Scalable Load and Store Processing in Latency Tolerant Processors

Amit Gandhi[†] Haitham Akkary Ravi Rajwar Srikanth T. Srinivasan Konrad Lai

[†]*Electrical and Computer Engineering*
Portland State University

Microarchitecture Research Lab
Intel Corporation

[†]*gandhi@ece.pdx.edu, {haitham.h.akkary, ravi.rajwar, srikanth.t.srinivasan, konrad.lai}@intel.com*

Abstract

Memory latency tolerant architectures support thousands of in-flight instructions without scaling cycle-critical processor resources, and thousands of useful instructions can complete in parallel with a miss to memory. These architectures however require large queues to track all loads and stores executed while a miss is pending. Hierarchical designs alleviate cycle time impact of these structures but the CAM and search functions required to enforce memory ordering and provide data forwarding place high demand on area and power.

We present new load-store processing algorithms for latency tolerant architectures. We augment primary load and store queues with secondary buffers. The secondary load buffer is a set associative structure, similar to a cache. The secondary store buffer, the Store Redo Log, is a first-in first-out structure recording the program order of all stores completed in parallel with a miss, and has no CAM and search functions. Instead of the secondary store queue, a cache provides temporary forwarding. The SRL enforces memory ordering by ensuring memory updates occur in program order once the miss returns.

The new algorithms eliminate the CAM and search functions in the secondary load and store buffers, and remove fundamental sources of complexity, power, and area inefficiency in load/store processing. The new organization, while being area and power efficient, is competitive in performance compared to hierarchical designs.

1. Introduction

Large instruction window processors capable of sustaining thousands of in-flight instructions can effectively tolerate relatively increasing memory latencies. They do so by executing thousands of useful miss-independent instructions in parallel with the pending miss [13]. These independent instructions constitute a significant portion of the instruction window following a miss [11]. Recent proposals have demonstrated how to design processors to sustain such large numbers of in-flight instructions without having to scale up the cycle critical register file and scheduler, and the reorder buffer [5, 17].

Continual Flow Pipeline processors [17] achieve resource efficiency by ensuring instructions that depend upon a long latency miss do not block cycle critical processor resources. These resources become available for subsequent miss-independent instructions to execute and complete. The approach is particularly effective because the majority of instructions following a long latency miss are independent of the miss. These instructions can execute and speculatively retire, freeing up their resources in the process. The small numbers of miss-dependent instructions drain out of the pipeline, releasing their resources, and wait in a simple first-in first-out data buffer. When the miss returns, these miss-dependent instructions re-enter the pipeline, re-allocate resources, and execute. The processor then integrates the results of the miss-independent and miss-dependent instructions together without requiring the miss-independent instructions to be re-examined. Since miss-dependents do not block resources, small sizes for the register file, scheduler, and reorder buffer are sufficient to sustain thousands of in-flight instructions to tolerate a long latency miss. The small sizes result in resource efficiency. High power efficiency arises because the numerous miss-independent instructions are not re-executed.

While the above proposals effectively address the register-file, scheduler, and reorder buffer for designing very large instruction window processors, they nevertheless require buffering all loads and stores to ensure correct memory ordering and data forwarding requirements.

A load might have incorrectly executed because either a memory dependence predictor incorrectly predicted the load to be independent of a miss-dependent store, or a store to the same address from another thread or processor executed, thus requiring the load to re-execute to enforce correct multiprocessor memory ordering. Detecting these conditions requires tracking all loads, dependent and independent. Conventional load queues implement this as a fully associative CAM of the store address against all loads in the load queues.

Store queues aid memory-address disambiguation between stores and loads, provide buffering for stores until retirement, and provide data to loads following the stores. These functions require searching the store queue because

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

a load may depend upon any store in the queue, and multiple stores to the same address can be simultaneously present in the store queue. Hierarchical solutions to the store queue design [1] ensure the latency tolerant processor meets cycle time constraints. However, these hierarchical structures occupy significant area and display power inefficiencies. The CAM and the search required for memory ordering and data forwarding is a fundamental source of complexity, power, and area inefficiency. Recent proposals [15, 16] address complexity of these structures by reducing their active power, optimizing search through filtering, and sectoring, but do not address the CAM itself and largely ignore the increasing area footprint of very large fully associative store queues.

We present new scalable load and store-processing mechanisms for memory latency tolerant processors. These mechanisms do not require searching large secondary load and store queues. We take advantage of a key property of latency-tolerant processors: in the presence of a long latency memory miss, a significant portion of the useful instructions (including loads and stores) following the miss, are independent of the miss [11], and do not need to re-execute when the miss returns. This allows us to optimize their processing. Further, the instruction window in latency tolerant processors needs to scale only in the presence of a long latency miss. The secondary buffers are operational only during the miss, while small conventional primary load and store queues are sufficient in the absence of misses.

The secondary load and store processing has three key actions:

1. Miss-independent stores temporarily update a cache and use it to forward to future independent loads. Their program order is recorded in a first-in first-out Store Redo Log (SRL),
2. The temporary updates are discarded when the miss returns and the slice executes. The independent stores re-update the cache using the store redo log appropriately interleaved in program order with the miss-dependent instructions, and
3. Internal and external stores are snooped by a set-associative secondary load address buffer. Recovery is checkpoint-based and checkpoint bits determine where to roll back. Because recovery is coarse-grain, exact load order information is not necessary, and thus a set-associative cache structure is sufficient. This allows a scalable solution for load processing without loss in performance.

The first action provides simple and fast store-to-load forwarding. Since these stores were independent of the miss, they do not have to re-execute when the miss returns. The second action ensures correct ordering. Discarding temporary updates restores the correct memory image to miss-dependent loads and stores. The third action ensures that in case of a memory-dependence prediction violation or a consistency violation, execution restarts from the appropriate point.

These actions do not require associatively searching either the large secondary store queue or the load buffer, and thus do not require CAM logic. Eliminating the CAM logic structure from each cell of the secondary load and store buffers and not requiring fully associative searches outweighs the performance and power overhead of re-updating the cache. Instead of optimizing the search of these structures, as some earlier proposals do [15, 16] via filtering and sectoring, we eliminate the search itself.

This paper makes the following contributions to load/store processing in latency tolerant processors:

- A novel redo algorithm for processing store operations that does not require searching a large secondary store queue for maintaining ordering.
- A simpler and smaller secondary store queue structure because it does not have a CAM. This means much smaller area and lower power demands.
- A simple secondary load buffer that is scalable and set-associative. The buffer has a cache organization. A store identifier per entry determines load and store program order and checkpoint bits allow rollback to recover from memory violations.

Section 2 describes a baseline latency tolerant processor, motivates the need for large load/store queues in such processors, and presents a complexity analysis of conventional load and store queues. Section 3 and Section 4 present our new load and store proposals. Section 5 presents experimental methodology and Section 6 presents power, performance, and area results. We discuss related work in Section 7 and conclude in Section 8.

2. Latency tolerant processor design

We begin by describing our baseline latency tolerant processor in Section 2.1. We qualitatively and quantitatively motivate the necessity for large load and store buffers for latency tolerant processors in Section 2.2 and we discuss the complexity of load and store processing in Section 2.3.

2.1 Baseline microarchitecture

Our baseline processor, shown in Figure 1 is a Continual Flow Pipeline (CFP) processor [17] implemented on a reorder-buffer-free Checkpoint Processing and Recovery (CPR) microarchitecture [1]. CPR removes scalability limitations for branch misprediction recovery and register reclamation mechanisms. A small number of selectively created register rename-map table checkpoints enable quick and efficient misprediction recovery. These checkpoints also enable CPR to implement an aggressive register reclamation scheme and provide precise interrupts. CPR decouples register reclamation from instruction re-

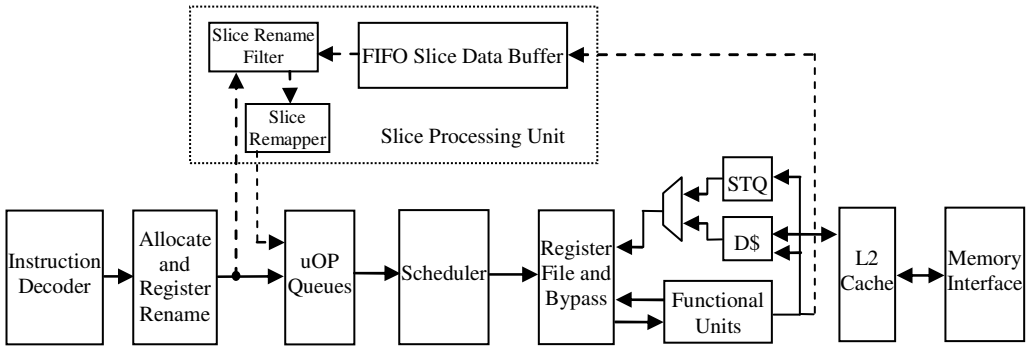

Figure 1 Block diagram of a CFP processor

tirement. Since CPR does not have a reorder buffer, checkpoint counters track instruction completion. A checkpoint is committed instantaneously, in a bulk commit manner, when all instructions within it have completed. CPR provides resource-efficient processor design to handle short and medium latencies. A CFP mechanism allows the processor to handle even very long latencies in a resource efficient manner.

In conventional processor designs, a long latency miss and its dependent instructions occupy cycle critical register file and scheduler resources while the miss is pending. These blocked instructions stall the processor for a long time since later miss-independent instructions, which are in the thousands and are a significant fraction of the useful window that can execute in the shadow of a miss, are unable to execute because they do not have resources. In a CFP processor, the long latency miss operations and their dependents do not occupy cycle-critical structures while the miss is pending. Doing so allows future miss-independent instructions to execute and complete in parallel with the outstanding miss. These miss-independent instructions speculatively retire, and their results are automatically integrated when the miss-dependent instructions (called the miss forward slice) later execute. The miss-dependent instructions, along with their ready source operands, leave the processor pipeline, release their scheduler, register file, and reorder buffer resources (if CFP is on a machine with a reorder buffer), and drain into a first-in first-out slice data buffer (SDB).

A poison bit associated with each physical register and store queue entry identifies the slice and propagates dependence information. A load that misses to memory sets its destination register's poison bit. Any subsequent instructions reading the register inherit the poison bit for their destination registers. This bit propagates through the load-miss dependence chain until the miss data returns. A store reading a poisoned destination also sets its store queue entry's poison bit. Memory dependences are also properly constructed. Destination registers of loads dependent on a poisoned store or predicted to depend on a poisoned store by a memory dependence predictor [4, 14] have their poison bits set.

2.2 Necessity for large load and store queues

In CFP processors, even though miss-independent instructions complete and speculatively retire thus releasing any cycle-critical microarchitecture resources, the processor must track all load and store operations in these instructions until all instructions are architecturally committed after the miss-dependent instructions are processed. This is to ensure correct memory ordering and data forwarding. While these operations do not occupy cycle-critical structures, they occupy large secondary buffers.

2.2.1 Tracking loads

In a CFP processor, even though miss-independent loads have completed, the conventional load queue needs to be large enough to buffer all load addresses: dependent and independent. The processor uses a memory dependence predictor to determine whether a load depends upon a miss-dependent store. This store might not have a known address. The address of the independent load has to be kept in a queue and must be checked when the miss-dependent store eventually executes. Further, to ensure proper multiprocessor memory ordering, the processor must record the addresses of all loads executed and speculatively retired, and check these load addresses against stores from external processors. Our experiments suggest a load queue size of at least 512 entries for the best performance configuration.

2.2.2 Tracking stores

As with load operations, CFP tracks dependent and independent stores. A miss-dependent store will result in all stores after it in program order to wait until the miss returns and the store executes. This is because store updates to memory changes architectural state, and therefore must occur in program order. Independent stores, even though they have completed, must wait until the prior stores complete in program order before updating memory.

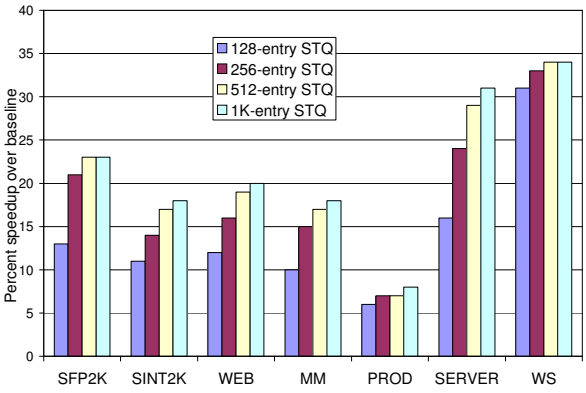

Figure 2 Impact of store queue size for a latency tolerant processor.

Figure 2 shows the performance sensitivity of a CFP processor to the size of its store queue (See Table 1 and Table 1 for configuration and benchmarks). The *y*-axis shows speedup over a configuration with only a 48-entry store queue, and the *x*-axis shows various store queue sizes from 128 to 1024 entries. As can be seen, the store queue must be at least 512 entries to achieve the best performance configuration, and such a size is a significant increase over current store queue sizes of 24-32.

Completed independent stores must also forward data to later independent loads. In an x86 CFP implementation, depending upon the application, the store queue forwards to 20-35% of loads. These loads must search the independent stores in a large store queue and source data. Multiple store queue entries may correspond to the same address and correct store identification is necessary. A load may also need data from multiple store queue entries, and a mechanism for properly aligning data is necessary. Since loads are critical operations and must get their data quickly, the store queue must forward data quickly, typically within the latency of L1 data cache hit. Doing so is difficult for very large store queues.

To make the store queue manageable, the CFP processor uses a two-level hierarchical store queue organization [1]. The first level store queue (L1 STQ) is small (~48 entries) and fast. It holds the most recent stores. The second level store queue (L2 STQ) is large (~1024 entries) and slow and holds older stores displaced from the L1 STQ. Store to load forwarding typically occurs from the L1 STQ because stores typically forward to nearby loads. While this organization provides good performance and does not affect critical cycle time, it places high demands on power and area and makes it a resource inefficient design.

2.3 Load/store processing complexity

Load queue complexity arises from the full CAM performed on internal and external store addresses. Store queue complexity arises because of the matching circuitry

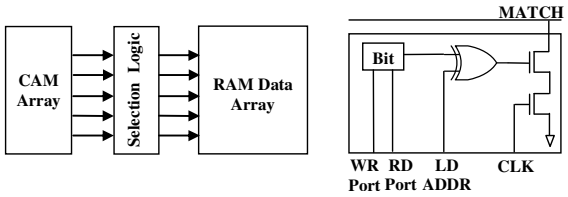

Figure 3 Store queue and CAM array cell.

required to compare issued load addresses with store addresses in the store queue, selecting the correct matching store for forwarding, and, forwarding data to the load from the selected matching store.

Figure 3 shows a store queue with a CAM, select circuitry, and the data array, and the CAM array cell. Each cell has storage for one address bit, one write port to drive the address into the store queue when a store issues, one read port to access the address when a store commits to memory, and a one-bit-comparator implemented as an XOR logic gate. A precharge/discharge signal performs an AND of all bit-comparator outputs within a store entry to generate an address match signal.

Processing multiple loads and stores per cycle requires additional comparators and ports. Every issued load activates the CAM array entries for all stores located prior to the load in the instruction window. This results in significant power consumption in the CAM structure, when the store queue size grows. Further, the CAM structures themselves contribute to leakage power.

Proposals for dealing with store queue complexity [1, 15, 16] have focused on reducing search bandwidth, reducing active power, and tolerating the match-and-search latency using hierarchy, sectoring, and filtering, in various combinations. While they are effective in reducing active power (by up to 90% in some cases), they do not eliminate the dynamic power associated with the numerous CAM cells, nor do they reduce the area and the leakage power of these large store queues.

3. A new secondary load buffer design

We propose a set-associative secondary load buffer. Unlike the primary load queue, it is not organized as a first-in first-out program order queue. Completed loads in the shadow of a miss allocate a L2 load buffer entry based on the load's data address. Miss-dependent loads allocate an L2 load buffer entry when they complete, after the miss data returns. Each entry consists of a tag, the identifier of the nearest store, and checkpoint bits. These checkpoint bits can be bulk reset to remove instantaneously all loads belonging to a given checkpoint from the load buffer. A forwarding store identifier is also stored for each entry to indicate the store, if any, that forwarded data to the load.

Enforcing load-store dependence: A store is assigned an identifier when it is allocated the store buffers. In our

design, a store identifier corresponds to the SRL entry the store is allocated. A wrap-around bit can determine the program order of any two arbitrary stores with a simple magnitude comparison of their identifiers. When a load allocates, it gets the store identifier of the last allocated store prior to it in program order. A magnitude comparison of the store identifier of the load and store determines their relative program order. When a store completes, it looks up the load buffer. On an address match, the load buffer entry's nearest store identifier and the forwarding store identifier (if set) are used to determine if a memory dependence violation occurred. When a memory dependence violation is flagged, the execution restarts from the checkpoint of the violating load, determined by the load entry's checkpoint bits. The load buffer differs from conventional cache organizations since multiple loads with the same address are allocated different entries in the set. In case of a store address hit and memory dependence violations on more than one load entry, a program order check of the violating loads determines the oldest violating load in program order, and a restart from the oldest load checkpoint is initiated.

Enforcing multiprocessor memory ordering: Snooping external stores for enforcing processor ordering does not require an order check between the snooped store and the hit load. A snoop address hit by an external store on any load initiates a restart from the load's checkpoint. If an external store snoop hits more than one load in the set, the restart is initiated from the oldest load checkpoint.

Because of the limited capacity in the load buffer array, a set-overflow may occur when a new load enters the load buffer. One option for handling these overflow cases is to use a small fully associative victim load buffer for overflow loads, or simply to take a memory ordering violation on the overflow.

The baseline checkpoint processing and recovery architecture ensures forward progress by guaranteeing that a new checkpoint is created on the next instruction after a prior restarted checkpoint, thus ensuring at least one instruction always retires.

4. A redo approach to processing stores

The new algorithm replaces the L2 STQ of a hierarchical store queue [1] (which requires search and forward capability) with a much simpler non-searched structure with significantly less area and power demands while providing competitive and at times better performance.

4.1 Decoupling ordering from forwarding

Our goal is to eliminate the search and forwarding functions required of the L2 STQ. This would allow us to reduce area and power by eliminating the CAM cells and the cycle-critical forwarding circuitry. We first sketch our approach below.

We replace the L2 STQ with a first-in first-out (FIFO) structure that does not forward and that records all stores in program order. This structure also does not require search capability. However, independent loads must access data from prior independent stores if necessary. For this, we use the data cache itself. Independent stores will update the data cache, even before they retire, so that later loads can access their data. However, since program order requires stores to update the cache in order, these cache updates are temporary. The data cache now provides the forwarding function of the secondary store queue, and provides temporary buffering space. To prevent the correct memory state (that should be seen by the miss-dependent instructions prior to a temporary store) from being lost, any dirty block that is temporarily updated must, prior to the update, be written back to the next level in the memory hierarchy.

Independent stores execute, temporarily update the cache, and complete. Future independent loads will access the cache (instead of the secondary store queue) to read data of prior independent stores.

Once the cache miss returns, dependent instructions re-enter the pipeline, re-allocate register and scheduler resources, and execute. These dependent instructions are interleaved with the completed miss-independent instructions in program order. To enable these dependent instructions to see the correct memory state, the temporary updates made by the independents are discarded. The independent instructions however have speculatively retired and do not need to be re-executed because they do not depend upon the cache miss.

To ensure memory updates occur in program order, the independent stores (recorded in the secondary store queue in program order) must "re-update" the cache in program order and be properly interleaved with the execution of miss-dependent instructions, after the cache miss is serviced. These independent stores do not re-enter the pipeline and do not consume execution resources: they just update the cache from the secondary store queue consuming only cache write bandwidth. Data dependences are maintained because a dependent load executes only when a prior independent store has re-updated the cache.

Since the independent stores have to be redone in program order from the secondary store queue, we call that structure the Store Redo Log (SRL). The SRL is the L2 STQ without any search and forward capability.

Independent stores from the SRL do not consume execution bandwidth and their re-update of the cache consumes only cache write-bandwidth. These store re-updates occur at the same time these updates would have occurred with a large conventional store queue: when the miss data returns and dependent stores execute and update the cache allowing all stores (including miss independent stores) behind them in the window to proceed with the cache updates. The additional write bandwidth consumption in the data cache due to the temporary updates by independent

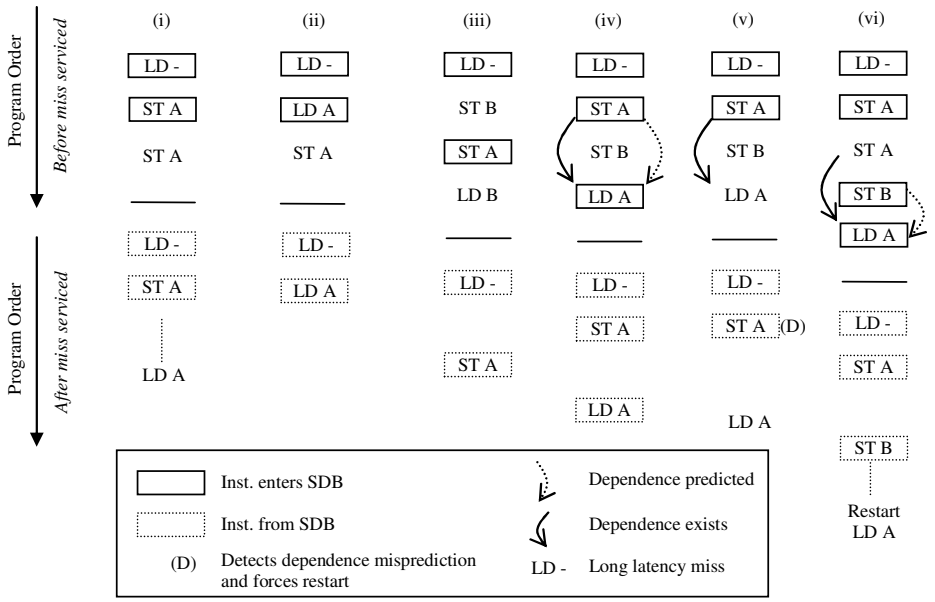

Figure 4 Examples showing various hazard conditions handling in SRL.

stores occurs at a time such store updates would have stalled anyway in a conventional STQ design due to miss latency delays. Therefore, the SRL algorithm does not require increase in cache write bandwidth.

4.2 Correct ordering without SRL CAM

We now step through some scenarios to show the SRL workings. Consider the instruction sequences shown in Figure 4. Six sequences (i—vi) are shown. In CFP, miss-dependent instructions and the later miss-independent instructions execute out-of-order and during two phases: the independents execute while the dependents are waiting for the miss to complete (phase 1), and the dependents execute when the miss is serviced (phase 2). Stores in the SRL drain during the second phase. A horizontal solid line separates these two phases in the figure. Solid boxes denote instructions in the first phase that enter the SDB (Slice Data Buffer), and dotted boxes denote instructions in the second phase that execute from the SDB. The figures only show instructions executing in the pipeline and do not show cache updates of independent stores from the SRL. The first instruction in all sequences (LD-) is the long-latency cache miss. Because of memory dependence prediction, some independent load instructions, and their dependence chain may also become part of the miss slice instructions if they are predicted to depend on a store.

Case (i) shows an example of how the SRL algorithm handles a write-after-write hazard, case (ii) demonstrates the handling of a write-after-read hazard, and cases (iii) through (vi) show how read-after-write dependences are maintained.

4.2.1 Write-after-write hazard avoidance

In case (i), two stores occur to the same memory address, A, forming a write-after-write hazard. One store is miss-dependent (shown with a box) and a subsequent store is miss-independent. The miss-independent store, though after the miss-dependent store in program order, executes first, and temporarily updates the cache. Subsequent independent loads can access this data. When the miss returns, the temporary cache updates are discarded and dependents re-enter the pipeline and execute. The two stores (including the completed independent store) will update the cache through the SRL in program order, avoiding a write-after-write hazard.

4.2.2 Write-after-read hazard avoidance

In case (ii), a miss-independent store follows a miss-dependent load to the same address forming write-after-read hazard. The load drains out forming part of the slice, while the store updates the cache temporarily. When the miss returns, the update of the miss-independent store is discarded. The miss-dependent load on execution will then read the correct data from prior to the independent store, thereby avoiding a write-after-read hazard

4.2.3 Read-after-write hazard handling

Cases (iii) and (iv) show common sequences where the miss-dependent instructions and miss-independent instructions have no inter-dependences.

In (iii), the independent store (ST B) forwards to the independent load (LD B) in the first phase (either via the

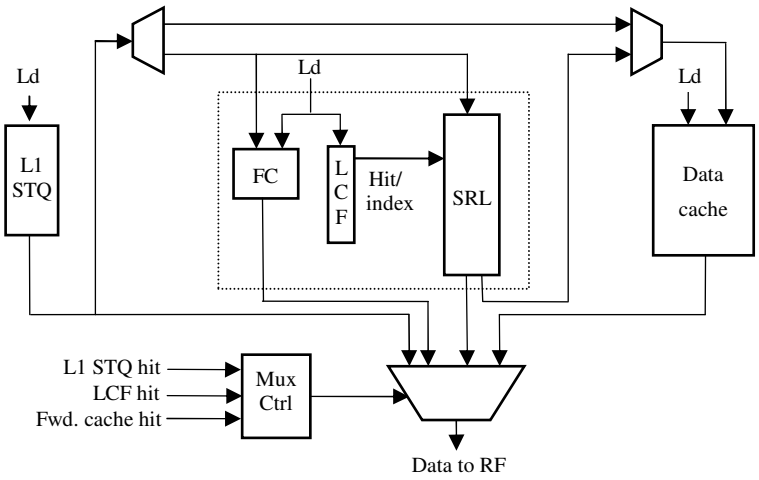

Figure 5 SRL algorithm implementation.

L1 STQ or through a temporary cache update). On re-insertion, of these two instructions, only the independent store is re-done.

In (iv), a dependence (shown by the solid arc) between a load (LD A) and a prior store (ST A) exists and is correctly predicted by the memory dependence predictor (shown by the dotted arc). No dependence exists between the independent store (ST B) and other instructions shown. On re-insertion of the waiting instructions (shown by boxes), the dependent load (LD A) will correctly forward from the prior store (ST A) either via the primary store queue or after the store has updated the cache (depending upon when the operations are scheduled).

Case (v) shows a situation where dependence via memory exists between a load (LD A) and a prior store (ST A). The store is miss-dependent and drains the pipeline waiting for the miss but the memory dependence predictor incorrectly predicts a lack of dependence. The load (LD A) is treated as an independent, executes, and enters the secondary load buffer. When the waiting instructions re-enter the pipeline and execute, the store (ST A) also executes. The store (ST A) looks up the secondary load buffer, detects the memory-dependence violation, and restarts from the checkpoint prior to LD A.

Case (vi) demonstrates a complex memory dependence violation sequence. Here, the memory dependence predictor predicts an independent load (LD A) to be dependent upon a miss-dependent store (ST B in the slice). The load however must receive data from the unboxed independent store (ST A). In phase 2, when the waiting instructions re-enter the pipeline and execute, depending upon the implementation, the boxed store (ST A) may go to the primary store queue and the load may incorrectly read the result of this store. When the unboxed ST A leaves the SRL and updates the cache, it looks up the load buffer, hits LD A, and detects the memory dependence violation. A restart from the LD A checkpoint follows. This demonstrates that in case any dependence violation (read-after-write) does indeed occur, the program order execution of all stores from the SRL will detect such a situation through the secondary load buffer.

Since the SRL algorithm can detect all violations, it will provide a correct execution under all situations, including similar cases not discussed above.

4.3 SRL implementation

Figure 5 shows the SRL algorithm implementation. The baseline data cache has support for our checkpoint architecture. Two additional bits, a speculative bit and a speculatively valid bit, per checkpoint are used for each cache block. The L1 data cache stores speculative state: the data in a block is not committed until its checkpoint has committed. When a checkpoint is committed, the corresponding speculatively valid and speculative bits in the cache are bulk cleared, marking those cache blocks as committed. A checkpoint squash requires a bulk clear of the speculatively valid bits. Only one version of a given cache block is allowed: stores from only one checkpoint can write a given cache block. Only one version of a given cache block is allowed: stores from only one checkpoint can write a given cache block; a store from another checkpoint to this cache block will stall and will prevent subsequent stores from the L1 STQ from draining.

In addition to the SRL queue, we have two new structures: a Forwarding Cache (FC) and a Loose Check Filter (LCF). These structures are small arrays and are low-complexity and simple performance enhancements to the SRL algorithm.

Forwarding Cache (FC): To avoid design changes to the L1 data cache, we use a separate small forwarding cache. Miss-independent stores update this cache and use it, instead of the data cache, to forward data to subsequent independent loads. Updates in this cache are discarded when the store redo occurs. As we discuss in the results

section, performing temporary updates to a small forwarding cache has design benefits over changing the data cache.

Loose Check Filter (LCF): During the store redo mode, some loads might need data from the SRL (to forward from an independent store). These loads might have been miss-dependent and have re-inserted from the SDB, or might be new loads from the front-end that issue during slice reinsertion. Since the SRL does not have a CAM, and because stores in the SRL might not have completely updated the cache when these loads issued, these loads would have to stall until the SRL drains completely. To avoid this stall, we use the LCF. The LCF is a direct-mapped non-tagged array of 6-bit counters. The LCF is indexed by hashing the memory address. Multiple addresses may map to the same counter. The LCF is based on a counting bloom filter [6], a variant of the Bloom filter allowing entries to be removed [2]. When a store enters the SRL, the LCF entry counter corresponding to its address is incremented. The counter is decremented when this store leaves the SRL. The LCF tracks whether an address has an entry in the SRL or not. A zero counter in the LCF entry corresponding to the load's address guarantees no store for the address exists in the SRL. This allows a load to safely issue. Counter overflows are handled by stalling store allocation in the SRL. We find the LCF effective in ensuring loads stall only when necessary.

For the small fraction of loads that stall because of a matching entry in the SRL, we add a limited form of forwarding to the SRL we call indexed forwarding. This still does not require a CAM or a search. The key idea for indexed forwarding is that most stalled loads that depend on stores in the SRL are newly fetched loads that entered the pipeline after the miss-dependent instructions were re-inserted but before the SRL stores have updated the cache. The store that forwards to such a load is the last matching store inserted into the SRL. We therefore store the index of the corresponding SRL entry with the LCF counter. When such a load issues, the SRL is indexed using the index stored with the LCF counter. The address and data from the SRL entry are read. A single comparator outside the SRL is used to perform a complete address and age check of the load against the store. This form of limited forwarding does not require any CAM or search.

In summary, our SRL algorithm implementation replaces the CAM and search circuitry of a conventional store queue with a FIFO structure (SRL), a direct-mapped array (LCF), and a small forwarding cache.

Algorithm overview: We now describe the algorithm in the context of Figure 5. The L1 STQ is a conventional small and fast store queue that provides forwarding capability for active instructions in the pipeline. The SRL is used only in the presence of a long latency cache miss, when the instruction window becomes large.

When a cache miss occurs, all subsequent miss-dependent instructions enter the SDB. All stores, dependent and independent, leave the L1 STQ in order, and enter the SRL. When leaving the L1 STQ, the miss-independent stores write their address and data value into the SRL. They also update the forwarding cache. Dependent stores have not completed and therefore do not write into the SRL. They however get an SRL entry allocated. This entry index is recorded with the miss-dependent store in the SDB.

When the cache miss returns, all updates in the forwarding cache are discarded. When the miss-dependent instructions execute, the stores re-inserted from the SDB re-allocate L1 STQ entries. When these stores complete, they leave the L1 STQ and update their SRL entry (the index of which they had recorded when they entered the SDB). The SRL thus holds, in program order, all store data in the shadow of the cache miss, and subsequently updates the data cache in program order.

To ensure that stores maintain correct ordering with prior loads (write-after-read dependency), the store at the head of the SRL updates the cache (during redo mode) only after all prior loads have completed execution. This is tracked using a bit-array, implemented with head and tail pointers. Every load and store gets an entry in the bit array (in program order), but only loads set and clear the bit. A load's bit in the array is set at allocate and is cleared when it completes execution. The load increments the head pointer when its bit is cleared. A store that is at the head of the bit array can be sure all prior loads have executed. We find that delaying store updates until loads have executed did not hurt performance.

Table 1 Baseline processor model.

Processor frequency	8 GHz
Rename/issue/retire width	4/6/4
Branch mispred. penalty	Minimum 20 cycles
Scheduling window size	64 Int, 64 FP, 32 Mem
Map table checkpoints	8
Register file	192 int, 192 fp.
Store buffer size	48
Load buffer	1K entries
Memory dependence pred.	Store sets
Functional units	Pentium 4 equivalent
Branch predictor	Gshare-perceptron hybrid 64K gshare, 256 perceptron
Hardware data prefetcher	Stream-based (16 streams)
Trace cache	4-wide
L1 Data cache	32 KB, 3 cycles
L2 Unified cache	1MB, 8 cycles
L1/L2 line size	64 bytes
Memory lat (Req to use)	100 ns

5. Experimental Methodology

We use a detailed execution driven timing simulator for simulating the IA32 instruction set and micro-ops. The

Table 2 Benchmark suite.

Suite	# of Bench	Desc./Examples
SPECFP2K (SFP2K)	13	www.spec.org
SPECINT2K(SINT2K)	10	www.spec.org
Internet (WEB)	10	SPECJbb, WebMark
Multimedia (MM)	14	MPEG, speech, photoshop
Productivity (PROD)	7	SYSMark2k, Winstone
Server (SERVER)	7	TPC-C
Workstation (WS)	13	CAD, rendering

simulator executes both user and kernel instructions, models all system activity such as DMA traffic and system interrupts, and models a detailed memory subsystem.

Table 1 shows parameters of our baseline Continual Flow Pipeline [17] processor with Checkpoint Processing and Recovery [1]. Table 2 lists the simulated benchmark suites, and the number of unique benchmarks within each suite. Unless specified, all performance numbers in graphs in the paper are percent speedups calculated over the baseline processor.

6. Results

Section 6.1 presents a performance comparison of the SRL proposal with a hierarchical store queue proposal and an ideal store queue and provides SRL statistics. Section 6.2 presents a power and area comparison of SRL and the hierarchical store queue. Section 6.3 evaluates the benefit of LCF and indexed forwarding. Section 6.4 presents the effect of LCF size and LCF hashing function on SRL performance, and Section 6.5 presents the performance benefit of using a forwarding cache instead of the data cache for temporary updates.

6.1 SRL performance

Figure 6 shows a performance comparison of SRL, the hierarchical store queue, and an ideal store queue. The

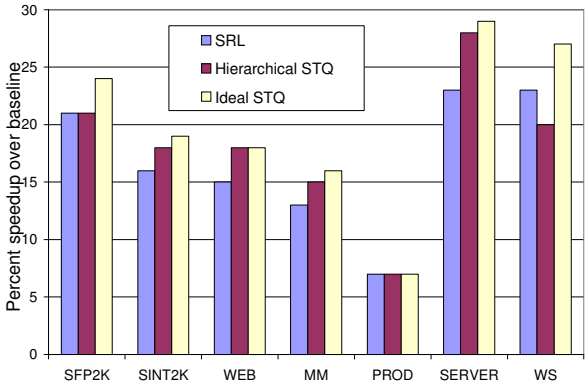

Figure 6 SRL performance comparison.

Table 3 SRL statistics.

	Redone Stores (%)	Miss-dependent Stores (%)	Miss-dependent Uops (%)	SRL Load Stalls/ 10000 uops	% execution time SRL is occupied
SFP2K	47.6	26.7	16.4	11	49.1
SINT2K	7.3	1.3	2.2	5	16.5
WEB	1.9	0.6	4.9	9	21.8
MM	6	2.7	6.5	6	18.3
PROD	0.3	0.1	0.4	1	5.7
SERVER	4.2	1.1	7.5	17	41.7
WS	9.4	8.5	2.6	3	13.9

SRL parameters are 48-entry L1 STQ with a 3-cycle access, a 1K-entry FIFO SRL, a 2K-entry LCF, and a 256-entry 4-way associative forwarding cache. The hierarchical store queue parameters 48-entry L1 STQ with a 3-cycle access, a 1K-entry L2 STQ with an 8-cycle access, and a 1K-entry Membership Test Buffer). The ideal store queue is a 1K-entry L1 STQ with a 3-cycle access. The y-axis is the percent speedup over a baseline with a 48-entry store queue. (A large store queue is necessary for a latency tolerant processor to sustain a large instruction window and the store queue alone does not result in high performance).

We see that SRL provides performance competitive to the hierarchical store queue, and outperforms it for the WS benchmark suite. This occurs because for stores missing the L1 STQ, SRL forwarding occurs at the L1 cache hit latency, while forwarding from the L2 STQ for the hierarchical solution occurs at the L2 cache hit latency. The hierarchical solution performs better than the SRL proposal for four suites (SINT2K, WEB, MM and SERVER) because the hierarchical solution has a full CAM structure and forwarding capability, unlike the SRL where only the last store mapping to an LCF entry can forward using indexed forwarding.

With SRL, some loads end up stalling even with indexed forwarding. However, SRL does not require the CAM structures or search required in the hierarchical solution or the ideal solution. It achieves performance within 6% of an ideal store queue implementation.

Table 3 shows key SRL statistics. The floating-point benchmarks (SFP2K) have high cache miss ratios and longer dependence chains and hence have a large fraction of stores in the shadow of cache misses, which can be seen from the percentage of stores redone (column 2). These benchmarks also have more miss-dependent stores (column 3) and more miss-dependent instructions (column 4) compared to other benchmarks. The restricted forwarding capability in SRL causes a small fraction of loads to stall (column 5) across all benchmarks.

Figure 7 shows the SRL occupancy distribution for the fraction of time the SRL is occupied. Stores enter the SRL

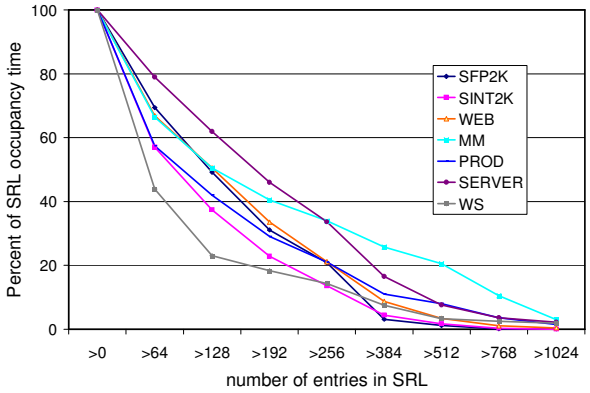

Figure 7 SRL occupancy distribution during the time the SRL is occupied.

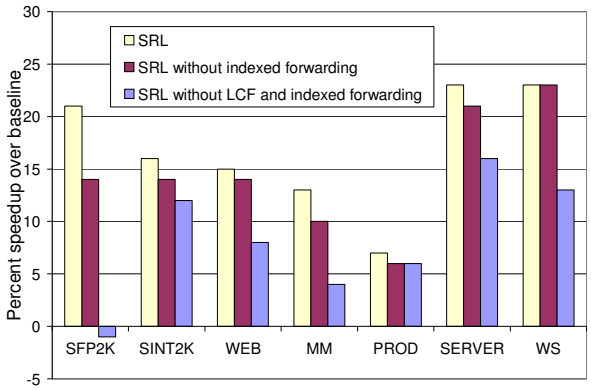

Figure 8 Impact of LCF and forwarding in SRL.

only in the presence of missed loads and Table 3's column 6 shows the percentage of execution time the SRL is occupied. For all benchmarks, we find a 1K entry SRL is sufficient to hold all stores in the shadow of a load miss.

6.2 Power and area analysis

In this section, we compare the power and area of our SRL proposal with that of the hierarchical store queue [1]. As mentioned earlier, the hierarchical store queue has a full CAM L2 store queue. We do not study the data portions of the store queues since that will be common to all proposals.

Hierarchical L2 STQ power and area: We designed a 512-entry store queue (the L2 STQ) with an 8 cycle access latency using a 90nm CMOS technology [12]. The design was optimized to reduce leakage power. A banked structure was used to reduce dynamic power. The CAM array entry has 36 address bits and 8 byte-mask bits (to determine address match when loads and stores have unaligned addresses or different data sizes). We performed SPICE simulations of our design and calculated the dynamic and leakage power of the L2 STQ. The area of the L2 STQ was 1.4 mm^2. Using SPICE simulations, the leakage power came to 95mW and the dynamic power (if every load looked up the L2 STQ) came to 4.4W. Since only 10% of loads look up the L2 STQ in the hierarchical design, the total dynamic power of the 512-entry L2 STQ if 440mW.

Store Redo Log power and area: We also designed the address array of a 512-entry SRL with a 2K-entry LCF. Each entry in the SRL queue consists of 6 bytes of address, and thus the total SRL queue size is 3KBytes. Each entry in the LCF consists of 10-bit SRL queue index and 6 bits count value, for a total of 2 bytes per entry. The total size of the LCF is therefore 4KBytes. The combined size of the SRL and LCF is 7K bytes. The area of the SRL queue was 0.35mm^2. The dynamic power was 30mW and the leakage power was 40mW. Adding the separate forwarding cache to the SRL has minimal impact on these parameters. The SRL with the forwarding cache has an area of 0.45mm^2, a leakage power of 48mW, and a dynamic power of 37mW.

Comparing to the area and power estimates for a hierarchical store queue design presented earlier, the SRL algorithm gives significant reduction in area and power over a highly optimized hierarchical store queue circuit that uses best current techniques for minimizing CAM and search activity to reduce dynamic power.

6.3 LCF and SRL indexed forwarding

The results presented thus far assume an SRL implementation with a loose-check filter (LCF) and indexed forwarding (Section 4.3) to prevent unnecessary load stalls. In this section, we study the performance impact of LCF and indexed forwarding.

Figure 8 compares the performance of SRL without LCF, SRL with LCF but without indexed forwarding, and SRL with LCF augmented with indexed forwarding. We use a 2K-entry LCF for this experiment. Lack of LCF affects SFP2K benchmarks the most since they have a large number of cache misses and miss dependent stores occupying the SRL. Adding an LCF significantly cuts down on loads stalling due to potential matches in the SRL resulting in more than 15% performance gain for SFP2K. Further adding indexed forwarding support to the SRL (which still does not require a CAM match) results in higher performance. Thus, the LCF and indexed forwarding play an important role in achieving high performance.

6.4 LCF size and LCF hashing function

Figure 9 presents the sensitivity of SRL performance to the LCF size and hashing function. Two different LCF hashing functions are studied: Lower Address Bits indexing (LAB), and 3-Piece Address XOR indexing (3-PAX), which computes an index from the XOR of the lower, middle and upper address bits. For each hash function, results for 256-entry and 2K-entry LCF sizes are pre-

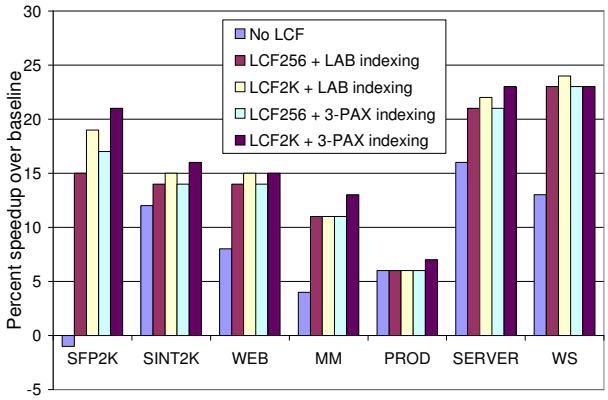

Figure 9 LCF sizes and hashing function impact on SRL performance.

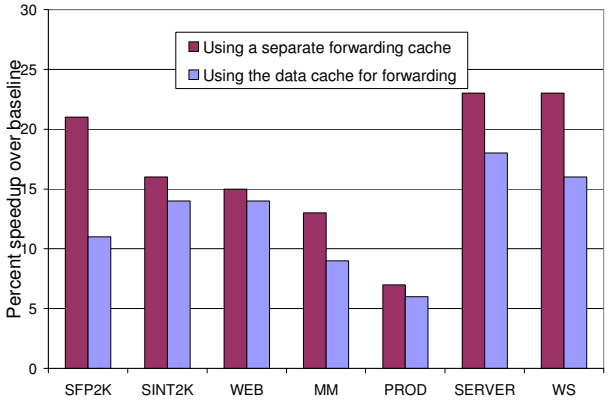

Figure 10 Forwarding design option impact.

sented, and performance without LCF is shown for reference. Results presented elsewhere in this paper are for a 2K-entry LCF and a 3-PAX hashing scheme. From Figure 9 we see little sensitivity to hash function and greater sensitivity to LCF size, especially for the SFP2K benchmark suite. A 256-entry LCF performs within 2% of a 2K-entry LCF and significantly better than without an LCF, across all benchmarks. While the performance difference between the two hashing schemes is small, the 3-PAX hashing significantly benefits several individual benchmarks.

6.5 Data cache for temporary updates

We have avoided using the data cache to buffer temporary updates in our SRL implementation, and employ a small forwarding cache. While doing so has the advantage of not having to change the data cache design, it also has performance benefits. This is because if we use the data cache for buffering temporary updates for forwarding from independent stores to independent loads, any earlier modified data for the block being written must be written back to the next level of cache. This adds latency. Further, the data cache may suffer from associativity limitations, which may stall store processing. Because temporary updates are discarded in the data cache, a load may observe an additional miss to the next level of cache during the redo phase.

Figure 10 compares the performance of SRL with and without a 256-entry 4-way associative forwarding cache and shows that a forwarding cache significantly helps improve performance.

7. Related Work

Sethumadhavan et al. [16] propose filtering schemes using Bloom filters to reduce both the number of LSQ lookups and the number of entries to be searched during each lookup. They also use address predictors to reduce the number of loads tracked in the LSQ, leading to a smaller LSQ. Park et al. [15] reduce search bandwidth demand on the store queue using a store-sets predictor that prevents loads predicted to be independent of prior stores from looking up the store queue. They present ways to reduce the search bandwidth demand on the load queue and explore a segmented load/store queue (LSQ) with pipelined, variable-latency segments. Akkary et al. [1] propose a hierarchical store queue consisting of a small, fast L1 STQ, and a large, slow L2 STQ, combined with a fast filtering mechanism, the Membership Test Buffer to reduce L2 STQ lookups.

The above three schemes reduce the search bandwidth requirements of large store queues and/or reduce the number of loads that need to be tracked to maintain proper memory ordering. Our load buffer algorithm and store redo approach avoid having to search large load and store queues, thus providing significant savings in area and power.

Gharachorloo et al. [8] use speculative execution to improve the performance of memory consistency models, and suggest using value matching to detect consistency violations. Gniady et al. [9] use speculative execution and a hardware log of speculative updates to processor and memory state for every instruction for improving the performance of sequential consistency. Their implementation rolls back to the "sequentially consistent" memory state if a violation is about to occur. Cain and Lipasti [3] propose a value-based replay mechanism for enforcing uniprocessor and multiprocessor ordering constraints that eliminate the need for associative look-ups in the load queue. Our load buffer eliminates the fully associative lookup of the load queue as well and does not need to store load data since it does not need a replay mechanism.

Proposals for storing speculative versions of data and for detecting memory dependence violations between tasks/threads in the context of speculative multithreading exist [7, 10] and are orthogonal to the proposals in this paper.

8. Conclusions

We have presented a new proposal for processing loads and stores in very large instruction window, latency-tolerant processors. The proposal does not require address CAM and search of the secondary load and store buffers. The secondary load buffer is a set associative buffer with store identifiers for determining load and store order and checkpoint bits with bulk reset mechanism. The key idea behind the store queue proposal is that redoing stores in the shadow of load misses, after the miss data is fetched from memory, to fix memory dependences provides better power and area characteristics than a scheme which constantly enforces dependences among a very large number of loads and stores, many of which have unknown addresses. Our proposal performs competitively compared to a very large conventional store queue while significantly reducing the area and power of the load and store address queues. The reductions are primarily because our proposal eliminates the requirement for search and CAM of large load and store queue structures.

Acknowledgements

We thank Saurabh Dighe, Shih-Lien Lu, and Dinesh Somasekhar for discussions on low-power circuit design techniques and for help with the circuit design and simulation tools.

References

[1] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. In *Proceedings of the 36th International Symposium on Microarchitecture*, December 2003.

[2] B. H. Bloom. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7), July 1970.

[3] H. W. Cain and M. H. Lipasti. Memory Ordering: A Value-Based Approach. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, June 2004.

[4] G. Z. Chrysos and J. S. Emer. Memory Dependence Prediction Using Store Sets. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, June 1998.

[5] A. Cristal, D. Ortega, J. Llosa, and M. Valero. Out-of-Order Commit Processors. In *Proceedings of the Tenth International Symposium on High-Performance Computer Architecture*, February 2004.

[6] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. *IEEE/ACM Transactions on Networks*, 8(3), 2000.

[7] M. Franklin and G. S. Sohi. A Hardware Mechanism for Dynamic Reordering of Memory References. *IEEE Transactions on Computers*, 45(5), May 1996.

[8] K. Gharachorloo, A. Gupta, and J. L. Hennessy. Two Techniques to Enhance the Performance of Memory Consistency Models. In *Proceedings of the 1991 International Conference on Parallel Processing*, August 1991.

[9] C. Gniady, B. Falsafi, and T. N. Vijaykumar. Is SC + ILP = Rc? In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, May 1999.

[10] S. Gopal, T. N. Vijaykumar, J. E. Smith, and G. S. Sohi. Speculative Versioning Cache. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, February 1998.

[11] T. Karkhanis and J. E. Smith. A Day in the Life of a Data Cache Miss. In *Workshop on Memory Performance Issues*, June 2002.

[12] K. Kuhn, M. Agostinelli, S. Ahmed, S. Chambers, S. Cea, S. Christensen, P. Fischer, J. Gong, C. Kardas, T. Letson, L. Henning, A. Murthy, H. Muthali, B. Obradovic, P. Packan, S. W. Pae, I. Post, S. Putna, K. Raol, A. Roskowski, R. Soman, T. Thomas, P. Vandervoorn, M. Weiss, and I. Young, A 90 nm Communication Technology Featuring SiGe HBT Transistors, RF CMOS, Precision R-L-C RF Elements and 1 um2 6-T SRAM Cell. presented at *IEEE International Electron Devices Meeting*, December 2002

[13] A. R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. A Large, Fast Instruction Window for Tolerating Cache Misses. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, May 2002.

[14] A. Moshovos and G. S. Sohi. Streamlining Inter-Operation Memory Communication Via Data Dependence Prediction. In *Proceedings of the 30th International Symposium on Microarchitecture*, December 1997.

[15] I. Park, C. L. Ooi, and T. N. Vijaykumar. Reducing Design Complexity of the Load/Store Queue. In *Proceedings of the 36th International Symposium on Microarchitecture*, December 2003.

[16] S. Sethumadhavan, R. Desikan, D. Burger, C. R. Moore, and S. W. Keckler. Scalable Hardware Memory Disambiguation for High ILP Processors. In *Proceedings of the 36th International Symposium on Microarchitecture*, December 2003.

[17] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual Flow Pipelines. In *Proceedings of the Eleventh Symposium on Architectural Support for Programming Languages and Operating Systems*, October 2004.