Partition the Banks, not the Functionality, of Large-Window Load/Store Queues

Simha Sethumadhavan Doug Burger Stephen W. Keckler
Computer Architecture and Technology Laboratory
Department of Computer Sciences
The University of Texas at Austin
cart@cs.utexas.edu - www.cs.utexas.edu/users/cart
Technical report number: TR-06-39 - Aug 2006

Abstract

Designing scalable memory ordering hardware is one of the most important challenges for large-window, out-of-order processor design, due to its complexity, power, and its criticality for high performance. Recent research has aimed to partition the functionality of load/store queues (LSQ) into three components: ordering violations detection, value forwarding, and store buffering for commit, to avoid frequent access to large, associative, energy-inefficient structures. This approach adds microarchitectural complexity but has been shown to be effective. In this paper, we describe a family of energy-efficient distributed load/store queue designs that avoid the need for partitioning the LSQ functionality, but which achieve comparable energy efficiency with performance comparable to an ideal LSQ. These designs interleave LSQ banks based on cache-line addresses, and deal with the resultant overflow challenges by prioritizing older instructions and occasionally rejecting younger instructions using flow-control techniques. The experimental results show that on average, there is no performance degradation for address-interleaved LSQs that are undersized *up to three-quarters of a 256 memory instruction window for both SPEC and EEMBC benchmarks. In addition, each LSQ access consumes only as much energy, on average, as a 8-entry, fully associative traditional LSQ.*

1 Introduction

The load/store queues (LSQs) that provide hardware memory disambiguation and ordering have become some of the most challenging structures to design in modern, high-ILP processors. For processors that aim to exploit large–and growing–instruction windows, the LSQs can incur significant power, area, performance, and complexity overheads. A recent approach by numerous researchers has been to partition the functionality of LSQs into their three constituent functions: *value forwarding* from stores to loads, *ordering-violation detection*, and *store-value buffering* for commit [3, 7, 14, 11, 9]. By breaking up the LSQ into various combinations of these three functions, researchers have been able to size each structure appropriately for its function, reducing energy consumption as window sizes grow. This strategy has been shown to be effective, but suffers from two disadvantages: the added complexity of multiple interacting structures, and the physical centralization of each component.

For large-window, wide-issue processors, primary memory systems will become increasingly partitioned, due to both increased bandwidth requirements [10] and due to increasing communication delays [1]. Some of the recent LSQ proposals that separate the functionality into multiple structures will likely become significantly more complex when distributed among many primary memory system banks. Ideally, future LSQs will fulfill two requirements. First, they will be distributed along with the level-one data caches with the same interleaving function, allowing a load or store to be routed to a single partition of the primary memory system, where the cache lookup and memory ordering can be performed locally. Second, future LSQs should require accesses to only comparatively small, energy-efficient structures, that do not become more complex even as instruction window sizes grow.

In this paper, we describe a family of LSQ designs which are broken into address-interleaved banks, partitioning based on address rather than functionality. The classic problem with address-interleaved LSQ banks is bank overflow; since the mapping of load/store banks to address is dynamically determined, too many loads and stores may map to one LSQ bank. The classic technique for dealing with such structural hazards to flush the pipeline, which can cause too much performance degradation in many cases.

The key idea in this family of designs is to provide low-overhead LSQ bank overflow handling. Each design partitions the in-flight memory operations into age-ordered bins. For example, a window supporting up to 256 memory operations in flight might assign each consecutive 32 loads or stores to one bin, permitting up to eight bins total. The oldest bin is the *high-priority* bin. The LSQ banks handle loads or stores from the high-priority bin differently from low-priority bins. These designs apply a range of flow-control techniques to the low-priority operations (LPs), including interlocking them using virtual channels, or NACKing them and sending them back to the issue window. High-priority operations (HPs) are either provided reserved space in each LSQ bank, or, on a high-priority overflow, cause a rare pipeline flush followed by refetching of the HP operations and a temporary throttling of subsequent operations.

We use the TRIPS architecture for a large-window substrate to evaluate these ideas. The scalar operand networks [13], which exist in the TRIPS design as well as others [15, 12] are useful for implementing the flow-control policies (NACK and virtual channels). We show that the virtual-channel approach to flow control demonstrates a negligible (2%) performance loss over an idealized distributed LSQ that never overflows, while requiring each load or store to compare against 8 addresses, on average, for a window size of 256 memory operations and 1024 instructions. These results show that partitioning based on functionality is unnecessary, particularly if the primary memory system is distributed.

In the next section, we discuss related approaches, including prior distributed LSQ work as well as the recent

body of work on functionality-partitioned LSQs. Section 3 describes the various implementations, including the underlying microarchitecture, the different flow control mechanisms and policies for managing LSQ overflow conditions, and the physical design of the LSQ bank, which must be able to handle unordered loads/stores due to the address interleaving of LSQ banks. In Section 5, we analyze the performance of various approaches, as well as analyzing the scalability of this approach to a finer degree of partitioning. Finally, we conclude in Section 6.

2 Related Work

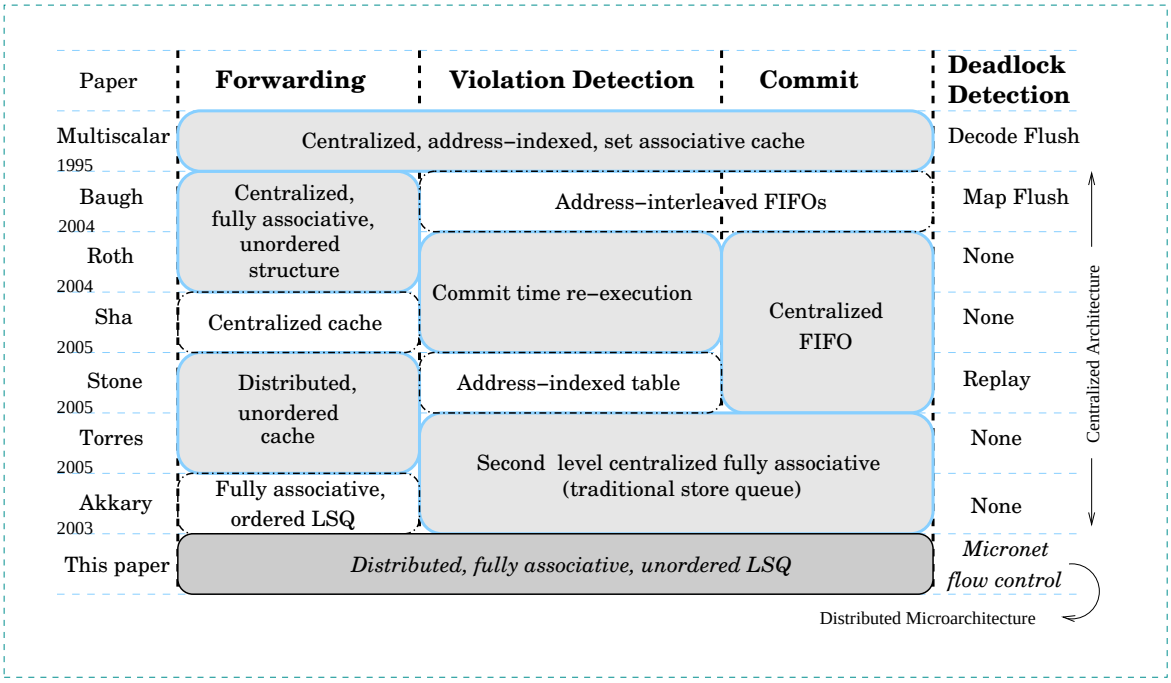


Figure 1: LSQ Related Work.

An LSQ fulfills three necessary functions: (1) it forwards store values to later loads when their addresses match, (2) it detects load misspeculations, if a load issued with older unresolved stores in flight that were later found to match the load's address, and (3) it acts as a store buffer that is used to commit stores to memory upon retirement.

Recent work (see Figure 1) has proposed seperating some or all of the three LSQ functions into separate structures, to accelerate some of the individual structures. This approach results in increased complexity, and occasional area increases due to redundant information held in multiple structures, but the potential power savings, and in some cases, the access latency improvements, are considerable. In this paper, we propose a LSQ organization that provides all the benefits of functional decomposition without the increased complexity of the other mechanisms.

Both Baugh and Zilles [3] and Roth [7] use a small, fast, centralized, unordered, fully associative forwarding buffer and a non-associative FIFO for buffering and committing stores. The papers differ on the misspeculation detection function: Baugh and Zilles use a centralized, addressed-indexed structure, whereas Roth uses enhancements of a load re-execution proposed by Cain and Lipasti [5]. The scalability and applicability of these methods to distributed architectures has not been discussed.

Sha et al. [9] extend Roth's scheme by using a modified dependence predictor to match loads with the precise store buffer slots from which they are likely to receive forwarded data. This solution completely eliminates the associative store forwarding buffer but instead requires large multi-ported dependence/delay predictors (approximately 16KB combined), thus effectively improving power at the expense of area.

Stone et al. [11] use a set associative cache for forwarding, a non associative FIFO for commit, and an address-indexed timestamp table for checking speculation. In addition to the added complexity of this scheme, the timestamp table may generate signal extra mis-speculations due to address aliases and conservative handling of partial flushes. Since the authors have no means to enforce low overhead flow control for instructions targeting the LSQ, their address-indexed structures must be heavily oversized for good performance. For instance, the authors use an 8K entry, 8-bit wide timestamp table and a 80-bit wide, 512 set, 2-way forwarding cache even though there can be only 1K instructions in flight at any time.

Torres et al. [14] propose a distributed, unordered, address-interleaved store-load forwarding buffer but a centralized, age-ordered store queue for speculation checking and commit. While the distributed store forwarding buffers increase forwarding bandwidth, the centralization of the other two structures forces all loads and stores to be routed to a central place for verification and commit, hampering scalability.

Akkary et al. [2] propose two-level store buffers which are both centralized, fully associative and age ordered. Stores are first entered in the L1 store buffer, and when it overflows they are moved to the L2 store buffer. Both buffers support forwarding and speculation checking but stores commit from the second level buffer. This scheme reduces power, but still requires a worst-case sized L2 and uses area-inefficient CAMs.

It should be noted here that there have been no comprehensive studies evaluating different strategies for recovering from LSQ bank overflow for distributed microarchitectures. While many of the above papers have referenced the traditional set of possibilities: stalling fetch or stalling issue to prevent overflows, flushing on an overflow and incorporating a mechanism (usually serialized execution) to guarantee forward progress and replaying the overflowed instruction from the issue window, they do not address efficient ways to deal with overflow for partitioned LSQs. In addition, the costs for each of the above mechanisms in distributed microarchitectures could be higher than centralized designs because of increased distance between the execution and the memory units.

The work presented in this paper shows how to leverage the interconnection network flow control techniques to achieve similar levels of performance without oversizing the buffers at all, merely by dividing the LSQ into banks.

3 Distributed LSQ Microarchitecture

In a hypothetical age-indexed and partitioned LSQ, load and store instructions could be directed to different LSQ banks using memory sequence numbers. This approach naturally load balances the memory instructions across partitions, without requiring any extra entries in the partitions, and guaranteeing no overflows. However, this organization would defeat the original purpose of partitioning the LSQ, which is to have LSQ banks tightly coupled with the data cache banks that will receive their loads and stores. Distributing age-interleaved partitions to address-interleaved caches will effectively randomize the communication among LSQ and cache banks.

Address-interleaved LSQs are a much better match for address-partitioned memory systems, but the mapping of loads and stores to banks is unknown until the instructions execute. In the worst case, all in-flight memory instructions might map to the same partition. This behavior is uncommon in many benchmarks, but may arise when the application is loading from or storing to a sequential array of characters. Even so, sizing each partition for the worst case results in a total LSQ capacity that–for N banks–is N times the memory instruction window size. If the partitioned LSQ banks are sufficiently undersized that overflows are not an extremely rare event, a graceful, low-overhead mechanism for handling overflows is necessary. This section first describes the microarchitecture trends and design principles of a partitioned architectures that necessitate a new approach to LSQ design. It then discusses the microarchitecture of an unordered LSQ design amenable to address interleaving.

3.1 Partitioned Microarchitecture

The architectural trends motivating the design of a partitioned LSQ include (1) very large instruction windows with hundreds of in-flight memory instructions, and (2) partitioning of microarchitectures for scaling to higher instruction execution and local memory bandwidth. While recent literature has many examples of these trends, our mechanisms are built on top of the TRIPS microarchitecture. The TRIPS processor is a partitioned microarchitecture that enables a window of up to 1024 instructions and up to 256 simultaneously executing memory instructions. All major components of the processor are partitioned and distributed including instruction fetch, instruction issue, and memory access.

The processor itself is composed of an array of 16 execution units connected via a routed operand network.

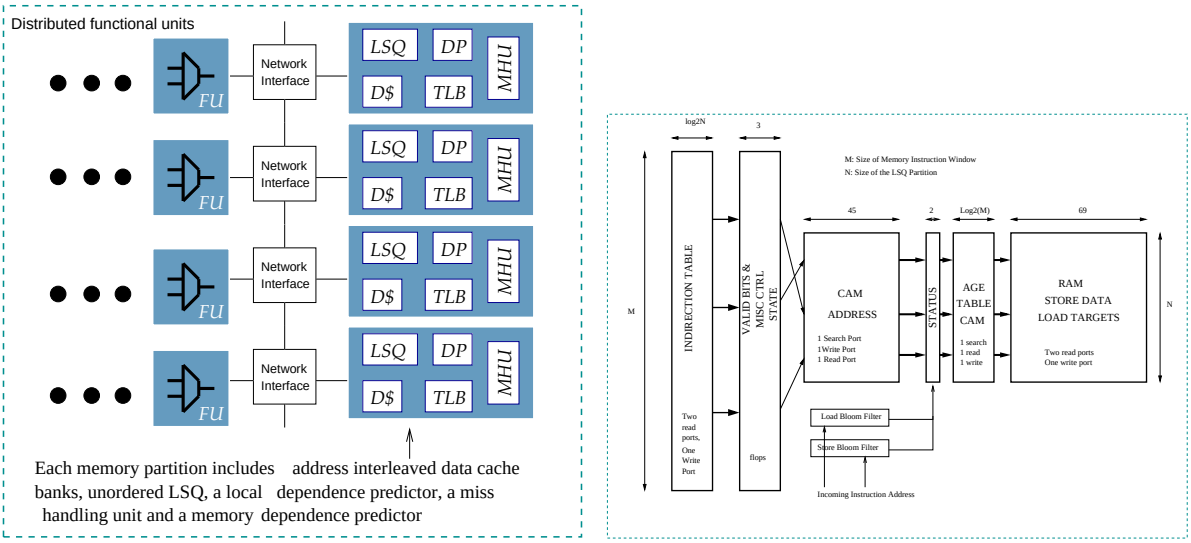
Instructions are striped across 4 instruction cache banks which are accessed in parallel to fetch TRIPS instruction blocks. Instructions are delivered to the execution units where each instruction waits until its operands arrive. The primary memory system (level-1 data cache, LSQs, dependence predictors and miss handling units) is divided into multiple banks which are also attached to the routed operand network. Cache lines are interleaved across the banks, which enables up to 4 memory instructions per cycle to enter the level-1 cache pipelines. Figure 2(a) shows a highly abstracted view of the parts of the TRIPS microarchitecture that are relevant to memory instructions; additional details about the TRIPS architecture can be found in [4].

The features of the TRIPS architecture most relevant to the design of LSQs can be distilled down to a few principles which are not unique to TRIPS. First, is distributed cache in which multiple level-1 cache banks must independently preserve the proper ordering of load and store instructions. Second, is the set of distributed execution units which independently decide which instructions to issue each cycle. Third, is distributed instruction fetch which provides higher instruction fetch bandwidth but does not easily allow age tags to be assigned to memory instructions in fetch order. Finally, a distributed architecture with multiple execution and memory units must include some form of interconnection network. TRIPS employs a mesh-routed operand network which can be augmented to provide multiple virtual channels. Some of the solutions outlined in this paper rely on the network buffers and flow control to store in-flight memory instruction packets. However, other interconnection networks are feasible and they could likewise be augmented with additional queueing to buffer in-flight memory instructions. The bottom line is that while we examine partitioned LSQ design in a TRIPS context, we believe that these concepts apply to other architectures that share some of these basic characteristics.

3.2 Unordered LSQ design

In a traditional age-indexed LSQ, the instructions in the LSQ are stored in a structure that is indexed by instruction's memory sequence number. An age-indexed LSQ uses the implicit age ordering to optimize the LSQ search functions and to quickly allocate and deallocate entries. However, for address interleaved LSQs, the current age-indexing techniques cannot be used without sizing the LSQ to match the memory instruction window size (maximally sized LSQ). In this section, we provide a overview of techniques that allow age-indexing to work with undersized address interleaved LSQs. Figure **??** illustrates the undersized age-indexing LSQ design.

The basic idea is to manage the undersized LSQ as a free-list; as and when memory instructions arrive at a partition, a slot is allocated from a pool of free LSQ entries, slots are returned to the LSQ when entries are deallocated either on flush or commits. Using this allocation policy results in an LSQ where a slot allocated to the instruction has no relation to the instruction sequence number (unordered LSQ) and for this type of LSQ,



(a) MicroArchitecture (b) Unordered LSQ

Figure 2: Microarchitecture Illustrations: (L) Abstract view (R) Unordered LSQ

without any additional structures, the LSQ functions may take multiple cycles to complete. For instance, during store commits, all entries in the unordered LSQ have to be scanned, one every cycle, to detect the oldest store to commit and this can result in longer commit latencies which are known to negatively impact performance.

The commit operations in the unordered LSQ can be optimized by creating a table that is indexed by the age of the instruction and holds the slot allocated to a instruction in the LSQ (Indirection table or INT). By using this table, when stores are ready to commit, the LSQ slot with the store data can be looked up in the indirection table with the age of the store.

The second function of the LSQ — store forwarding — is more involved to implement in an unordered LSQ and can result in performance penalties because without the ordering information a more extensive search involving *all matching entries* in required in the LSQ. The process of store forwarding in the unordered LSQ works as follows: An incoming load instruction performs associative search on the address CAM to identify all matching store instructions. Then, the age of each matching entry is read out from the LSQ and compared against the age of the incoming load. If the matching store is older than the load, then the store should forward to the load; the store is marked as matching and noted for later processing by setting a bit corresponding to the store in in the matching vector (a 1-b wide array indexed by the store's age). Once all matching stores have been identified, the process of collecting the load data from the matching stores begins by scanning the matching vector backwards in age order. Then, when all of the load data has been received store forwarding is complete. Thus, the process of store forwarding with N matches takes at least $N + 1$ cycles to complete and at most $N + 9$

cycles to complete. In contrast in an age-indexed LSQ with N matching stores, forwarding can take at most 8 cycles (or the maximum datum size.)

Two optimizations can improve the performance of the unordered LSQ: first, instead of scanning the array and reading out the ages of the matching instruction, a special age CAM (AT-CAM)[1] can be utilized to identify older stores instructions Second, the load can start scanning for matching stores starting from stores that belong to the youngest branch before the incoming load and processing the stores in the branch order. The results of both these optimizations are presented in the results section.

Detecting violations and handling flushes are fairly straightforward. If the incoming instruction is a store, the associative search identifies matching addresses and younger loads. A violation is triggered if there are any matching younger loads. For flushes, when a LSQ identifies all instructions to be flushed by searching the AT-CAM with the age of the instruction to be flushed.

Since the LSQ is undersized, special hardware is required to detect and report overflow conditions. A simple counting mechanism with $N + 1$ counters is used for this purpose (N is the maximum number of unresolved branches inflight). N of these counters hold the count of number of memory instructions that have arrived from each inflight branch while the remaining counter keeps a cumulative count. When the cumulative count reaches a the total size of the LSQ an overflow is signalled. On deallocation, the number of memory instructions corresponding to the flushed branch is subtracted from the cumulative count. In case of the TRIPS processor, N=8 and corresponds to the total number of inflight blocks.

To decrease the power consumption of the LSQ we use separate load and store Bloom filters (BFs) with flash clearing [8]. BFs reduce the power hungry associative lookup with power efficient direct mapped lookup in the common case. Seperate load and store BFs are associated with each inflight branch to efficiently implement flash clearing. If a memory instruction is between inflight branch b_0 and b_1 then the memory instruction is noted in the Bloom filter belonging to that branch. All the bits in the BF are flash cleared when the branch is resolved or branch is flushed and all instructions in the branch have committed. For the TRIPS processor, each inflight branch naturally maps to an inflight block.

[1]The AT-CAM is special variant of the regular CAM. Instead of outputting just equality matches the CAM outputs a greater/lesser/equal results. The same CAM will be later used for other optimizing flushes also.

4 Mitigating LSQ Overflows

The microarchitecture of the unordered LSQ in the previous section is ideal for an address-interleaved cache architecture, it must function correctly even if all of the in-flight memory instructions are sent to a single partition. A brute-force approach in which the LSQ tables are maximally sized will result in area and power overheads in the common case. This section examines several techniques for undersizing the LSQ tables in each partition, while still maintaining correctness and deadlock freedom.

The principal question when using an undersized LSQ is what to do when a memory instruction arrives at a full LSQ. An obvious (but flawed) mechanism to deal with overflows gracefully is to simply stall the load/store pipeline on an overflow until a slot in the LSQ partition becomes free. This approach does not work because of deadlocks; younger memory instructions can reach the LSQ ahead of the older instructions, filling up the LSQ and preventing the older instructions from reaching the queue and committing. A heavyweight alternative, commonly used in conventional processors to handle a variety of resource overflows, is to flush the pipeline and resume execution. In this case, the smaller the LSQ, the more frequent the flushes; if flushes are too frequent, performance will drop precipitously.

The goal then is to reduce the size of the LSQs as much as possible, without dramatically increasing the flush frequency. The remainder of this section examines mechanisms to buffer memory instructions in different parts of the system if they cannot be accepted by the LSQ. We examine three obvious places to buffer these instructions: in the execution units, in the memory units (but before the LSQ), or in the network connecting the execution units to the memory units. The buffering space is much less precious than the LSQ since the buffered locations need not be searched for memory conflicts, which mitigates the area and power overheads of employing more buffer storage.

These buffering approach effectively stall processing of certain memory instructions, which could potentially lead to deadlock. However, memory instructions can be formed into groups based on age, with all of the instructions in a group having similar ages. In a machine with a block-oriented instruction set such as TRIPS, the memory instruction groups correspond to the instruction blocks. One block is non-speculative, while multiple blocks can be speculative. By choosing to prioritize the non-speculative instructions over the speculative instructions, our solutions can reduce the circumstances for deadlocks and flushing. One possible design would reserve LSQ entries for the non-speculative block, but our experiments indicated that this approach did not provide any substantive performance benefits and resulted in larger than a minimum sized LSQ.

4.1 Memory Instruction Retry

One common alternative to flushing the pipeline in conventional processors is to replay individual offending instructions, either by retracting the instruction back into the issue window, or by logging the instruction in a retry buffer. In TRIPS retrying means sending an offending instruction back to the ALU where it was issued and storing it back into its designated reservation station. Since the reservation station still holds the instruction and its operands, only a short negative-acknowledgement (NACK) message needs to be sent back to the execution unit. No additional storage in the system is required as the reservation station cannot be reassigned to another instruction until the prior instruction commits. The issue logic may retry this instruction later according to a number of possible policies.

Figure 3a shows the basics of this technique applied to LSQ overflows. When a speculative instruction arrives at a full LSQ, the memory unit sends the NACK back to that instructions execution unit. This policy ensures that speculative instructions will not prevent a non-speculative instruction from reaching the LSQ. If a non-speculative instruction arrives at a full LSQ, then the pipeline must be flushed.

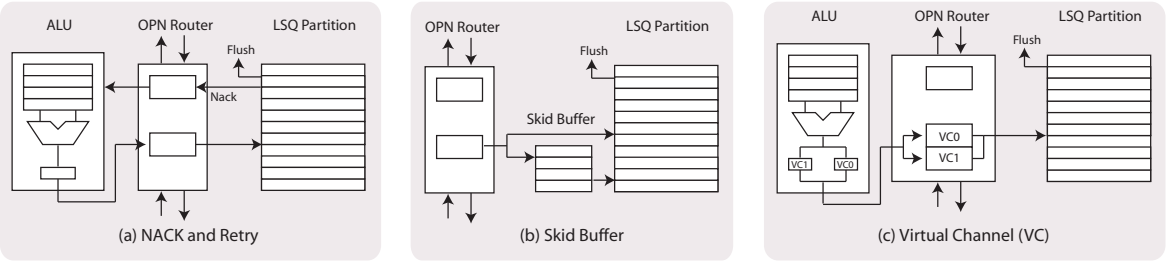


Figure 3: LSQ Flow Control Mechanisms.

A range of policies are possible for determining when to reissuing a NACKed memory instruction. If the instruction reissues too soon (i.e. immediately upon NACK), it can degrade performance by clogging the network, possibly requiring multiple NACKs for the same instruction. This can delay older non-speculative instructions from reaching the LSQ partition, as well as general execution and instructions headed to other LSQ partitions. Alternatively, the reservation stations can hold back NACKed instructions until a fixed amount of time has elapsed. Waiting for a set amount of time requires a counter per NACKed instruction, and may be either too long (incurring unnecessary latency) or too short (increasing network contention).

Instead, our approach triggers re-issue when the non-speculative block commits, which has the desirable property that LSQ entries in the overflowed partition are likely to have been freed. This mechanism does have two extra overheads, however: an additional state bit for every reservation station is required, to indicate that the

instruction is ready but waiting for a block to commit before reissuing, and a control path to wake up NACKed instructions when the commit signal for the non-speculative block arrives.

4.2 Skid Buffers

A second technique is to store memory instructions waiting to access the LSQ in a skid buffer located in the memory unit. As shown in Figure 3b, the skid buffer is simply a FIFO into which memory instructions can be inserted and extracted. To avoid deadlock, our skid buffers only hold speculative memory instructions. If an arriving speculative memory instruction find the LSQ full, it is inserted into the skid buffer. If the skid buffer is also full, the block is flushed. Arriving non-speculative instructions are not placed in the skid buffer. If they find the LSQ full, they trigger a flush.

When the non-speculative block commits and the next oldest block becomes non-speculative, all of its instructions that are located in the skid buffer must be extracted first and placed into the LSQ. If the LSQ fills up during this process, the pipeline must be flushed. Like retry, the key to this approach is to prioritize the non-speculative instructions and ensure that the speculative instructions do not impede progress. Skid buffers can reduce the ALU and network contention associated with NACK and instruction replay, but may result in more flushes if the skid buffer is small.

4.3 Virtual Channel-Based Flow Control

A third approach is to use the buffers in the network that transmits memory instructions from the execution to the memory units as temporary storage for memory instructions when the LSQ is full. In this scheme, the operand network is augmented to have two virtual channels (VCs): one for non-speculative traffic and one for speculative traffic. When a speculative instruction is issued at an ALU, its operands and memory requests are transmitted on the lower priority channel. When a speculative memory instruction reaches a full LSQ and cannot enter, it remains in the network and asserts backpressure along the speculative virtual channel. Non-speculative instructions use the higher priority virtual channel for both operands and memory requests. A non-speculative memory instruction that finds the LSQ full triggers a flush to avoid deadlock. Figure 3c shows a diagram of this approach.

This virtual channel approach has a number of benefits. First, no new structures are required so logic overhead is only minimally increased. Additional router buffers are required to implement the second virtual channel, but our experiments show that two-deep flit buffers for each virtual channel is sufficient. Second, no additional ALU

or network contention is induced by NACKs or instruction replays. Third, the higher priority virtual channel allows non-speculative network traffic to bypass speculative traffic. Thus non-speculative memory instructions are likely to arrive at the LSQ before speculative memory instructions, which reduces the likelihood of flushing.

Despite its conceptual elegance, this solution requires a number changes to the baseline network and execution engine. The baseline TRIPS implementation has the following pertinent features: it provides a single operand network channel that uses on-off flow control to exert back-pressure, each router contains a four-entry FIFO to implement wormhole routing, the microarchitecture can flush any in-flight instructions located in any tile or network router when the block they belong to is flushed, and finally, all of the core tiles (execution, register file, data cache) of the TRIPS processor connect to the operand network and will stall issue if they have a message to inject and the outgoing network FIFO is full.

Adjusting this network to support VC requires several augmentations: (1) an additional virtual channel in the operand network to separate speculative from non-speculative network traffic, including the standard buffer capacity and control logic needed by virtual channels, (2) virtualization of the pipeline registers, which must stretch into the execution and register tiles to allow non-speculative instructions to be proceed even if speculative instructions are stalling up the virtual network, (3) issue logic in these tiles that selects non-speculative instructions over speculative logic when the virtual network is backed up, and (4) a means to promote speculative instructions from the speculative virtual channel to the non-speculative channel when its block becomes non-speculative.

The trickiest part of the design is the promotion of speculative network packets to the non-speculative virtual channel when the previous non-speculative block commits. The TRIPS microarchitecture already has a commit signal which is distributed in a pipelined fashion to all of the execution units, memory units, and routers. When the commit signal indicates that the non-speculative block has committed, each router must nullify any remaining packets in the non-speculative virtual channel and copy any packets belonging to the new non-speculative block from the speculative VC to the non-speculative VC.

4.4 Reserving LSQ Entries

The three techniques of retry, skid buffers, and virtual channels all eliminate flushes on speculative memory instructions by buffering them when the LSQ is full. To further reduce flushes requires a means of increasing the likelihood that space is available when non-speculative instructions arrive at the LSQ. Aside from increasing

the total size of the LSQ partition, there are two obvious ways to adjust the effective size of the LSQ. One is to decrease the number high-priority memory instructions relative to the number of low-priority ones. While changing the ratio in TRIPS would require changing the block size, a more conventional architecture which dynamically assigned memory instructions to priority groups could accomplish this just by altering the size of the groups that are formed.

A second way is to reserve some number of LSQ slots for the high priority memory instructions. If enough slots are reserved to capture the maximum number of high priority instructions, then flushes can be completely eliminated. For TRIPS, this would mean a minimum LSQ size of 32 entries regardless of the number of partitions. Reserving a large number of LSQ entries for the worst-case behavior can reduce overall performance since the effective LSQ size for low-priority memory instructions is smaller. This effect can be mitigated by reserving only a few entries (for example 4 entries out of 16 or 32 total LSQ entries) for the high-priority instructions, and sharing the remainder among high and low priority instructions. The effectiveness can be improved by allocating the reserved entries only when all other entries in the LSQ are full. Section 5 further examines the effect of reserving LSQ entries.

5 Results

In this section, we present experiments, quantitative data and qualitative arguments to answer the following two questions about the proposed LSQs: (1) Can the proposed LSQ designs perform as well as maximally sized LSQs? and (2) How does the the LSQ design impact area, power and complexity requirements?

Simulation Infrastructure: The parameters of the simulated microarchitecture, the compiler details and the simulation methodology are summarized in Table 1. The microarchitecture timing simulator used in this study is a cycle accurate event-driven simulator and is validated to be within 11% of performance compared to the actual hardware RTL of the TRIPS prototype. On a suite of hand optimized benchmarks the simulated microarchitecture outperforms Alpha 21264 microarchitecture by 2x-3x. For this study, we analyze performance using the EEMBC benchmark suite and 12 SPEC CPU 2000 (8 FP + 4 INT) benchmarks with MINNESPEC medium sized reduced inputs. In addition to these benchmarks, we also use specially constructed synthetic benchmarks that stress the interconnection network and allow us to study the performance of the LSQ flow control schemes in isolation.

Parameter	Configuration
Overview	Microarchitecture supports block-atomic execution and predicated execution. Out-of-order execution with up to 1024 instructions inflight, up to 256 register reads, up to 256 register writes and up to 256 memory instructions can be simultaneously in flight. Up to 4 stores and 4 registers can be committed every cycle.
Compiler	Compiler generates single-entry-multiple-exit blocks with up to 128 instructions (32 ld/st's per block). The instructions in each block are statically assigned to the execution units (but are dynamically issued.)
Instruction Supply	IL1 partitioned into 5 banks (8KB/bank, 1-cycle hit). Local/Gshare Tournament predictor (10K bits, 3 cycle latency) with speculative updates; Local: 512(L1) + 1024(L2), Global: 4096, Choice: 4096, RAS: 128, BTB: 2048. 16-entry FA Instruction TLB replicated at each bank.
Execution Resources	16 Int/FP units, arithmetic operations are single cycle, mult uses pipelined functional units. Each execution unit is fed by a single issue 64-entry direct-mapped issue window which selects oldest ready instruction for issue. The instructions are *statically* mapped to issue windows.
Data Supply	4-bank cache-line interleaved DL1 (8KB/bank, 2-way assoc, writeback, write-around 2-cycle hit) with one read and one write port per bank to different addresses. Up to 16 outstanding misses per bank to up to four cache lines, 2MB L2, 8 way assoc, LRU, writeback, write-allocate, average (unloaded) L2 hit latency is 15 cycles, Average (unloaded) main memory latency is 127 cycles. 16-entry FA replicated DTLB per bank. Best case load-to-use latency is 5 cycles. Store forwarding latency is variable, minimum penalty is 1 cycle.
Interconnection Network	The 16 Int/FP units, the 4 data cache banks, the 4 register banks and the global controller are connected through a packet switched mesh network with wormhole routing. The banks are arranged in 5x5 grid, with data banks and register banks on adjacent edges. Each router uses round-robin arbitration. There are four buffers in each direction per router and 25 routers. The hop latency is 1-cycle.
Simulation	Execution-driven simulator validated to be within 11% of RTL design. 28 EEMBC benchmarks, 12 SPEC benchmarks (everything but C++&F90 benchmarks, gcc, perlbmk, vpr, crafty, sixtrack, apsi and mcf) and two synthetic worst case benchmark. SPEC run to completion on MINNESPEC medium reduced inputs except gzip, equake, bzip2, art, mesa, wupwise and mgrid for which we skipped 1B and simulated 100M with medium reduced inputs.

Table 1: Features of the distributed microarchitecture, compiler and simulation methodology for this study.

5.1 Performance of the Unordered LSQ

In the baseline unordered LSQ, store forwarding with N matches takes at least N cycles to complete. But, in an ordered LSQ, by virtue of order optimized search, the store forwarding penalty is always less than D cycles (where D is the datum size of the incoming load instruction, D is typically 8) irrespective of the number of matching stores. Despite, this difference we did not observe performance degradations due to the increased penalty in the unordered LSQ. This result can be explained by the fact that for more than 95% of loads there are one or zero matching stores (see Table 2). In addition, by using the optimized search method, *i.e.* searching backwards starting from the youngest matching unresolved block and going backwards, the number of loads that need to search past one matching store is reduced to less than 1%.

5.2 Performance of Flow Control Mechanisms

The performance of the flow control techniques is affected by the both number of overflows, and the performance penalty for each action resulting from an overflow (a NACK, stalling a virtual channel, or flushing the pipeline).

Benchmarks	*Number of matching stores*									
	Baseline Search					*Optimized Search*				
	0	1	2	3	4+	0	1	2	3	4+
SPEC	96.3	3.0	0.1	0.0	0.6	96.3	3.5	0.3	0.0	0.0
EEMBC	90.5	4.8	1.1	1.2	2.4	90.5	8.6	0.8	0.1	0.0

Table 2: Number of matching stores reported as a fraction of total number of dynamic loads.

The purpose of experiments in this section is to understand the relative costs of these two factors, first, using simple microbenchmarks, and then for SPEC and EEMBC benchmarks.

The number of overflows is determined by the size of each LSQ partition, the number of memory operations reaching each partition, and the rate at which memory instructions reach the partitions. The performance penalty of the flow control mechanism varies, depending on the number of flushes, the routing delays to and from the NACK'ed reservation station, and/or the congestion in the network.

Microbenchmark Results: To estimate the expected- and worst-case bounds and to understand the flow control performance in isolation for all three flow control mechanisms proposed in the paper, we constructed two microbenchmarks that drive the memory system with typical and high loads. The first microbenchmark (labelled TYP) shows expected case behavior, in which memory operations are evenly distributed among all banks. This microbenchmark contains a single-block loop that executes 1000 times, with 32 load instructions in the block, eight of which target each of the four banks. All of the instructions target the same address each iteration.

The second benchmark (WC) simulates the worst possible execution scenario for the interconnection networks, where all loads and stores in the window are mapped to the same cache and LSQ bank. Like TYP, WC runs a single block for 1000 iterations, with all 32 load operations in the block targeting cache bank 0 every iteration. For the TYP benchmark, the maximum number of instructions that will reside in each bank is 64, since there are 256 maximum in flight and 4 banks and for the WC benchmark the maximum number is 256 at each bank.

We show the results of the WC benchmark and TYP benchmark in Figure 4 for FLUSH, NACK, SKID and VC schemes for LSQ sizes ranging from 32 to 160. For each benchmark, we normalize the performance (measured in cycle counts) to a configuration with maximally sized, 256-entry LSQ partitions that never overflow. For all of the schemes, we show performance with no slots reserved for the non-speculative instructions.

For the SKID scheme, we evaluate performance with two different configurations (a) each partition utilizes a 128 entry skid buffer, the largest size accessible in 1 cycle at 65nm at 1.25GHz and (b) each partition utilizes a skid buffer that is sized according to the number of memory requests expected at each LSQ partition (labelled

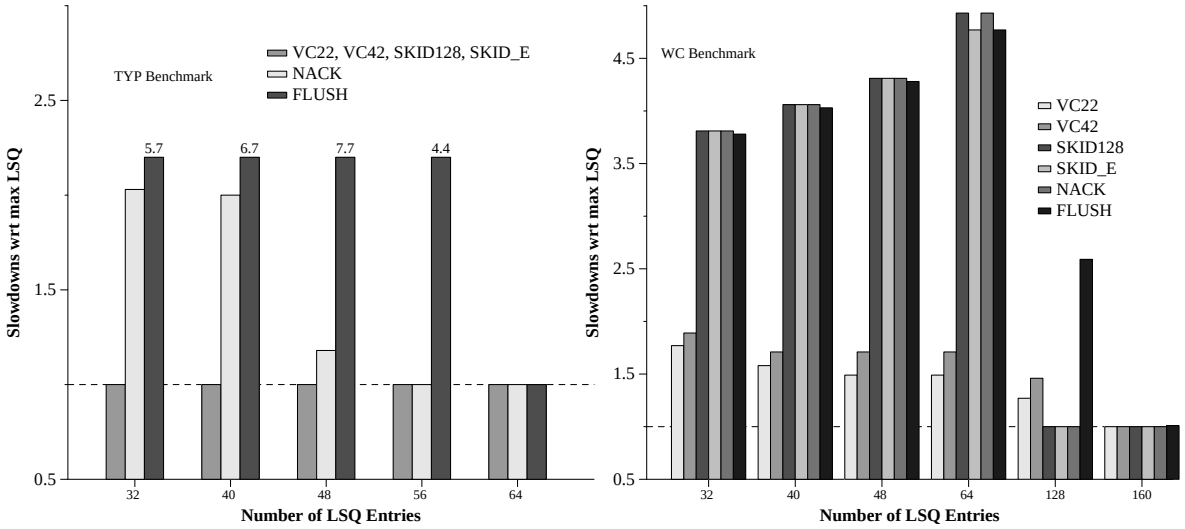


Figure 4: Performance of the LSQ flow control mechanisms under typical and worst case conditions. Slowdowns for TYP benchmark with flush scheme is annotated on the bar.

SKID_E), *i.e.* the skid buffer sizes are either (256 - LSQ partition size) for WC or (64 - LSQ partition size) for TYP.

For the VC mechanism, we show performance with 4 and 6 buffers at each router. For the 4-entry buffer, the buffer slots are equally divided between the two virtual channels and for the 6 entry buffer, 4 slots are reserved for the low priority channel and 2 slots are reserved for the high priority channel. The baseline configuration also 4 entry buffers but does not have any virtual channels.

As can be seen from the TYP benchmarks charts 4, the degradations due the flush scheme are much higher than the NACK scheme for any configuration. Between the remaining schemes there is no difference between the flow control scheme when the memory instructions are equally distributed between the cache banks. However, when overflows become more numerous due to load imbalances, as with the WC benchmark, the flow control schemes show more varied behavior.

First, we examine the performance between similar configurations like VC22 and VC42, SKID128 and SKID_E which have roughly the same cost for handling overflows. By comparing the performance of these benchmarks we can understand how the delivery of operands to the LSQ affects performance. We observed that the number of flushes for VC22 and VC42 for similar sized LSQs were different causing the disparity in performance; this is probably due to fact that deeper buffers promote more out-of-order execution consequently causing more capacity violations. Second, we examine performance across differently sized LSQs. The performance improves with smaller LSQ sizes (64 and less) because the LSQs fill up faster and the probability of flushes occurring

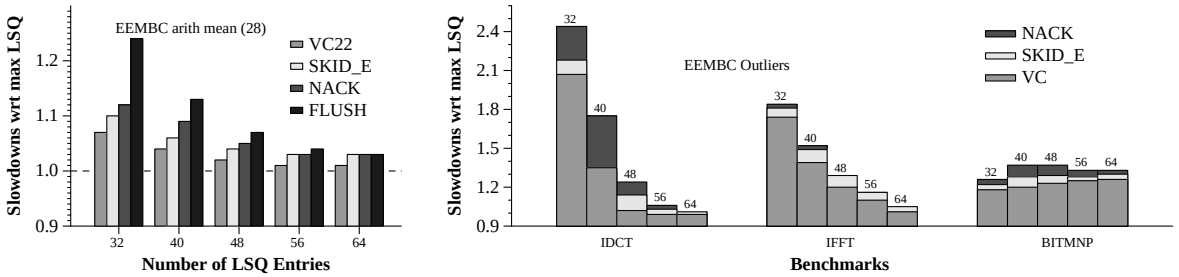


Figure 5: Left: Average LSQ Performance for the EEMBC benchmark suite. Right: Three worst benchmarks. bitmnp shows a different trend because there are fewer LSQ conflict violations in bitmnp when the LSQ capacity is decreased.

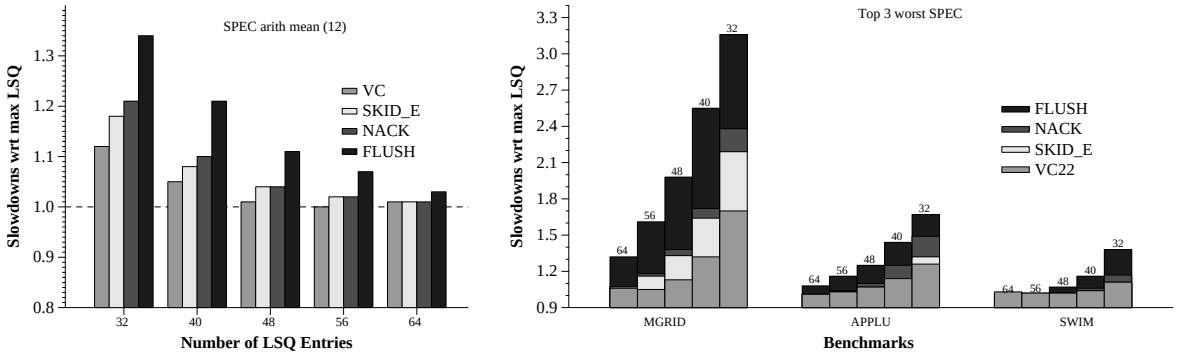


Figure 6: Left: Average LSQ Performance for the SPEC benchmark suite. Right: Three worst benchmarks.

earlier increases. Early flushes are always better than late flushes, given equal number of flushes. Finally, we note that The VC scheme is 2x faster compared to any other flow control scheme for all undersized LSQs even for the WC benchmark. The performance degradations are much lower for VC because the virtual channel promotion policies *naturally maintain instructions in program order* thereby reducing out-of-order LSQ subscription and the number of capacity violations. For rest of the performance analysis, we use skid buffers sized according to expected number of requests and the virtual channel scheme with equally provisioned virtual channels because they perform as well or better than the larger sized structures.

SPEC and EEMBC Results: Figures 5 and 6 show the performance of the different flow control schemes for the EEMBC and SPEC benchmarks respectively. For both these benchmarks, the VC flow control method works best and flush scheme results in unacceptable performance degradations. For LSQs that are as small as 40-entries (70% of the steady state number of memory requests) the VC mechanism shows a moderate performance degradation of 5% and 4% for SPEC and EEMBC benchmarks respectively. For 48-entry LSQs, with the VC scheme, the performance degradations drop to 1% and 2% respectively. Most of the performance degradation

```
realLow_1 = &realData_1[l_1];
imagLow_1 = &imagData_1[l_1];
realHi_1 = &realData_1[i_1];
imagHi_1 = &imagData_1[i_1];
 :
 :
realData_1[l_1] = *realHi_1 - tRealData_1;
imagData_1[l_1] = *imagHi_1 - tImagData_1;
realData_1[i_1] += tRealData_1;
imagData_1[i_1] += tImagData_1;
```

Figure 7: Code snippet from aiifft benchmark.

in SPEC (see Figure 6 Right) is due to FP benchmarks as these benchmarks typically have a larger fraction of inflight memory references than the SPEC INT programs.

Program Analysis of Outliers: We studied the worst three benchmarks (see Figure 5) from the EEMBC suite to understand the large performance degradations. While for most of the benchmarks, the memory accesses tend to be evenly distributed across the cache banks with occasional bursty behavior, the benchmarks with large degradations showed continuous unbalanced access patterns. In these benchmarks, the code generated by the compiler was such that most of the memory references for the benchmarks were targeted at one or few of the banks. For instance, consider the frequently executed code sequence in aiifft01(see Figure 7).

The inner loop contains 2 reads and 2 writes to two different arrays. The code generated by the compiler aligns the both the arrays to 256 byte boundaries and since the arrays are accessed by same indices, all of the four accesses end to the same bank. This problem is exacerbated by loop unrolling which our compiler performs quite aggressively. We changed the alignment of the arrays by hand and verified that performance improves. However, it is not clear if this alignment optimization can be automatically detected and fixed by the compiler without the programmer's help.

Another frequent code sequence that caused load imbalances is frequent use of static scalar variables. Static scalar variables in general cannot be register allocated (precise exceptions, consistency requirements etc.) and remain allocated to one bank for the life time of a program. If the program repeatedly uses the static variable then imbalances occur. For example, a file pointer declared as static and a program that writes to a large in memory file pointed by the file pointer.

These two case studies and the data in this section illustrate the important point that imbalances in partitioned memory systems cannot be easily detected and optimized for by the compiler and that low overhead hardware

Benchmark (aiifft)	*Number of LSQ Reserved Slots*									
	LSQ Size 48					*LSQ Size 40*				
	0	4	8	16	32	0	4	8	16	32
VC										
Performance	1.38	1.20	1.06	1.12	1.39	1.20	1.11	1.03	1.06	1.24
Inst/Flush	337	788	1.4M	No F	No F	643	1441	No F	No F	No F
NACK										
Performance	1.52	1.35	1.26	1.47	2.33	1.29	1.17	1.12	1.25	1.88
Inst/Flush	283	683	43K	No F	No F	515	1291	No F	No F	No F
Inst/NACK	21.58	2.21	1.15	0.70	0.23	55.25	5.29	2.56	1.16	0.36

Table 3: Performance improvements from reserving LSQ slots. No F stands for no flushes. A value of less than 1 for Inst/NACK indicates that instructions were NACK'ed multiple times.

mechanisms are essential for dealing with capacity violations.

Improving flow control performance:

As described in section 4.4, the performance of the flow control mechanisms can be improved by reserving a few slots for the non-speculative instructions to avoid some or all of the costly capacity violations. Table 3 illustrates the performance of the `aiifft` benchmark with the VC and NACK scheme with varying number of slots reserved for the non-speculative instructions. For the VC scheme, as the number of slots is increased performance gradually increases, and the number of committed instructions per flush increases. But increasing the number of reserved slots beyond 16 diminishes performance because reserving too many slots effectively stifles out-of-order execution. Similarly, for the NACK scheme, reserving more slots for the non speculative instructions increases performance until too many NACK'ed requests congest the network and the performance drops precipitously. While reserving slots is beneficial for the aiifft benchmark, in general we observed that for most of the benchmarks, not reserving any slots was as at least good as reserving slots.

5.3 Discussion of Power, Area and Complexity

Table 4 summarizes and compares the mechanisms described in this section based on metrics of performance, complexity, area, and power.

Power and Energy efficiency: Two mechanisms, address partitioning and Bloom filtering, are key to achieving high power efficiency in LSQs for large window processors. Firstly, partitioning the LSQ by addresses naturally reduces the number of entries incoming memory instruction has to search against. Over and above that, Bloom filtering reduces the number of memory instructions performing associative searches thereby improving power efficiency. As shown in Table 5, nearly 70-80% of the memory instructions (both loads and stores) can be

Metric	Performance	Complexity	Area	Power
Flush	Poor if flush frequent	Lowest	Best	Poor if flush frequent
NACK	Fair	Simple	Good	Poor if replay and flush frequent
SKID	Good	Moderate	High	Good by reducing replay
VC	Best	High	Moderate	Best (no replay)

Table 4: Summary of LSQ Optimization Mechanisms.

Benchmarks	*Average LSQ Activity Factor*					
	VC		*SKID*		*NACK*	
	40	48	40	48	40	48
SPEC	.21	.21	.27	.30	.30	.31
EEMBC	.26	.27	.38	.39	.38	.39

Table 5: Number of matching stores reported as a fraction of total number of dynamic loads

prevented from performing associative searches.

However, using Bloom filters incurs additional some additional power for reading and updating the filters for every memory instruction. In our implementation, each incoming memory instruction reads eight 32-bit registers (one corresponding to each inflight block) and simultaneously writes to one of the registers. Assuming a 1.25GHz clock, at 65nm power dissipated for a Bloom filter read/write is 110nW. Since eight of the filters have to be accessed in parallel, the total power is approximately, 1μW. In contrast, the dynamic power of the the CAM under similar technology constraints is 40mW (from a scaled synthesized CAM). Using the activity factors presented in Table 5, the average equivalent power for each memory access is about 8-10mW. At the same technology, this is roughly equivalent to the power consumed by an 8 entry, 64-wide CAM.

Area Analysis: Among the proposed mechanisms, assuming that the issue window is designed to hold onto instructions until explicit deallocation, the NACK mechanism is the most area efficient. Basically, it requires a cumulative storage of 1024 bits to identify the NACK'ed instructions (one bit for every instruction in the instruction window) and minimal changes to the issue logic to select and re-issue the NACK'ed instructions. The VC mechanism is next best in terms of area efficiency The area overheads of the VC are due to the additional storage required for pipeline priority registers in the execution units to avoid deadlocks and the combinational logic in routers to deal with promotion.The skid buffer scheme require the largest amount of storage, although most of the structure can be implemented as RAMs. A 24-entry skid buffer supplementing a 40-entry LSQ, approximately increases the LSQ partition by 3-4%.

It should also be noted here that the unordered, unified LSQ does not duplicate any state; in contrast, decomposed

LSQs typically maintain multiple redundant copies of the same data to support the constituent functions thorough physically different structures. For instance, a copy of the store address tag could be present in the forwarding structure and the same tag could also be present in the commit queues. Although much of this storage is in form of area-efficient RAMs, decomposition inherently introduces more area overheads than the unified LSQ design proposed in this paper.

Complexity: Among the schemes proposed in the paper, the VC scheme is probably the most difficult to implement. As explained in an earlier section, the VC scheme requires virtualization of not only the network routers but also the the execution units that feed the router. For instance, when the low priority channel in the network is backed up, the issue logic must supply the network with a high priority instruction even though it may be in the middle of processing a low priority instruction. The NACK scheme comes second or third depending on the baseline architecture – if the baseline allows instructions to be held in the issue queues until commit, implementing NACK is as simple as setting a bit in a return packet and routing it back to the source instead of the destination. However, if instructions are immediately deallocated upon execution from the windows, NACK may be considerable more complex. The skid buffer solution is probably the simplest of all the solutions: it requires some form of priority logic for selecting the oldest instructions, mechanisms for handling invalidations in the skid buffer and arbitration for the LSQ between instructions in the skid buffer and new instructions coming into the LSQ partition. Despite the apparent complexity of the schemes described here, we believe that that the schemes are realistic and lend themselves to efficient and simple implementation in hardware.

6 Conclusions

Load/store queues have generated so much recent work in the architecture community because they are one of the hardest structures to scale, due to their associative nature and the static uncertainty about relationships among memory operations. The recent work on partitioning the functionality of LSQs into distinct structures has shown great promise in making them energy-efficient for large windows. In this paper, we have presented another possible LSQ design that creates additional opportunities for scaling by constructing address-interleaved LSQs and using them with low overhead overflow-handling mechanisms.

We proposed using the buffering provided by on-chip micronetworks to gracefully handle overflows in address-interleaved LSQs, employing priority-based virtual channels to avoid deadlock scenarios efficiently. The virtual channel flow-control approach works extremely well, with no degradation in performance for the SPEC and EEMBC benchmarks using four 48-entry LSQ partitions to support a 1024 instruction window processor with a

maximum number of 256 loads and stores in flight. We also show that the energy-efficiency of the small, address-interleaved LSQ banks can improved further by the addition of simple Bloom filters. With the combination of flow-control supported address-interleaving and Bloom filters, the per-access energy of the LSQ is reduced to approximately the energy consumed by an 8-entry fully associative traditional LSQ.

Address-interleaved LSQs have been suggested as effective mechanism for scaling LSQs for nearly a decade now [6] but until this work, researchers had no graceful way of handling overflows, so had to oversize partitions to keep flushes sufficiently low. We believe that the solutions presented in this paper provide a long-term solution to this problem, even to more partitions and larger instruction windows. Whether the complexity of the flow control mechanisms outweighs the complexity of the partitioned-functionality approach (especially in an interleaved memory system) is an open question. Although all of the techniques we have presented are applicable to conventional ISAs and microarchitectures, the added complexity of these techniques is much less for distributed microarchitectures like TRIPS simply because many of the necessary mechanisms are already extant in the design.

7 Acknowledgments

We thank the anonymous reviewers, Joel Emer and Amir Roth for their suggestions that helped improve the quality of this paper. This research is supported by the Defense Advanced Research Projects Agency under contracts F33615-01-C-4106 and NBCH30390004 and an NSF instrumentation grant EIA-0303609.

References

[1] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate vs. ipc : The end of the road for conventional microprocessors. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 248–259, June 2000.

[2] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *Proceedings of the 2003 Intl Symposium on Microarchitecture*, Dec. 2003.

[3] L. Baugh and C. Zilles. Decomposing the load-store queue by function for power reduction and scalability. In *P=ac^2 Conference, IBM Research*, Oct. 2004.

[4] D. Burger, S. Keckler, K. McKinley, M. Dahlin, L. John, C. Lin, C. Moore, J. Burrill, R. McDonald, and W. Yoder. Scaling to the end of silicon with EDGE architectures. *IEEE Computer*, 37(7):44–55, July 2004.

[5] H. W. Cain and M. H. Lipasti. Memory ordering: A value-based approach. In *Proceedings of the 31st International Symposium on Computer Architecture*, June 2004.

[6] M. Franklin and G. S.Sohi. ARB:a hardware mechanism for dynamic reordering of memory references. *IEEE Transactions on Computers*, 45(5):552–571, 1996.

[7] A. Roth. Store vulnerability window (svw): Re-execution filtering for enhanced load optimization. In *Proceedings of the 32th International Symposium on Computer Architecture*, June 2005.

[8] 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 2003 Intl Symposium on Microarchitecture*, Dec. 2003.

[9] T. Sha, M. M. Martin, and A. Roth. Scalable store-load forwarding via store queue index prediction. In *Proceedings of the 2005 Intl Symposium on Microarchitecture*, Nov. 2005.

[10] G. S. Sohi and M. Franklin. High-bandwidth data memory systems for superscalar processors. In *Proceedings of the 4th International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 53–62, Sept. 1991.

[11] S. S. Stone, K. M. Woley, and M. I. Frank. Address-indexed memory disambiguation and store-to-load forwarding. In *Proceedings of the 2005 Intl Symposium on Microarchitecture*, Nov. 2005.

[12] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. Wavescalar. In *36th Annual International Symposium on Microarchitecture*, pages 291–302, December 2003.

[13] M. B. Taylor and W. Lee. Scalar operand networks. *IEEE Trans. Parallel Distrib. Syst.*, 16(2):145–162, 2005. Member-Saman P. Amarasinghe and Member-Anant Agarwal.

[14] E. F. Torres, P. Ibanez, V. Vinals, and J. M. Llaberia. Store buffer design in first-level multibanked data caches. In *Proceedings of the 32th International Symposium on Computer Architecture*, June 2005.

[15] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software: RAW machines. *IEEE Computer*, 30(9):86–93, September 1997.