Late-Binding: Enabling Unordered Load-Store Queues

Simha Sethumadhavan [§] Franziska Roesner [§]
Joel S. Emer[†] Doug Burger [§] Stephen W. Keckler [§]

[§]Department of Computer Sciences
The University of Texas at Austin
cart@cs.utexas.edu

[†]VSSAD, Intel
Hudson, MA
joel.emer@intel.com

ABSTRACT

Conventional load/store queues (LSQs) are an impediment to both power-efficient execution in superscalar processors and scaling to large-window designs. In this paper, we propose techniques to improve the area and power efficiency of LSQs by allocating entries when instructions issue ("late binding"), rather than when they are dispatched. This approach enables lower occupancy and thus smaller LSQs. Efficient implementations of late-binding LSQs, however, require the entries in the LSQ to be unordered with respect to age. In this paper, we show how to provide full LSQ functionality in an unordered design with only small additional complexity and negligible performance losses. We show that late-binding, unordered LSQs work well for small-window superscalar processors, but can also be scaled effectively to large, kilo-window processors by breaking the LSQs into address-interleaved banks. To handle the increased overflows, we apply classic network flow control techniques to the processor micronetworks, enabling low-overhead recovery mechanisms from bank overflows. We evaluate three such mechanisms: instruction replay, skid buffers, and virtual-channel buffering in the on-chip memory network. We show that for an 80-instruction window, the LSQ can be reduced to 32 entries. For a 1024-instruction window, the unordered, late-binding LSQ works well with four banks of 48 entries each. By applying a Bloom filter as well, this design achieves full hardware memory disambiguation for a 1,024 instruction window while requiring low average power per load and store access of 8 and 12 CAM entries, respectively.

Categories and Subject Descriptors

C [**Computer Systems Organization**]; C.1 [**Processor Architectures**]; C.1.1 [**Single Data Stream Architectures**]: Subjects: Single-instruction-stream, single-data-stream processors (SISD)Additional Classification:; C [**Computer Systems Organization**]; C.2 [**Computer-Communication Networks**]; C.2.1 [**Network Architecture and Design**]: Subjects: Packet-switching networks

General Terms

Design, Performance

Keywords

Late-binding, Memory Disambiguation, Network flow control

PREPRINT: *To Appear in the International Symposium on Computer Architecture 2007.*

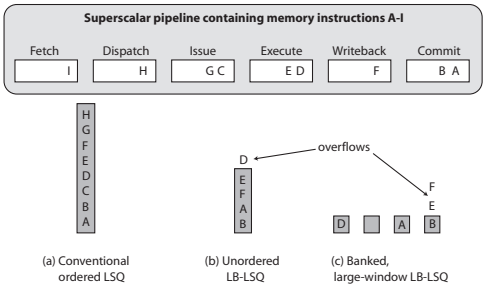

Figure 1: High-Level Depiction of Ordered vs. Unordered LSQs.

1. INTRODUCTION

A long-standing challenge in computer architecture has been handling memory dependences in a scalable manner without sacrificing programmability or parallelism. Dataflow computers in the 1970's and 80's supported high concurrency, but required unconventional programming models (write-once memory semantics) to prevent incorrect memory behavior. Modern superscalar processors enforce memory dependences while supporting standard programming models using load/store queues (LSQs). These LSQs, however, are one of the primary structures inhibiting the scaling of superscalar processors to larger in-flight windows, since they have typically supported only a few tens of in-flight loads and stores due to power and complexity limits.

These limits have led an industry-wide shift to chip multiprocessors (CMPs). Much current research is focusing on CMPs in which the individual are processors smaller, lower power, and simpler, effectively reducing single-thread performance to support more threads. This trend is reducing the instruction-level parallelism that can be exploited, placing a higher burden on software and/or programmers to use a fine-grained chip multiprocessor effectively. An alternative to CMPs are tiled large-window microarchitectures that can scale to thousands of in-flight instructions, while potentially exploiting multiple threads in addition to providing a large window for single-threaded execution. Memory disambiguation at this scale, however, has been a key limitation for these types of architectures. This paper describes an LSQ design that both provides power and area improvements for conventional superscalar designs and which provides the ability to scale to thousands of instructions power efficiently. The two new techniques that provide this scaling are *unordered late binding*, in which loads and stores allocate LSQ entries after they issue, and *lightweight overflow control*, which enables good performance with unordered LSQs that are divided into multiple small, address-interleaved partitions.

Late binding: Traditional LSQs allocate entries at fetch or dispatch time, and deallocate them at commit. Thus, entries in a conventional LSQ physically *age-ordered*, a feature that LSQ designs exploit to provide the necessary functionality efficiently. When an LSQ reaches capacity, the microarchitecture typically throttles fetch until a load or store commits and is removed from the LSQ. Figure 1 shows a simple six-stage pipeline diagram with nine memory instructions (loads and stores labeled A through I) in different stages. As shown in Figure 1a, a conventional eight-entry LSQ is full after H is dispatched, stalling the fetch of later instructions.

Unordered, late-binding LSQs (ULB-LSQs) can reduce both the average occupancy time and average number of entries in use. ULB-LSQs achieve this reduction by allocating entries only when a load or store issues, instead of when it is fetched, permitting a smaller, more efficient structure. Figure 1b shows the smaller ULB-LSQ that must be sufficiently large only to capture the in-flight loads and stores after they have issued. Figure 2 shows the potential savings of late-binding, issue-time allocation. On the Alpha 21264 microarchitecture, only 32 or fewer memory instructions must be buffered in the LSQ for 99% of the execution cycles, even though the original 21264 design had a combined LQ/SQ capacity of 64 entries.

To achieve this reduction, however, the entries in an ULB-LSQ must be *unordered*; loads and stores are allocated LSQ entries in their dynamic issue order as opposed to the traditional in-order fetch sequence. Maintaining age order with issue-time allocation would require a complex and power-inefficient compacting-FIFO-like circuit [7]. This paper describes an ULB-LSQ design that requires only small additional complexity while performing comparably to a traditional ordered LSQ.

Low overhead overflow handling: A second advantage above and beyond the reduced size is that ULB-LSQs lend themselves naturally to address partitioning, enabling smaller banks that are indexed by address. However, smaller banks will experience more overflows. In small-window superscalar processors, flushing the pipeline on an ULB-LSQ overflow is an acceptable policy, since the ULB-LSQ can be sized to save area and power over a conventional design while overflowing infrequently. However, ULB-LSQs can also provide efficient memory disambiguation for large-window processors, in which thousands of instructions may be in flight [12, 14, 24], by exploiting the ability to address interleave LSQ banks. The late binding and unordered organization are both necessary to support an undersized, address-interleaved LSQ, since the mapping of loads and stores to LSQ banks cannot be known at fetch/dispatch time.

In large-window designs, however, the probability of overflows grows, since the global ULB-LSQ is divided into smaller interleaved partitions that are more susceptible to overflows when the distribution of loads and stores to banks is unbalanced. Figure 1c shows how load imbalance for an address-interleaved ULB-LSQ increases the likelihood of overflow. In the example, each of four address-interleaved partitions holds one entry. Whereas instruction *E* could have been held in the centralized ULB-LSQ of Figure 1b, *E* conflicts in the banked example with instruction *B*, which was mapped to the same partition.

As the banks are shrunk and the number of overflows increases, the conventional techniques of throttling fetch or flushing the pipeline become too expensive. By incorporating techniques to handle overflows with low overhead, an ULB-LSQ for a large-window design can consist of small partitions, resulting in low energy per load or store, while still supporting high performance across a large-window processor. We observe that in a distributed, large-window processor, LSQ banks can be treated as clients on a micronetwork, with overflows handled using traditional network flow-control techniques adapted for a distributed processor microarchitecture. We evaluate three such techniques in this paper: *instruction replay*, *skid buffers*, and *micronet virtual channels*.

This paper shows that ULB-LSQs with appropriate overflow support work well for both small and large-window processors. For an Alpha 21264 microarchitecture with an 80-instruction window, a 32-entry ULB-LSQ using flush-on-overflow provides the same performance as a 64-entry split LQ/SQ. For the 1,024-instruction window TRIPS processor, four banks of 48 entries each–using virtual channel flow control to handle overflows–provides the same performance as an idealized 1,024-entry LSQ. By adding a Bloom Filter [22], the banked ULB-LSQ design incurs, for a kilo-instruction window, an average of only eight and twelve CAM entries searched per load and store, respectively.

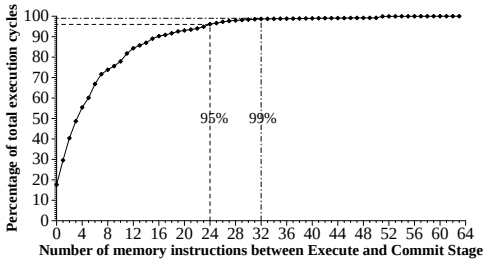

Figure 2: Potential for undersizing: In an Alpha 21264, for 99% of the execution cycles across 18 SPEC2000 benchmarks, only 32 or fewer memory instructions are in flight between execute and commit.

2. PRIOR LSQ INNOVATIONS

Most of the recent LSQ optimization proposals, including proposals for non-associative LQ/SQs, focus primarily on reducing the dynamic power and latency of LSQs. Typically these schemes also reduce or eliminate the power-hungry CAMs, but add numerous predictors and other significant supporting structures outside of the LSQ, effectively increasing the area of the total memory disambiguation logic.

Cain and Lipasti eliminate the associative load queue by having loads compare their speculative values to the cached values at commit time to detect mis-speculations [13]. Roth proposed enhancements to Cain and Lipasti's scheme, to reduce the number of loads that have to perform commit time checking [20]. These mechanisms eliminate the CAM from the LQ but require additional bits in the load queue RAM (64bits) to hold the load values. Roth's mechanism requires additional storage of 1KB (SSBF) for supporting commit-time filtering.

Sha, Martin and Roth 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 [27]. This solution, coupled with the non-associative load queue scheme proposed in Roth's earlier work, can completely eliminate all the associative structures from the LSQ. However, instead of the associative CAMs, this solution requires large multi-ported dependence/delay predictors (approximately 23.5KB, as reported in [4]), effectively improving dynamic power at the expense of area.

Garg, Rashid and Huang propose another mechanism for eliminating associative LQ/SQs [6]. In the first phase of two-phase processing, loads and stores speculatively obtain their value from a L0 cache. In the second phase, the memory instructions are

Scheme	ROB Size	Unoptimized				Optimized									Unoptimized :optimized Ratio
		Depth		Storage (KB)		Depth		CAM Width		RAM Width		Storage (KB)		Supporting Structures	
		LQ	SQ	LQ	SQ	LQ	SQ	LQ	SQ	LQ	SQ	LQ	SQ	(KB)	
SVW - NLQ	512	128	64	1.00	1	128	64	0	12	92	64	1.44	0.78	1	1.61
SQIP	512	128	64	1.00	1	128	64	0	0	92	64	1.44	0.5	23.5	12.72
Garg et al.	512	64	48	0.50	0.75	64	48	0	0	92	0	0.72	0	16	13.38
NoSQ	128	40	24	0.31	0.375	40	0	0	0	92	0	0.45	0	11	16.65
FnF	512	128	64	1.00	1	128	0	0	0	92	0	1.44	0	23.75	12.59
Stone et al.	1024	N/A				N/A								18	N/A
LateBinding-Alpha	80	32	32	0.25	0.5	32		12		92		0.5		0.0625	0.75
LateBinding-TRIPS	1024	1024		16		192		12		92		3		0.25	0.20

Table 1: Area for LSQ and supporting structures for recent related work

re-executed in program order, without any speculation, and access the regular L1 cache. Any difference in the load values between the two phases results in corrective action. This mechanism, while eliminating the CAM, requires a 16KB L0 cache and an age-ordered queue for holding the values read during the first phase.

Subramaniam and Loh [4] and Sha, Martin, and Roth [28] both propose methods for completely eliminating the store queue, by bypassing store values through the register file and LQ, respectively. Sha, Martin and Roth avoid speculative store buffering by executing stores in program order, while Subramaniam and Loh do so by using the ROB or the physical register file. Both proposals use sophisticated dependence predictors that are smaller than their earlier schemes but still require additional area for dependence predictors and combinational logic for renaming.

Akkary, Rajwar, and Srinivasan propose two-level store buffers, each of which are centralized, fully associative and age ordered [5]. Stores first enter 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 always commit from the second level buffer. This scheme reduces power and delay, but still requires a worst-case sized L2 store buffer and is thus area inefficient.

Stone, Woley, and Frank suggest partitioning the LSQ into its three functions, and distributing and address interleaving all of them [3]. The three structures are a set associative cache for forwarding (80-bit wide, 512 sets, 2-way), a non associative FIFO for commit, and an address-indexed timestamp table (8K entry, 8-bit wide) for checking speculation. The partitioning removes the CAMs but has high area overheads.

Torres et al. propose using a distributed, address-partitioned, forwarding buffer backed up by a centralized, fully associative age-indexed LSQ [1]. Baugh and Zilles use a small centralized forwarding buffer, but address-partitioned unordered structures for violation detection [10]. Both these schemes increase the area required for memory disambiguation to optimize for latency and power.

Table 1 summarizes the ROB size, the area of the LSQ before and after the optimizations, the size of the supporting structures required by these optimizations (but which may be already be present in the design for performance reasons), and finally the ratio of total area required for memory disambiguation, before and after the optimizations. We computed the area of the memory structures, in bytes of storage, assuming the CAM cell area to be three times larger than the RAM cell. We also assumed 40-bit addresses and 64-bit data, and that all the unoptimized designs had 12-bit partial addresses in CAMs and rest of the address bits in the RAMs. The depth of the queues, however, is different for each of these structures. The table shows that the proposed schemes add area overhead between factors of 1.5 to 16.5. When discounting the dependence predictor, the area overheads are between factors of 1.5 and 13.

In contrast to all of the above schemes, the design proposed in this paper uses late binding to reduce the area and latency without any additional state outside of the LSQ. Dynamic power reduction, however, requires additional state in the form of address-based Bloom filters [22]. These structures take up only few hundreds of bytes and can even be further reduced by optimizations suggested by Castro et al. [2].

Other researchers have also applied issue-time binding, explicitly or implicitly, to improve the efficiency of microarchitectural structures. Monreal et al. use late allocation to reduce the number of physical registers [19]. The effectiveness of some LSQ optimizations like address-based filters [22] or the small associative forwarding buffers [10, 8, 1] can also be explained in part by late allocation.

3. BACKGROUND

Most LSQs designed to date have been age indexed, because age indexing supports the physical sorting of instructions in the LSQ based on their age. LSQs must support three functions: commit of stores, detection of memory ordering violations, and forwarding of earlier store values to later loads. The physical ordering makes some of these operations simpler to support, but is not fundamentally required for committing of stores and violation detection, and only provides speed benefits for store forwarding in rare cases.

In an age-indexed LSQ, the address and value of an in-flight memory instruction is stored in an LSQ slot obtained by decoding the age of the memory instruction. This organization results in a LSQ that is *physically* ordered; the relative ages of two instructions can be determined by examining the physical locations they occupy in the LSQ. For example, it is simple to determine that instruction at slot 5 is older than instruction at slot 8 because slot 5 is physically "before" slot 8. Additionally, this mechanism allows determining the relative order between all instructions in the LSQ that satisfy some criterion (i.e. a matching address). For example, if slots 25, 28 and 29 are occupied in the LSQ, linearly scanning the the LSQ from position 29 will reveal the most recent older instruction first (28) and then then next oldest (25) and so on. In some cases, circuit implementations exploit the physical ordering to accelerate LSQ operations. To understand the design changes that an unordered LSQ requires, it is instructive to examine how LSQ ordering supports the three functions that the LSQ combines.

Commit: The LSQ buffers all stores to avoid potential write-after-write hazards between stores to the same address that execute out-of-order. Additionally, the stores cannot be written out until they are non-speculative. Once a store is determined to be non-speculative, the store address and value are written to the cache using the age supplied from the ROB/control logic. With ordering, age-based indexed lookup is sufficient. Without ordering, a search is necessary to find the oldest store to commit.

Operation	Search	Input	Output	Num matches	Sorting Required
Forwarding	$\geq$	LD age	Older STs	≥ 1	Yes
Violation	$\leq$	ST age	Younger LDs	≥ 1	No
Commit	==	ROB age	ST to commit	1	No

Table 2: Summary of LSQ operations and ordering requirements

Violation Detection: The LSQ must report a violation when it detects that a load following a store in program order, and to same address, executed before the store. To support this operation, the LSQ buffers all in-flight memory data addresses, and when a store arrives at the LSQ, checks for any issued loads younger than and to the same address as the store. If there are any matching loads, a violation is reported. For this operation, the LSQ need only determine the set of younger load instructions. It does not require sorting among the multiple matching loads based on their age. In ordered LSQ circuit implementation, the age of the incoming instruction is decoded into a bit mask and all bits "before" the decoded bit vector are set. In the case of store forwarding with multiple matches, the most recent store and successive older stores can be accomplished by linearly scanning the bit mask.

Store Forwarding: The LSQ must support forwarding from the uncommitted buffered stores in the LSQ to in-flight loads that issue after the stores. When a load arrives at the LSQ, it checks for older stores to the same address. If there are matching stores, the LSQ ensures that the load obtains its value from the most recent matching store preceding the load. To support this functionality when a load matches multiple stores, the LSQ sorts the matching stores based on their ages and processes the matching stores until all the load bytes have been obtained or until there are no matching stores.

The age-indexing policy requires an LSQ that is sized large enough to hold all in flight memory instructions (2^{age} slots), which results in a physically ordered LSQ. The ability to sort through multiple matching instructions is especially useful for forwarding values from multiple matching stores to a load, but a coarser age comparison is sufficient for implementing the other LSQ operations (Table 2). Additionally, the LSQ allocation policy is conservative. Even though the LSQ slots are occupied only after the instructions execute they are allocated early, during instruction dispatch. Traditional age-indexed LSQs are thus both overdesigned in terms of functionality and overprovisioned in terms of size.

4. AN UNORDERED, LATE-BINDING LSQ DESIGN

Late-Binding LSQs address the inefficiencies resulting from the "worst-case" design policies used in traditional LSQs. By allocating the memory instruction in the LSQ at issue, the sizes of ULB-LSQs are comparartively reduced. Allocating memory instructions at issue requires a set of mechanisms different from allocation in age-indexed LSQs. When a load or store instruction arrives at the ULB-LSQ the hardware simply allocates an entry from a pool of free LSQ slots instead of indexing by age. This allocation policy results in an LSQ that is physically *unordered* in which the age of the instruction has no relation to the slot occupied by the instruction.

To compensate for the lack of ordering, the ULB-LSQs take a more direct approach to determining the age by explicitly storing the age information in a separate age CAM. The age CAM is a special type of CAM that can output greater/lesser/equal results instead of just the equality matches. The LSQ functions that used the implicit age information in the age-indexed LSQ for implementing the LSQ operations now use the explicit associative age CAM to determine younger and older instructions. Figures 3 and 4 illustrates and contrasts the structures used in the ULB-LSQ and traditional LSQ implementations, where M is the memory instruction dow size, and U is the ULB-LSQ size.

To support commits, the small age CAM is associatively searched with the age supplied by the ROB. The address and data from the exact matching entry are read out from the CAM and RAM respectively, and sent to the caches. This extra associative search is avoidable if the baseline architecture holds the ULB-LSQ slot id allocated to the store in the ROB.

To support violation detection, when a store arrives it searches the address CAM to identify matching loads, and searches the age CAM using the greater-than operator to identify younger loads. The LSQ then performs a logical OR of the results of the two searches. If any of the resulting bits is one then a violation is flagged. Detecting violations is simpler in this LSQ compared to age-indexed LSQs, since no generation of age masks is necessary.

Supporting forwarding is more involved because the ULB-LSQ does not have the total order readily available. In the case of only one match, the loss of order does not pose a problem; however when there are multiple matches, the matches must logically be processed from most recent to the oldest. In the ULB-LSQ, on multiple store matches, the age of each match is read out from the ULB-LSQ, one per cycle, and decoded into a per-byte bit vector. Bytes to forward to the load replace bytes from other stores if the later-found store is more recent in program order. This step reconstructs the physical ordering between the matches from the ULB-LSQ, but may take multiple cycles to do so. Once the ordering is available, store forwarding proceeds exactly as in an age-indexed LSQ [23]. Thus, compared to the age-indexed LSQ, which may already require multiple cycles to forward from multiple stores, the ULB-LSQ requires additional cycles for creating the decoded bit-vector. However, as we will show in the next section, these additional cycles rarely affect performance because multiple store forwarding is uncommon in many benchmarks.

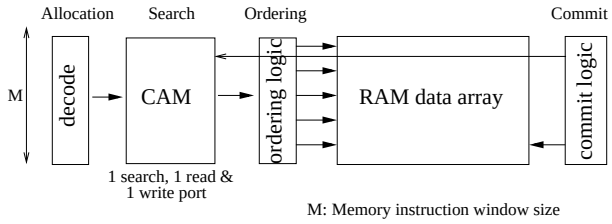

Figure 3: The Age-Indexed LSQ

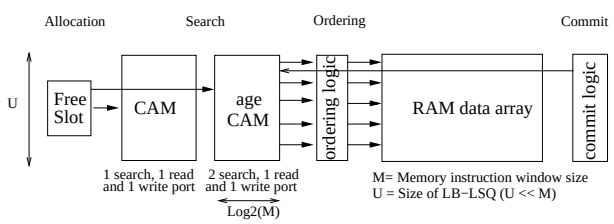

Figure 4: The ULB-LSQ Microarchitecture

4.1 Performance Results

The performance of the ULB-LSQ depends on the number of entries in the LSQ, which affects the number of LSQ overflows. Performance will also be affected by the relative cost of each overflow and the additional penalty for multi-cycle store forwarding (in the case of multiple address matches). We modeled the ULB-LSQ in the sim-alpha [15] simulator and simulated single Simpoint regions of 100M for the 18 SPEC benchmarks compatible with our experimental infrastructure.

In this set of experiments, ULB-LSQ overflows are handled by partially flushing the pipeline and re-starting execution from the oldest unarrived memory instruction at the time of the overflow. The penalty of an overflow is 15 cycles, which matches the cost of a branch misprediction. Table 3 summarizes the unordered LSQ behavior and statistics. The first two columns show the number of memory instructions between the commit and execute stages for 95% and 99% of the execution cycles. The next eight columns show the number of overflows per 1000 memory instructions, and the performance normalized against the Alpha, for ULB-LSQ sizes ranging from 16 to 40 entries. The final column shows the percentage of dynamic loads that forward from more than one store. From the table, for 14 of the 18 benchmarks, for 99% of the cycle time, there are 32 or fewer uncommitted but executed memory instructions. This explains why a 32 entry ULB-LSQ does not show any performance degradation.

To isolate the performance impact of slower multiple forwarding, we increased the latency of store forwarding in the baseline simulator without changing the LQ/SQ size. In this experiment, if a load matches with N (N $>$ 1) older stores, then store forwarding takes an additional N cycles even if load does not have to get their data from all N stores. The results showed no performance degradation for any of the benchmarks because loads rarely need to get data from multiple stores. From Table 3, the number of loads that have to forward from two or more stores is less than 0.2% on the average.

Another interesting trend in the Table 3 is that the performance impact of the number of overflows is different for High vs. Low IPC benchmarks. For instance, for the 16 entry LSQ, even though 179.art incurs more flushes per 1K instructions than 178.galgel (651 vs. 463), the performance degradation is higher in the case of 178.galgel (24% vs. 96%). This data indicates that unless low-overhead overflow mechanisms are invented for machines that support high ILP, the performance impact of undersizing the LSQ can be quite high.

4.2 Power

This section examines the dynamic power consumption of an ULB-LSQ against the Alpha LQ/SQ organization. The ULB-LSQ design holds 32 memory operations, while the Alpha has a separate 32-entry LQ and SQ. Although the size of the ULB-LSQ is smaller, each memory operation has to access the same number of address CAM entries in both designs because the Alpha has a partitioned LQ/SQ. Even so, the ULB-LSQ will have higher dynamic power per access because of the additional accesses to the age CAM.

To measure the increase in power per access, we assumed that the LSQ power is approximately equal to the power consumed by the CAM and the RAM. Thus, the power-per-access of the Alpha LQ or SQ will be purely from the address CAM (P_{addr}) and the RAM (P_{ram}), while the power consumed by the ULB-LSQ will be the power from the address CAM, the age CAM (P_{age}) and the RAM. Since the ULB-LSQ and Alpha have the same number of entries, their sizes will be the same. Thus the power increase will be $(P_{age} + P_{addr} + P_{ram})/(P_{addr} + P_{ram})$.

We synthesized the design using IBMs 130nm ASIC methodology with the frequency set at 450MHz, and verified that the age CAM will fit in the same cycle time as the address CAM. Even though the delay of the age CAM was approximately 20% more than the address CAM, the delay was still not long enough to be on the critical path. Thus, assuming that the LB-CAMs can be run at the same clock frequency as the traditional LQ/SQ, the power increase is simply the ratio of the capacitances. From our synthesis, the capacitance of the address CAM (32x40b, 1 search, 1 read and 1 write port) was 144.5pF (P_{addr}) while the capacitance of the age CAM was 12.86pF (P_{age}). Thus the power overhead is roughly 8% even after neglecting the power due to the RAM.

However, both the ULB-LSQ and the Alpha LSQ can benefit from Bloom filters. With a 64 entry, 8-bit counting Bloom Filter [16], we observed that 88.7% of the age and address searches can be filtered. Applying this optimization, the additional power required by the unordered design can be reduced to under 1% of the power consumed by the original Alpha LSQ.

4.3 Implementation Complexity

The ULB-LSQ design differs from a traditional LSQ in the following ways: (1) the entries are managed as a free-list, (2) multiple store forwarding requires additional control logic to scan through matching entries, and (3) the LSQ must detect and react to overflows. These differences are not a significant source of complexity because many existing microarchitectural structures implement similar functionality. For example, MSHRs and the physical register files are managed as free-lists. The scanning logic has been implemented in traditional LSQs which do not flush on multiple matches. Overflow is flagged when there are no free LSQ entries, and is simple to implement. Furthermore, the ULB-LSQ does not require any modifications to the load/store pipeline. The LSQ operations are pipelined in the exact same way as the age-ordered LSQ implementation in the POWER4 [26].

These results showed that for small-window processors like the Alpha 21264, even with simplistic overflow handling mechanisms, the queue sizes can be reduced by half without affecting performance. The smaller queue size directly translates to improvements in static power, dynamic power, and latency. Most important, the unordered nature of the ULB-LSQ allows it to be partitioned without introducing additional area overheads as explained next.

5. LARGE-WINDOW PROCESSORS

While high-performance uniprocessor design has fallen out of favor with industry due to scaling difficulties, many researchers are examining novel and scalable means for extracting more ILP by using a larger instruction window. Such designs often employ some form of partitioning to implement larger microarchitectural logical structures without sacrificing clock cycle time. In particular, the architectural trends motivating the design of a partitioned LSQs include (1) very large instruction windows with hundreds of in-flight memory instructions, and (2) partitioning of microarchitectures for scaling to higher execution and local memory bandwidth.

5.1 Partitioning the LSQ

In a partitioned microarchitecture, an LSQ partition is best matched with a partition of an address-interleaved level-1 cache. Partitioning an age-ordered LSQ is not straightforward as the per-instruction age identifiers do not indicate dependence relationships between load and store instructions. Distributing age-interleaved partitions to address-interleaved caches will effectively result in significant cross-partition communication among LSQ and cache banks and between matching loads and stores to the same address.

Benchmark	Occupancy		Flushes per 1K mem instr				Normalized IPC				Baseline IPC	% of LDs matching 0 or 1 STs
	%program cycles		LSQ entries				LSQ entries					
	95%	99%	16	24	32	40	16	24	32	40		
164.gzip	7	22	6	4	3	1	1.00	1.00	1.00	1.00	1.60	99.95
175.vpr	14	21	10	2	0	0	0.93	1.00	1.00	1.00	0.87	99.72
177.mesa	8	14	8	0	0	0	0.05	1.00	1.00	1.00	1.16	99.8
178.galgel	24	24	463	88	0	0	0.04	0.56	1.00	1.00	2.70	100
179.art	35	43	651	131	33	11	0.76	0.91	1.00	1.00	0.63	99.95
183.equake	3	6	0	0	0	0	1.00	1.00	1.00	1.00	0.96	99.91
188.ammp	11	15	4	0	0	0	1.01	1.00	1.00	1.00	1.31	99.91
189.lucas	11	13	0	0	0	0	1.02	1.00	1.00	1.00	0.76	100
197.parser	10	17	4	0	0	0	1.00	1.00	1.00	1.00	1.17	99.84
252.eon	15	20	28	2	0	0	0.94	1.00	1.00	1.00	1.17	98.58
253.perlbmk	9	13	2	0	0	0	1.00	1.00	1.00	1.00	0.83	98.91
254.gap	6	12	0	0	0	0	1.00	1.00	1.00	1.00	1.11	99.92
256.bzip2	7	9	0	0	0	0	1.00	1.00	1.00	1.00	1.82	99.9
173.applu	22	25	2	1	0	0	0.99	1.00	1.00	1.00	0.62	100
181.mcf	51	51	360	251	171	98	1.01	1.01	1.01	1.01	0.20	99.92
176.gcc	31	32	28	22	2	1	0.99	1.00	1.00	1.00	1.21	99.88
171.swim	15	15	1	0	0	0	1.00	1.00	1.00	1.00	0.88	100
172.mgrid	19	35	23	10	4	2	1.07	1.00	0.99	1.00	0.87	99.98
Average							0.88	0.97	1.00	1.00	1.10	99.79

Table 3: Performance of an ULB-LSQ on an 80-window ROB machine.

Address-interleaved LSQs are a better match for address interleaved partitioned memory systems. However, address interleaving also means that there is no guarantee that the loads and stores will be distributed evenly across the partitions. Such an LSQ must function correctly even when all in-flight memory instructions map to the same partition. Such imbalance is uncommon, but may arise when the application is loading from or storing to a sequential array of characters. A system could tolerate the imbalance by sizing each partition for the worst case, but the total LSQ size would be N times the instruction window size, for N partitions. In the remainder of this paper, we explore better solutions that instead undersizes the LSQ partitions in the same manner as Section 4 and gracefully tolerates the rare overflow conditions with minimum effect on performance.

Related Work: Research proposals for clustered architectures [30, 9] employ multiple partitions of an age-indexed LSQ, but instead of reserving a slot in each of the LSQ partitions, they use memory bank predictors [25] to predict a target bank and reserve a slot there. If the bank prediction is low-confidence, slots are reserved in all banks. While this approach is better than conservatively reserving a slot in each partition, it still wastes space because of conservative dispatch allocation. The first memory disambiguation hardware to be address indexed was MultiScalar's Address Resolution Buffer (ARB) [17]. Loads and stores would index into this structure, where age tags were stored to assist in forwarding values and detecting ordering violations. The ARB caused a high overhead pipeline flush if it overflowed.

5.2 Large-window Processor Model

To examine the viability of the partitioned ULB-LSQ in a large-window processor, we use the TRIPS processor microarchitecture as a baseline. The TRIPS processor is a partitioned microarchitecture that enables a window of up to 1024 instructions and up to 256 in flight memory instructions. All major components of the processor are partitioned and distributed including fetch, issue, and memory access. The overflow handling mechanisms described in the rest of this paper are built on top of the TRIPS microarchitecture.

The processor 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 5.2 shows an abstract view of the TRIPS microarchitecture, highlighting the parts that are relevant to memory instructions. Additional details about the TRIPS microarchitecture can be found in [21].

The features of a distributed microarchitecture most relevant to the design of LSQs can be distilled down to a few principles which are not unique to TRIPS. First is an address-interleaved distributed cache in which multiple level-1 cache banks independently preserve the proper ordering of load and store instructions to the same address. Second is the set of distributed execution units which independently decides which instructions to issue each cycle. 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 techniques for mitigating overflows rely on buffering in-flight memory instructions within the network buffers. However, other interconnection networks can could be enabled to buffer in-flight memory instructions as well. While we examine partitioned LSQ design in a TRIPS context, we believe that these concepts apply to other architectures that share these basic characteristics.

Jaleel et al. point out that blindly scaling the larger window LSQs can be detrimental to performance due to the increase in the number of replay traps [18]. In their study on a scaled Alpha-21264 core, such traps can occur when load instructions violate the consistency model, when load needs to partially obtain the data from the LSQ and the cache, when a load miss cannot be serviced because of structural hazards and when a load instruction executes prematurely. TRIPS avoids these traps and does not suffer from the performance losses described in [18]. In particular, TRIPS avoids

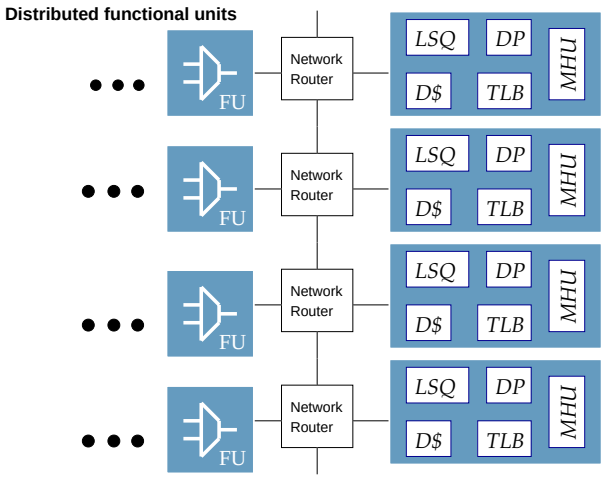

Figure 5: Overview of the distributed memory system in TRIPS: Each of the memory partitions includes a portion of the address-interleaved level-1 cache, a portion of the unordered LSQ, a local dependence predictor, a miss-handling unit, and a copy of the TLB.

load-load traps with weak memory ordering, wrong-size traps by supporting partial forwarding in LSQ, and load-miss traps by using larger MSHRs [23]. Like the Alpha, TRIPS also uses a dependence predictor to reduce the number of load-store replay traps which occur when a load instruction is executed prematurely.

6. MITIGATING LSQ OVERFLOWS

Ideally, a distributed LSQ should be divided into equal-sized banks, where the aggregate LSQ entries equals the average number of loads and stores in flight, but which shows only minor performance losses over a maximally sized LSQ. When a bank overflows, however, if the microarchitecture does not flush the pipeline, it must find someplace to buffer the load. We examine three places to buffer these instructions: in the execution units, in extensions to the memory units, or in the network connecting the execution units to the memory units. The buffering space is much less expensive than the LSQ space since the buffered locations need not be searched for memory conflicts, which mitigates the area and energy overheads of large LSQs. The penalty associated with these schemes correspond to different "load loops" and changes as the time for load execution changes [11].

These buffering approaches 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 microarchitecture that is block-oriented like 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.

6.1 Issue Queue Buffering: 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 6a 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.

A range of policies are possible for determining when to reissue 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. Increased network from NACKs can also 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 NACKed instructions until a fixed amount of time has elapsed. Waiting 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 has two minor overheads, however: an additional state bit for every reservation station, 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.

6.2 Memory Buffering: Skid Buffers

A second overflow-handling technique is to store memory instructions waiting to access the LSQ in a skid buffer located in the memory unit. As shown in Figure 6b, the skid buffer is simple priority queue 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 finds 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.

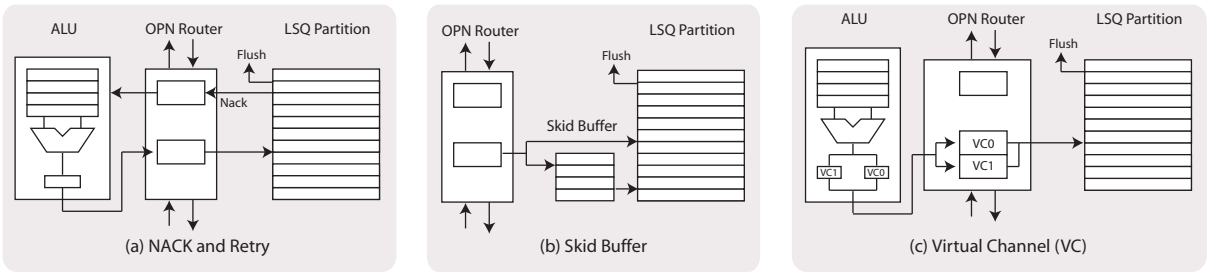

Figure 6: LSQ Flow Control Mechanisms.

6.3 Network Buffering: Virtual Channel-Based Flow Control

A third approach to handle overflows 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 6c 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 includes a number of 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 and the microarchitecture can flush any in-flight instructions located in any tile or network router when the block they belong to is flushed. 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 VCs 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 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 congested, 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 this 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.

6.4 Flow Control Mechanisms Performance

We implemented these flow control mechanisms on a simulator that closely models the TRIPS prototype processor [21] which has been validated to be within 11% of the RTL for the TRIPS prototype processor. The microarchitectural parameters most relevant to the experiments are summarized in Table 4.

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 these experiments, we used skid buffer sizes that are sized slightly larger than the expected number of instructions at each partition, (72 - LSQ partition size). For the virtual channel scheme, we divided the four operand network buffers in the baseline equally between the two channels. Thus two buffers are provided for the speculative and non-speculative virtual channels for the VC scheme. We present results for 28 EEMBC benchmarks (all except cjpeg and djpeg) and 12 SPEC CPU 2000 (ammp, applu, art, bzip2, crafty, equake, gap, gzip, mesa, mgrid, swim and wupwise) benchmarks with minnespec medium sized reduced inputs. The other benchmarks are not currently supported in our infrastructure.

For four 48-entry LSQs and thus a total LSQ size of 192 entries (25% undersized), the flush scheme results in average performance loss of 6% for EEMBC benchmarks (Figure 7) and 11% for SPEC benchmarks (Figure 8). The worst-case slowdowns are much higher: 180% for idct and 206% for mgrid. These results support the perspective that traditional flow-control mechanisms are inadequate for distributed load-store queues. The VC mechanism is the most robust with 2% average performance degradation and less than 20% performance degradation in the worst case. As expected, the Skid buffer scheme performs better than the NACK scheme because it avoids network network congestion from the NACKed packets, at the cost of extra area.

For six of the SPEC and EEMBC benchmarks, the memory accesses are unevenly distributed and cause LSQ overflows that re-

Parameter	Configuration
Overview	Out-of-order execution with up to 1024 instructions inflight, Up to 256 memory instructions can be simultaneously in flight. Up to 4 stores can be committed every cycle.
Instruction Supply	Partitioned 32KB I-cache 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.
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. Best case load-to-use latency is 5 cycles. Store forwarding latency is variable, minimum penalty is 1 cycle.
Interconnection Network	The banks are arranged in 5x5 grid connected by mesh network. 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 simulated with single simpoints of 100M

Table 4: Relevant aspects of the TRIPS microarchitecture

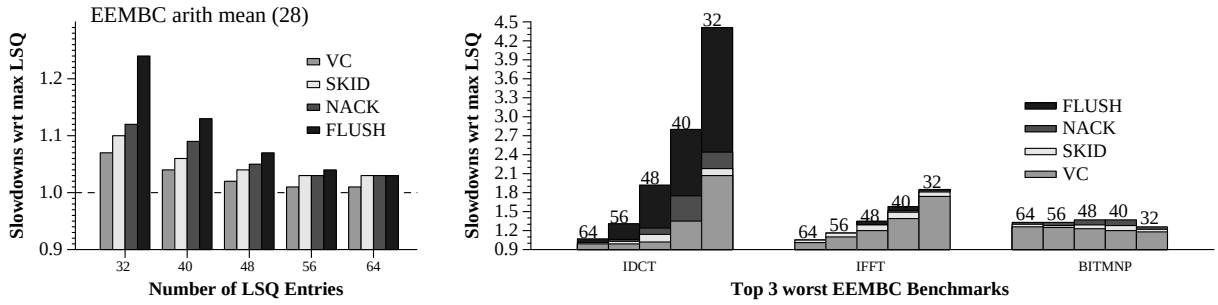

Figure 7: Left: Average LSQ Performance for the EEMBC benchmark suite. Right: Three worst benchmarks. bitmnp shows a different trend because there are fewer LSQ conflict violations in bitmnp when the LSQ capacity is decreased.

```
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 9: Code snippet from idct benchmark.

duce performance significantly. For instance, Figure 9 shows a frequently executed code sequence in idct in the EEMBC suite. The innermost loop contains two reads and two writes to two different arrays and the code generated by the compiler aligns both arrays to 256-byte boundaries. Since the arrays are accessed by same indices, all four accesses map to the same bank. This problem is exacerbated by the aggressive loop unrolling of the TRIPS compiler. The accesses could in theory be distributed by aligning the arrays differently, but aligning data structures to minimize bank conflicts is a difficult compiler problem.

6.5 Power Efficiency

Three mechanisms are necessary to achieve high power efficiency in large-window processor LSQs: address partitioning, late-binding and associative search filtering. First, partitioning the LSQ by addresses naturally divides the number of entries arriving at each memory partition. Second, late-binding reduces the number of entries in each partition by reducing occupancy. Finally, Bloom filtering reduces the number of memory instructions performing associative searches [22].

The Bloom filters for the TRIPS study use 8 32-bit registers, one for each in-flight block. The filters associated with each block are cleared when the block commits or is flushed (commonly called flash clearing). This clearing scheme does not require the counters often needed for Bloom filters and thus reduces the area of the filter significantly. As shown in Table 5, nearly 70-80% of the memory instructions (both loads and stores) can be prevented from performing associative searches in the TRIPS processor by using Bloom filtering.

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: Fraction of loads performing associative searches

Using Bloom filters, however, incurs additional some additional power for reading and updating the filters for every memory instruction. Using the 130nm ASIC synthesis methodology described in the Alpha evaluation section, the capacitance of the 48-entry TRIPS LSQ was computed to be 322pF. The capacitance of the Bloom filter was 64pF. With the activity reduction of 80% the effective capacitance of the combination is 120pF which roughly is the capacitance of a 12-entry, 40-bit unfiltered CAM.

In this evaluation, we have not included the additional power expended by the network routers as it is unclear if they will be signficant. Wang et al. [29] show a 20% increase in power for a

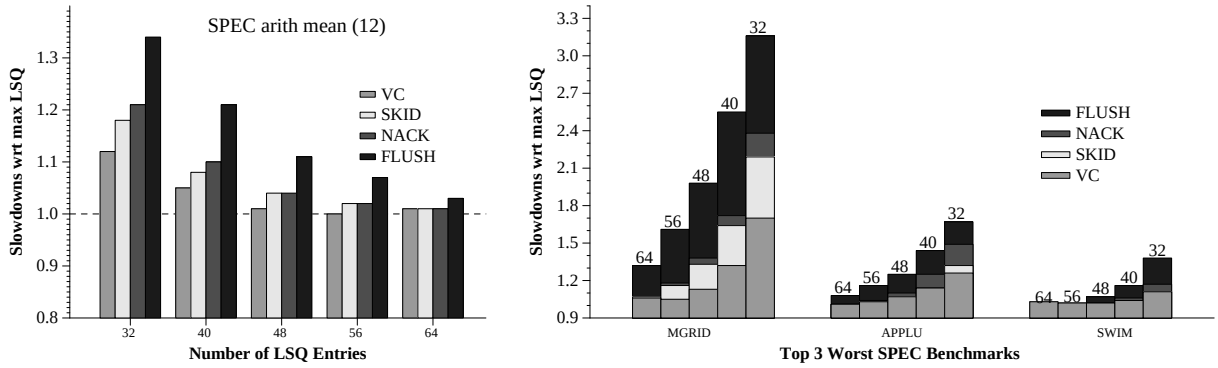

Figure 8: Left: Average LSQ Performance for the SPEC benchmark suite. Right: Three worst benchmarks.

four fold increase for implementing virtual channels. The power increase primarily comes from having four times as many buffers for implementing the Virtual channels. In our scheme we do not increase the number of buffers. We simply divide the number of buffers equally between the virtual channels.

6.6 Area

Among the three overflow handling mechanisms, the NACK mechanism is the most area efficient if the issue window is designed to hold instructions until explicit deallocation. On the TRIPS processor, the NACK scheme requires cumulative storage of 1024 bits to identify the NACK'ed instructions (one bit for every instruction in the instruction window) and 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 VCs 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 VC scheme does not require any additional router buffers since the speculative channels divide the number of buffers in the baseline. 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, increases the size of each LSQ partition by 4%. Overall, using our best scheme to support – the VC mechanism – a 1024 instruction window, as shown in Table 1, the area after optimizations is 80% smaller compared to the fully replicated LSQs at each LSQ partition.

6.7 Complexity

The VC scheme requires the most changes to the baseline as it requires virtualization of not only the network routers but also 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 changes required for the schemes described here, the mechanisms are feasible and operations required have been implemented in other parts of the processor.

7. CONCLUSIONS

Detecting load/store data dependences has been challenging ever since designers began to implement computer systems that execute more than one instruction per cycle. As processor designers have scaled ILP and instruction window size, hardware to enforce proper memory ordering has become large and power hungry. However, today's load/store queue (LSQ) designs make inefficient use of their storage by allocating LSQ entries earlier than necessary.

By performing late binding, allocating LSQ entries only at instruction issue, designers can reduce the occupancy and resultant size of the load/store queues. This reduction requires that the queues be unordered. While the unordered property requires some extra overhead, such as saving the CAM index in the ROB or searching for the load or store age at commit, the design is not intrinsically more complex, and can achieve performance equivalent to an ordered LSQ, but with less area and power.

Many of the recent papers that propose to eliminate the CAMs do so at the expense of increased state, resulting in increased area requirements. These designs certainly have some advantages, and one or more of them may well be the solution that some implementations eventually use. However, we have shown in this paper that unordered, late-binding LSQs can be quite competitive for superscalar designs, requiring only a small number of CAM entries to be searched on average, while consuming less area than the CAM-free approaches. It is unclear which approach suffers from worse complexity, and will likely remain so until each approach is implemented in a microarchitecture.

However, the most promising aspect of the ULB-LSQ approach is its partitionability, which was the original impetus for this line of research. Address-interleaved LSQ banks should be both late-bound and unordered; the ULB-LSQ design naturally permits the LSQ to be divided into banks, provided that a mechanism exists to handle the resultant increase in bank overflows. We observed that, for distributed microarchitectures that use routed micronetworks to communicate control, instructions, and data, that we could embed classic network flow-control solutions into the processor micronetworks to handle these overflows. We evaluate three such overflow control handling schemes in the context of the TRIPS microarchitecture. The best of these schemes (virtual micronet channels) enables a scalable, distributed, load/store queue, requiring four banks of only 48 entries each to support a 1024-instruction window.

When conjoined with a Bloom filter, this design greatly reduces the number of CAM accesses, resulting in an average of only eight and twelve CAM accesses per load or store, respectively.

Looking forward, this design supports a microarchitecture under design that can run a single thread across a dynamically specified collection of individual, light-weight processing cores. When combined, the LSQ banks in the individual cores become part of a single logical interleaved memory system, permitting the system to choose between ILP and TLP dynamically using a collection of composable, light-weight cores on a CMP.

8. ACKNOWLEDGMENTS

We thank M.S. Govindan for his help with the power measurements and K. Coons, S. Sharif and R. Nagarajan for help in preparing this document. This research is supported by the Defense Advanced Research Projects Agency under contract F33615-01-C-4106 and by NSF CISE Research Infrastructure grant EIA-0303609.

9. REFERENCES

[1] E. F. Torres and P. Ibanez and V. Vinals and J. M. Llaberia. Store Buffer Design in First-Level Multibanked Data Caches. In *ISCA*, 2005.

[2] F. Castro, L. Pinuel, D. Chaver, M. Prieto, M. Huang, and F. Tirado . DMDC: Delayed Memory Dependence Checking through Age-Based Filtering. In *MICRO*, 2006.

[3] Sam S. Stone and Kevin M. Woley and Matthew I. Frank. Address-indexed memory disambiguation and store-to-load forwarding. In *MICRO*, 2005.

[4] Samantika Subramaniam and Gabriel H. Loh. Fire-and-Forget: Load/Store Scheduling with No Store Queue at all. In *MICRO*, 2006.

[5] H. Akkary, R. Rajwar, and S. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 423–434, December 2003.

[6] Alok Garg and M. Wasiur Rashid and Michael Huang. Slackened Memory Dependence Enforcement: Combining Oppurtunistic Forwarding with Decoupled Verfication. In *ISCA*, 2006.

[7] Alper Buyuktosunoglu and David H. Albonesi and Pradip Bose and Peter W. Cook and Stanley E. Schuster. Tradeoffs in Power-Efficient Issue Queue Design. In *ISPLED*, 2002.

[8] Amir Roth. High Bandwidth Load Store Unit for Single- and Multi-Threaded Processors. Technical Report MS-CIS-04-09, Dept. of Computer and Information Sciences, University of Pennsylvania, 2004.

[9] R. Balasubramonian. *Dynamic Management of Microarchitecture Resources in Future Microprocessors*. PhD thesis, University of Rochester, 2003.

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

[11] E. Borch, E. Tune, S. Manne, and J. Emer. Loose loops sink chips. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, pages 299–310, 2002 February.

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

[13] H. W. Cain and M. H. Lipasti. Memory ordering: A value-based approach. In *ISCA*, 2004.

[14] A. Cristal, O. Santana, F. Cazorla, M. Galluzzi, T. Ramirez, M. Pericas, and M.Valero. Kilo-instruction processors: Overcoming the memory wall. *IEEE Micro*, 25(3):48–57, May/June 2005.

[15] R. Desikan, D. Burger, and S. W. Keckler. Measuring experimental error in microprocessor simulation. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 266–277, July 2001.

[16] Elham Safi and Andreas Moshovos and Andreas Veneris. L-CBF: A Low Power, Fast Counting Bloom Filter Implementation. In *ISPLED*, 2006.

[17] 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.

[18] A. Jaleel and B. Jacob. Using Virtual Load/Store Queues (VLSQs) to Reduce the Negative Effects of Reordered Memory Instructions. In *HPCA*, 2005.

[19] T. Monreal, A. Gonzalez, M. Valero, J. Gonzlez, and V. Vinals. Delaying Physical Register Allocation Through Virtual-Physical Registers. In *MICRO*, 1999.

[20] A. Roth. Store vulnerability window (svw): Re-execution filtering for enhanced load optimization. In *ISCA*, 2005.

[21] K. Sankaralingam, R. Nagarajan, R. McDonald, R. Desikan, S. Drolia, M. Govindan, P. Gratz, D. Gulati, H. Hanson, C. Kim, H. Liu, N. Ranganathan, S. Sethumadhan, S. Sharif, P. Shivakumar, S. W. Keckler, and D. Burger. Distributed microarchitectural protocols in the TRIPS prototype processor. In *MICRO*, December 2006.

[22] S. Sethumadhavan, R. Desikan, D. Burger, C. R. Moore, and S. W. Keckler. Scalable memory disambiguation for high ilp processors. In *36th International Symposium on Microarchitecture*, pages 399–410, December 2003.

[23] Simha Sethumadhavan and Robert McDonald and Rajagopalan Desikan and Doug Burger and Stephen W. Keckler. Design and Implementation of the TRIPS Primary Memory System. In *ICCD*, 2006.

[24] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual flow pipelines. In *ASPLOS*, pages 107–119, October 2004.

[25] Stefan Bieschewski and Joan-Manuel Parcerisa and Antonio Gonzalez. Memory Bank Predictors. In *ICCD*, 2005.

[26] J. M. Tendler, J. S. Dodson, J. J. S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Journal of Research and Development*, 26(1):5–26, January 2001.

[27] Tingting Sha and Milo M. K. Martin and Amir Roth. Scalable store-load forwarding via store queue index prediction. In *MICRO*, 2005.

[28] Tingting Sha, Milo M.K. Martin and Amir Roth. NoSQ: Store-Load Communication without a Store Queue. In *MICRO*, 2006.

[29] H.-S. Wang, X. Zhu, L.-S. Peh, and S. Malik. Orion: a power-performance simulator for interconnection networks. In *MICRO*, pages 294–305, 2002 December.

[30] V. V. Zyuban. *Inherently Lower-Power High-Performance Superscalar Architectures*. PhD thesis, University of Notre Dame, March 2000.