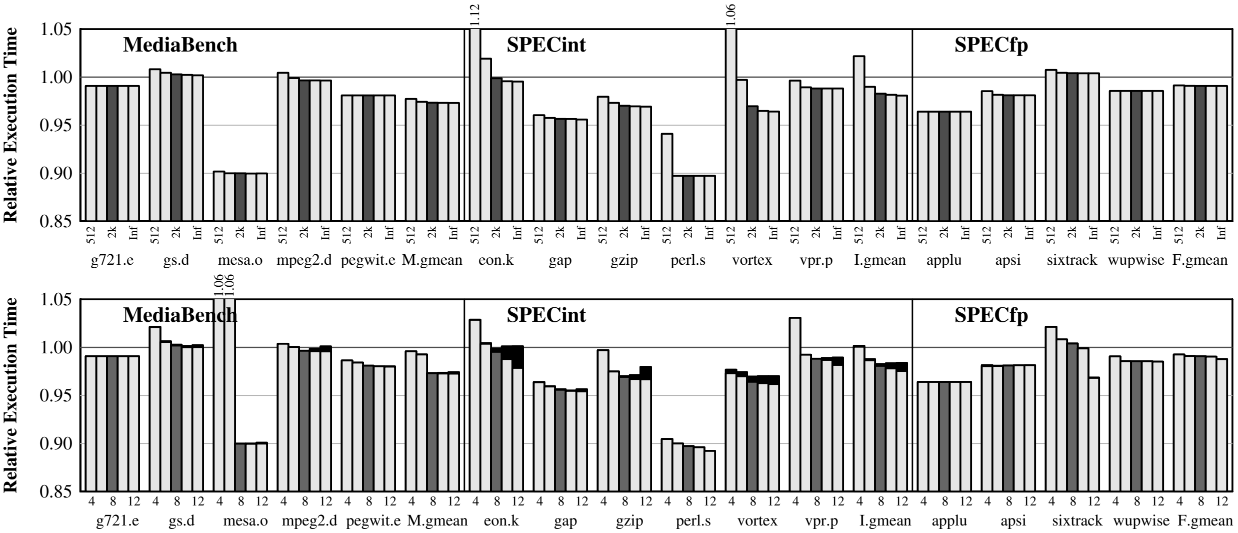

Figure 5. NoSQ bypassing predictor sensitivity analysis. Top: predictor capacity. **Bottom:** predictor history bits.

bandwidth reductions of 20% and one (*mesa.o*) experiences a 40% reduction in data cache reads.

4.6. Bypassing Predictor Sensitivity Analysis

Figure 5 shows NoSQ's sensitivity to the configuration of its bypassing predictor.

Capacity. The top graph shows the performance impact of predictor capacity with relative execution times for predictors with 512, 1K, our default 2K, 4K, and unbounded entries. All predictors use 8 bits of history and a hybrid design with the storage equally-split between the two tables. The results show that in a 128-instruction window: (i) the baseline 2K-entry predictor is almost as effective as a predictor of unbounded size, and (ii) reducing predictor size by a factor of four (to 512 entries and 2.5KB of storage) has little effect on MediaBench and SPECfp, but reduces the performance of SPECint by 4%.

Branch history. The bottom graph shows the impact of pattern history length for 4, 6, our default 8, 10, and 12 branch history bits. The dark upper segments of the bars show a predictor with unbounded capacity. For most benchmarks, 6 or 8 history bits capture most of the benefit. Only a few benchmarks benefit from more than 8 history bits (*e.g., eon.k* and *sixtrack*). Generally, longer histories reduce performance for the 2K-entry predictor due to the capacity pressure caused by an increase in the number of path history patterns per load.

5. Related Work

There have been many recent proposals to improve the scalability and reduce the complexity of store-load forwarding via a store queue. Proposals can be grouped into three general classes. The first class maintains the age-ordered store queue structure but uses partitioning, filtering, hierarchy, dependence speculation, and speculative forwarding through the primary data cache or other structures to reduce the frequency of associative store queue search or the number of entries examined per search [2, 5, 12, 18, 20]. A second class avoids associative search by abandoning the conventional age-ordered structure and replacing it with a cache-like address-indexed structure [6, 18, 21, 24]. A third class maintains the simplifying age-ordered structure but uses dependence speculation to replace associative search with speculative indexed access [19]. Our design differs from all of these in a fundamental way: rather than reducing the complexity of forwarding by optimizing the store queue, it eliminates the store queue and implements forwarding using different mechanisms based on speculative memory bypassing.

Fire and Forget (FnF) [22] is a concurrently-proposed alternative scheme for eliminating the store queue. FnF accomplishes this by turning store-load forwarding from a load-centric activity to a store-centric activity and using load queue index prediction to perform forwarding through the load queue instead. Both NoSQ and FnF employ SVW and distance-based prediction. Unlike NoSQ, FnF does not eliminate the out-of-order execution of stores or forwarding (bypassing) loads.

6. Conclusions

Designing a fast and efficient load/store unit that is tightly integrated enough into the data path to support efficient store-load forwarding is a challenge for both current and future processors. This paper presents an alternative approach to this problem: NoSQ, a microarchitecture without an explicit store-load forwarding unit in the out-of-order datapath. In the NoSQ microarchitecture, the out-of-order execution engine contains no store queue and does not execute stores. Bypassed loads do not execute in the out-of-order engine, and non-bypassed loads obtain their values from the data cache. All the functions of a traditional in-flight load/store unit—store-load forwarding, in-order store-commit, load mis-speculation detection—are pushed out of the core into the front-end (decode, rename) and back-end (commit) pipelines.

NoSQ exploits the synergy that exists between three mechanisms: (i) speculative memory bypassing (SMB), (ii) highly-accurate store-load forwarding prediction, and (iii) in-order load re-execution with store vulnerability window (SVW) re-execution filtering. NoSQ combines these mechanisms—using a few observations and modest modifications to smooth mechanism interfaces that do not naturally fit together, like SMB and SVW—and exploits the strengths of one to cancel the drawbacks of the other.

Timing simulation shows that NoSQ slightly outperforms (in terms of IPC) a conventional associative load/store unit design, despite being more speculative and having a longer back-end pipeline. The performance advantages of SMB more than offsets the performance overheads of mis-speculations. Prior work has found that for RISC ISAs, only 10–15% of the loads—or 2–3% of all instructions—can exploit SMB, and concluded that strictly as a performance technique, SMB was "not worth the effort" [9]. Perhaps when used to simplify the datapath in addition to providing a performance benefit, SMB may be worth the effort after all.

Acknowledgments

The authors thank their reviewers for their comments and suggestions for improving this manuscript. This work was supported by NSF CAREER award CCF-0238203, NSF CPA grant CCF-0541292, and donations from Intel.

References

[1] D. Adams, A. Allen, R. Bergkvist, J. Hesson, and J. LeBlanc. "A 5ns Store Barrier Cache with Dynamic Prediction of Load/Store Conflicts in Superscalar Processors." In *Proc. 1997 International Solid-State Circuits Conference*, Feb. 1997.

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

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

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

[6] 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*, pages 142–153, Jun. 2006.

[7] 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.

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

[9] G. Loh, R. Sami, and D. Friendly. "Memory Bypassing: Not Worth the Effort." In *Proc. 1st Workshop on Duplicating, Deconstructing, and Debunking*, pages 71–80, May 2002.

[10] 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.

[11] S. Onder and R. Gupta. "Load and Store Reuse using Register File Contents." In *Proc. 15th International Conference on Supercomputing*, pages 289–302, Jun. 2001.

[12] I. Park, C. Ooi, and T. Vijaykumar. "Reducing Design Complexity of the Load/Store Queue." In *Proc. 36th International Symposium on Microarchitecture*, pages 411–422, Dec. 2003.

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

[14] 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.

[15] 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.

[16] 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.

[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] S. Sethumadhavan, R. Desikan, D. Burger, C. Moore, and S. Keckler. "Scalable Hardware Memory Disambiguation for High ILP Processors." In *Proc. 36th International Symposium on Microarchitecture*, pages 399–410, Dec. 2003.

[19] 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.

[20] S. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. "Continual Flow Pipelines." In *Proc. 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2004.

[21] 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.

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

[23] S. Subramaniam and G. Loh. "Store Vectors for Scalable Memory Dependence Prediction and Scheduling." In *Proc. 12th International Symposium on High Performance Computer Architecture*, pages 64–75, Feb. 2006.

[24] 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.

[25] G. Tyson and T. Austin. "Improving the Accuracy and Performance of Memory Communication Through Renaming." In *Proc. 30th International Symposium on Microarchitecture*, pages 218–227, Dec. 1997.

[26] 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.