Decomposing the Load-Store Queue by Function for Power Reduction and Scalability

Lee Baugh and Craig Zilles

Dept. of Computer Science, University of Illinois at Urbana-Champaign
E-mail: {leebaugh, zilles}@cs.uiuc.edu

Abstract

Because they are based on large content-addressable memories, load-store queues (LSQ) present implementation challenges in superscalar processors, especially as issue width and number of in-flight instructions are scaled. In this paper, we propose an alternate organization of an LSQ that separates the forwarding functionality from checking that loads received their correct values. Two main techniques are exploited: 1) the store forwarding logic is only accessed by those loads and stores that are likely to be involved in forwarding, and 2) the checking structure is banked by address. The result of these techniques is that a small collection of small, low bandwidth structures can be substituted for the large, high bandwidth structures used in conventional designs. By our calculations, these proposed techniques reduce LSQ dynamic power by a factor of 3-5 while achieving equivalent performance.

1. Introduction

In a dynamically-scheduled processor, the load-store unit is typically implemented by composing a translation-lookaside buffer, a cache, and a load-store queue (LSQ). The LSQ typically provides the following four functions:

1. buffering store addresses and values for in-order retirement
2. forwarding in-flight store values to loads
3. detection of load/store ordering violations
4. detection of consistency violations

Commonly, the LSQ is implemented as a pair of age-ordered queues—one each for loads and stores—that can be associatively searched by address. This organization presents a scalability challenge to increasing superscalar width and number of in-flight instructions: increasing the number of ports (for increased width) and the number of entries (for in-flight instructions) significantly impacts the access time and power consumption of the structure.

The access time of the store queue is particularly critical because it is a component of the load-to-use latency. Typically, snooping the store queue must be performed in the same amount of time as the L1 data cache access, which is done in parallel, to avoid further complication of the instruction scheduler. Using Cacti[15], Roth estimates that only an 8-16 entry queue can be snooped in the time to access a 32KB 4-way interleaved, 2-way set-associative cache with 32B blocks [13], a severely limiting constraint.

In this work, we propose an LSQ organization that decouples the performance-critical store-bypassing logic from the rest of the load-store queue functionality. This organization is motivated by two insights:

1. **Store value bypassing is the only time-critical operation performed by the LSQ.** All other functions merely need to be performed before the instructions retire.
2. **Only a small and predictable fraction of loads and stores take part in store value bypassing.**

For store bypassing, we propose using a structure—the *store forwarding buffer* (SFB)—which is much like a traditional store queue but has fewer entries and fewer ports, yielding a reduction in access time and a significant reduction in power consumption. The structure size is reduced by allocating entries for only those stores predicted to bypass. Likewise, required bandwidth is reduced by only snooping for those loads that are predicted to require a bypass. Because these predictions can be wrong, a mechanism is required to detect misspeculations.

A second structure, the *memory validation queue* (MVQ) detects load-store ordering violations, coherence violations, and bypass mispredictions. This structure must observe all loads and stores to identify violations. To efficiently implement this structure, we bank it by address, achieving a large aggregate throughput and storage capacity through a collection of small, single-ported structures. Such banking provides scalability and reduced energy consumption at the cost of potential imbalance between banks. To tolerate bank conflicts, we decouple processing in the MVQ from instruction execution through the addition of a small wait queue. Validation is tolerant of queuing delay, because it merely needs to take place before the associated instructions commit. To avoid over-subscribing one bank, we provide an execution throttling mechanism that minimizes squashes based on the availability of wait queue entries.

Specifically, the contributions of this work are threefold:

1. We demonstrate that static instructions are invariant with regards to their desire to bypass, enabling simple and effective filters to limit store forwarding buffer (SFB) traffic.

2. We describe a load-store queue design that decouples store forwarding from other LSQ functions, decomposing the LSQ into small, low-bandwidth (and hence fast) *store forwarding buffer* and a latency tolerant *memory validation queue*, which can be made efficient and high-throughput through banking.

3. We demonstrate the mechanisms required to achieve good utilization of banked LSQ structures while minimizing squashing.

This paper is organized as follows. In Section 2, we describe the organization of our proposed LSQ design. In Section 3, we describe our experimental methodology and results. In Sections 4 and 5, we describe related and future work, and, in Section 6, we conclude.

2. Organization

In this section, we describe our proposed LSQ organization. Because we use LSQs as a building block of our design, we first describe their salient details. We then describe the two components of our proposed LSQ design in Sections 2.2 and 2.3, respectively.

2.1. Age-ordered Load/Store Queues

The most common implementation of a LSQ involves a pair of buffers (one for loads and one for stores) that hold instructions in program order (*i.e.*, "age-ordered"); see Figure 1. Instructions are allocated entries in their respective queues before dispatch into the instruction window (dispatch stalls if entries are not available). When instructions execute, they write their address (and value for stores) into their allocated entry. In parallel, they perform an associative search of the other queue, comparing addresses. If a store matches a load later in program order, a squash is signaled. If a load matches with one or more stores earlier in program order, the index of the youngest is selected (using a priority encoder, a process greatly facilitated by age ordering) and used to drive a RAM array that holds the store's value.

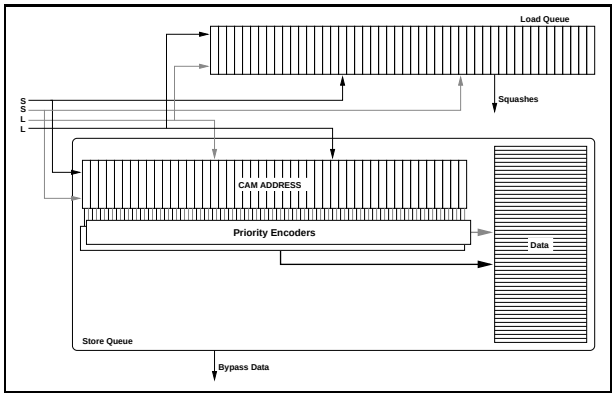

Figure 1. Traditional Monolithic Load/Store Queue Design. *The store bypass path, which involves a CAM lookup, priority encoder, and RAM access, is generally one of a processor's critical paths. A datapath that can support up to 2 loads and/or 2 stores per cycle is shown.*

Because all loads and stores are placed in the LSQ, each queue must be appropriately sized to allow good utilization of the reorder buffer even for instruction mixes rich in loads or stores. In recent processors, the queues have been sized to hold 25-40 percent of the maximum number of in-flight instructions (Alpha 21264: 32 loads/32 stores, 80 in-flight instructions [10], Pentium 4: 48 loads/32 stores, 128 in-flight instructions [5]).

2.2. Store Forwarding Buffer

As described in Section 1, we streamline the performance critical store queue by only using it for those instructions that require it. In Figure 2, the black bars show

the fraction of dynamic loads and stores that *matched* in the LSQ and hence required forwarding, for a machine with a 256-entry instruction window. On average, only 7 percent of dynamic loads and 20 percent of dynamic stores are involved in forwarding in our runs.

Because an instruction's disposition to forward or not is a property of the program, the instruction's PC can be used to segregate those instruction likely to forward from those that are not. Specifically, we find that a large fraction of static instructions are never involved in forwarding. Thus a single bit per static instruction is sufficient to effectively predict an instruction's forwarding behavior; all bits are initially cleared and an instruction's bit is set when it is first detected to require forwarding. This simple predictor is very effective for loads (filtering out 70%) and moderately effective for stores (filtering out 40%), as shown in Figure 2. As there are generally more loads than stores, it is desirable that more loads are filtered than stores.

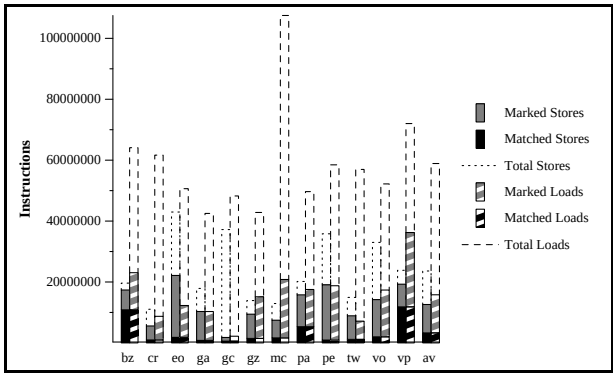

Figure 2. Only a small, predictable fraction of memory instructions is involved in bypassing. *Fraction of all loads and stores executed, we break them down into those that* ***matched*** *in the LSQ, those that are* ***marked*** *because previous instances of their static instruction matched, and those belonging to static instructions that never matched.*

Ideally, this prediction bit is stored in the instruction—an option when defining a new ISA or dynamically translating to an internal ISA [1, 7, 8, 11, 16]—because then the behavior only has to be learned once. Alternatively, this prediction can be implemented by associating an extra bit with each instruction in the I-cache. To handle programs with large working sets (not a problem for our SPEC2000 benchmarks) it may be beneficial to "page" these predictions into L2 ECC bits, as is done in the AMD Opteron with branch predictor information [9].

Once these predictions are available, the operation of the SFB is much like that of a traditional store queue. Like traditional systems, stores allocate entries in the age-ordered SFB prior to dispatch into the instruction window; the only difference is that only those stores predicted to require bypassing—what we call *marked* stores—need to allocate an entry.

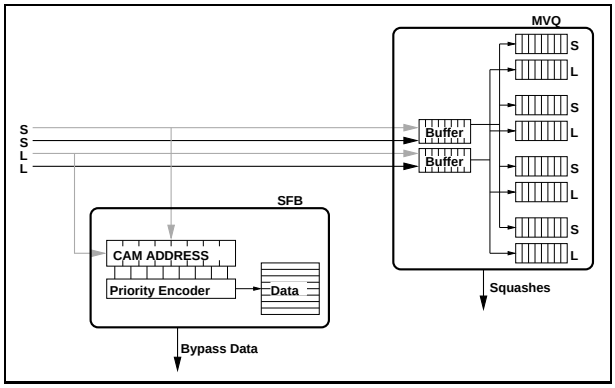

Figure 3. Proposed Decoupled Load/Store Queue Design. *The performance critical store forwarding buffer (SFB) is used by only those instructions likely to be involved in store forwarding, reducing its access time and energy consumption. The energy consumed by the rest of the LSQ functionality is reduced by banking it into a number of small, low-bandwidth structures. A datapath that can support up to 2 loads and/or 2 stores per cycle is shown, but only 1 load and 1 store can access the SFB.*

As only a fraction of loads and stores are marked, it is unnecessary (as our performance results show) to provide bandwidth to the SFB equal to the issue width for memory instructions. Thus only a subset of load/store units need be provided with ports to the SFB. Marked instructions must be slotted/scheduled to execute only on those function units.

2.3. Memory Validation Queue

As the SFB only provides the forwarding functionality of a traditional LSQ, additional structures are required. Buffering store values for in-order retirement is relatively straight-forward; two reasonable implementations are possible: 1) a separate (non-associative) RAM structure to hold addresses and values, or 2) using such a structure for unmarked stores in conjunction with the SFB. To handle the remaining functionality we provide a Memory Validation Queue (MVQ).

The role of the MVQ is to ensure that loads receive the correct value by forcing pipeline squashes when neces-

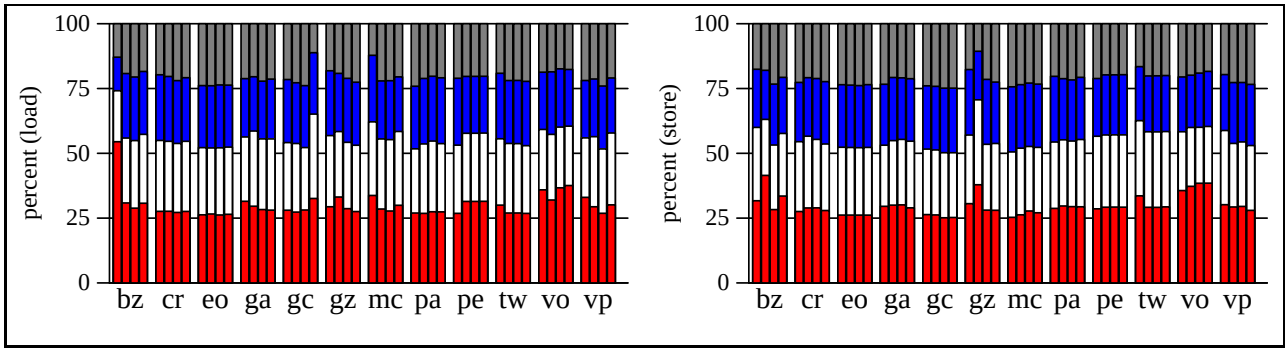

Figure 4. Address-based hashing can be used to partition dynamic memory instructions into roughly equal groups. *Fractions are sorted from largest to smallest, bottom to top. Data from four 100M instruction intervals are shown for each benchmark, starting at 0B, 3B, 5B, and 8B instructions.*

sary. In addition to the detection of load-store ordering and consistency violations required of traditional load queues, the MVQ must detect situations where load-store forwarding should have been performed on unmarked loads and/or stores.

We show a schematic diagram of an MVQ in Figure 3. It includes a buffer (or set of buffers, divided by instruction type or address hash) and a set of banks, each consisting of a pair of single-ported CAMs, one to hold loads and one to hold stores. The bank an instruction will end up in is determined by an address hash on the memory value. Unlike a conventional LSQ, instructions may be introduced into the MVQ out-of-order. Upon entering the MVQ, an instruction is inserted into an appropriate buffer, and waits to be serviced. When a bank becomes idle, it queries the buffer(s) for the oldest instruction with the appropriate hash, and removes that instruction from the buffer. If a store, the instruction is inserted into the bank's store queue and snoops the bank's load queue for matches; if a load, it is inserted into the bank's load queue and snoops the bank's store queue for matches. The following snooping circumstances may lead to a squash:

- A store snoops a matching load which, though issued earlier, is later in program order. The core is always squashed back to the load, and both instructions are marked as likely to match.
- A load snoops a matching store which was earlier in both issue and program order. If either instruction is not marked, then the store's value has not been forwarded; the core is squashed back to the oldest unmarked instruction in program order, and both instructions are marked as likely to match.

If several squashes arise, the core is squashed back to the oldest squashing instruction. Squashed instructions – whether squashed from the MVQ or for other reasons – are removed from MVQ whether they be in a queue or a buffer. To ensure that instructions do not retire before they are processed by the MVQ, memory instructions are not marked as completed in the reorder buffer until the MVQ validates their execution.

Thus, in many respects, the MVQ acts like a traditional LSQ, but, by virtue of factoring out the performance-critical store-forwarding logic, the structure becomes latency tolerant, enabling an energy efficient implementation. The primary technique that we exploit to simplify the implementation is banking by address. Banking allows a collection of small, low-bandwidth structures to be used as a single, large, high-throughput structure. The reduction of structure size and number of ports significantly reduces energy consumption, as we discuss in Section 3.

The most obvious drawback of banking is the potential for load-balancing problems, but we have found this to be a minor problem in practice. By using a hash function that incorporates many (*e.g.*, 16) address bits, we find that problems resulting from strided accesses can be minimized. Figure 4 shows that a relatively even distribution can be achieved in most cases (data shown for 4 banks, interleaving at the granularity of a 64b word[1], hashing bits [18:3] of the address). In general, the address distribution is remarkably constant over time. In the few cases (*e.g.*, the first sample from `bzip2`) where the distribution is skewed, we can attribute it to the existence of a small number of "hot" addresses (data not shown), and thus could not be avoided by the selection of a different hash function.

1 Banking is facilitated by the constraint in some RISC architectures (*e.g.*, Alpha, which we used in our simulations) that memory operations be naturally aligned. For Alpha, this restriction means no operation spans 64b boundaries. For ISA's without this restriction (*e.g.*, x86) support for insertion into multiple banks would be required.

2.3.1. Challenges Due to Banking The true challenges resulting from banking arise from addresses (and hence bank indices) not being available until execution time, namely: 1) MVQ entries cannot be allocated at dispatch time, making it difficult to manage the structure in an age-ordered manner, 2) bank conflicts can arise from simultaneously issuing multiple instructions destined for the same bank, and 3) it is difficult to guarantee that one bank will not be over-subscribed.

We address the first challenge by not using an age-ordered queue. Age ordering serves primarily two purposes: 1) simplification of priority encoding, and 2) simplification of the management of queue resources. As a priority encoder is only required when multiple matches occur, this is not a time critical operation (as it is in the SFB) because MVQ squashes are relatively rare. Performing priority selection sequentially (one cycle per compare) reduces performance by less than 0.001 percent in all cases.

The second challenge, that of bank conflicts, is easily addressed by placing a pair of small buffers (see Figure 3) to smooth out instantaneous bank imbalance. The addition of this buffer increases the latency of an MVQ insertion/snoop by a potentially variable amount (based on the number of recent conflicts), but, as the MVQ is only used to signal squashes, it is latency insensitive and its latency need not be predictable.

The third challenge is the most difficult, as there is a tension between fully utilizing the MVQ and not over subscribing any one bank. Our primary mechanism is to only issue memory instructions when there is space available in the buffer. This scheme is relatively easy to implement at the scheduler by tracking how many buffer entries have been allocated but not freed: we increment a counter when a memory instruction issues and decrement it when an instruction is removed from the buffer. To allow the maximum instruction throughput, the number of buffer entries must exceed the number of pipeline stages between issue and address generation scaled by the memory instruction issue width.

While this approach avoids oversubscribing the MVQ, it has the potential for deadlock if issue gets stalled (because the buffer is full due to 1 or more ways of the MVQ filling) and the oldest memory instruction hasn't issued. To reduce the likelihood of this occurance we limit the number of loads and stores dispatched into the instruction window to be slightly less than can be held in the MVQ proper (anticipating some imbalance). By throttling the number of memory instructions entering the window, we reduce the likelihood that an MVQ bank fills before a stalled instruction executes. In all but the smallest MVQ's the number of deadlock occurances is minimal, allowing a slow mechanism to be used for their detection. Upon detecting a deadlock the whole pipe is flushed, as the oldest instruction cannot make further progress.

3. Methodology and Results

We evaluated our proposed load-store queue design using timing simulations of SPEC2000 integer benchmarks. Our timing simulator uses the loader and system call functionality from SimpleScalar [3], but the pipeline model has been re-written to perform a true execution-driven simulation of Alpha binaries. Parameters for our simulated machine can be found in the table in Figure 5. Benchmarks are compiled with the Compaq Alpha compiler at the highest level of optimization, but without profile information. All the results presented in this paper are for 200 million instruction runs started after skipping the first 5 billion instructions.

Scheduler & Pipeline
4-issue, 12-stage pipeline, 256-entry instruction window, 4k gshare predictor with 8 bits of history.
Memory
64kB 2-way associative L1 instruction & data cache with 1 cycle latency, 1MB 8-way associative L2 cache with 20 cycle latency, 80-cycle memory latency.
Functional Units (latency)
4 Integer ALUs (1), 1 Integer MULT/DIV (3/12), 2 Memory ports (3 ld/2 st), 2 FP ALUs (2), 1 FP MULT/DIV (4/12)

Figure 5. Simulation Parameters.

In this section, we first demonstrate (in Section 3.1) that filtering based on previous forwarding behavior significantly reduces the required number of entries and ports on the store forwarding structure, relative to a traditional LSQ. We then show (in Section 3.2) that our throttling mechanisms are sufficiently effective to approximate the performance of an ideal MVQ with an MVQ with 4 banks, each with 16 entries. We conclude this section by demonstrating that an SFB/MVQ design enables equivalent performance with a 3-to-5-fold reduction in dynamic power.

3.1. Varying Store Queue Size and Ports

In a system using a conventional LSQ, performance is closely related to the LSQ's parameters. In particular, reducing the capacity of the queues, or the number of ports they expose, severely limits performance,

beneath some threshold which depends on the application and input set. In Figure 6, the load and store queue capacity is varied between 8 and 64 entries with both single read/write ported and double read/write ported queues. With the conventional LSQ—drawn with solid lines—single read/write ported performance trails double read/write ported by 5.8% with a capacity of 64 elements, and performance begins to drop drastically when the capacity is reduced below 32 elements, attaining a 13% slowdown by 8 elements. As instruction windows and memory issue widths increase, we expect these trends to continue. However, in a system equipped with an ideal (*i.e.*, unlimited capacity) MVQ, single read/write ported performance differs from double read/write ported by 1.5% at 64 entries, and the 64-entry queue performs only 1.8% better than the 8-entry queue.

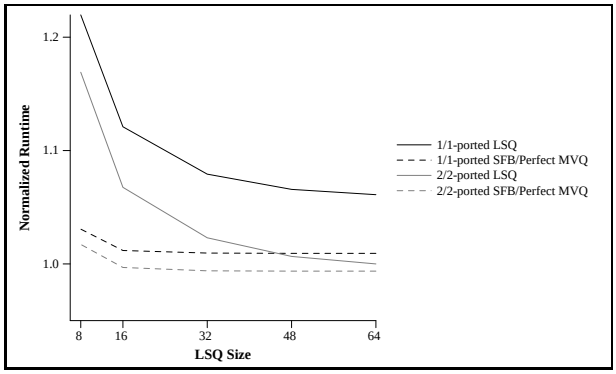

Figure 6. A conventional system is sensitive to its store queue parameters; an MVQ-equipped system is relatively insensitive. *Data averaged across samples of the SPEC2000 integer benchmarks, normalized to a 64-entry 2/2-ported LSQ.*

3.2. Varying MVQ Parameters

In Figure 6, an MVQ with unlimited capacity was used. As this is not practical, we consider the effect on performance of varying its properties. We change two factors: the number of banks in the MVQ and the capacity of both the store and the load queue in each bank. Results in Figure 7 show that four banks are required to provide sufficient bandwidth (recall that each bank can process only a single load or store per cycle), but performance is reasonable with queues as short as 16 entries. Increasing the number of banks beyond four only helps when each bank is shrunk to less than 16 entries, because it increases the aggregate number of entries. Thus the MVQ can be implemented with 4 banks each with

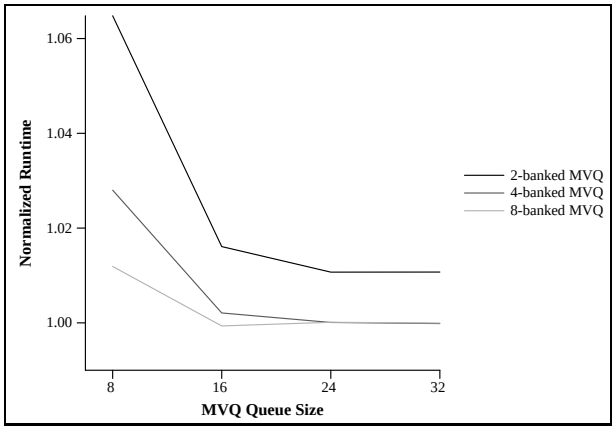

Figure 7. The MVQ requires 4 banks and 16 entries/bank, but benefits from little more. *Data averaged across the SPEC2000 integer benchmarks, normalized to the case with a perfect MVQ.*

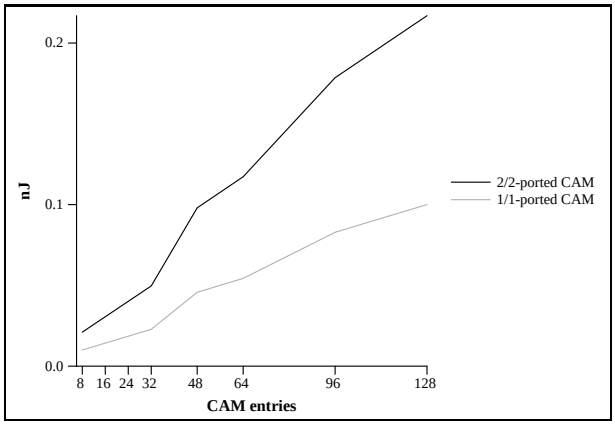

Figure 8. Query power in CAMs varies with the number of ports and the number of elements. $.09\mu$ *technology. Produced by Cacti 3.2*

16 entries (4x16) with minimal performance reduction, which we show significantly reduces power in the next section.

3.3. MVQ Power

Our previous results have shown that our SFB/MVQ combination can achieve performance comparable to a monolithic LSQ, but with a collection of smaller, low bandwidth structures. While this modestly reduces the access time of each structure, it provides a substantial power reduction. According to Cacti 3.2, energy per access for a CAM scales roughly linearly with both the number of entries and number of ports (see Figure 8). Thus querying a 48-entry 2-read/2-write-ported CAM

takes almost 7 times the power of a query to a 16-entry 1-read/1-write-ported CAM.

As a result, the use of smaller structures translates into as much as a 5-fold reduction in LSQ power. Accounting for the fact that the marked instructions access both the SFB and the MVQ, we computed energy consumption for both the LSQ and SFB/MVQ organizations. We looked at two performance points: a 2-ported 48-entry LSQ has roughly the same performance as a 2-ported 16-entry SFB with a 4x24-entry MVQ and a 2/2-ported 32-entry LSQ has roughly the same performance as a 1/1-ported 16-entry SFB with a 4x16-entry MVQ. Figure 9 shows the SFB/MVQ achieves roughly a 5-fold and 3-fold reduction of dynamic power, respectively.

A back-of-the-envelope calculation suggests our design compares equivalently or favorably to conventional LSQ designs in terms of static power and area. To analyze both the static power consumption and area requirements of our design we have made an estimation of the transistor count of both the traditional design and the SFB/MVQ[2]. This estimate suggests that a 48 entry 2/2 ported LSQ architecture uses roughly the same number of transistors as a 16 entry 2/2 ported SFB together with a 4-banked, 16-entry per queue MVQ. The latter architecture, by virtue of using structures with fewer ports, uses fewer wires, and the MVQ could employ slower, lower-leakage, smaller fabrication transistors without significant performance penalty. As the MVQ dominates the transistor count of our architecture, we expect this to favorably influence static power and area.

4. Related Work

There has been a lot of recent work in the design of load-store queues. The closest related work to ours is that of Roth, who independently made the observation that not all loads and stores need to be considered for forwarding [13]. To handle the non-forwarding related operations of the LSQ, he proposes to use filtered load re-execution, as was proposed by Cain and Lipasti [4], which eliminates the necessity of a load queue at the expense of re-executing a fraction of loads at retirement.

Sethumadhavan et al. previously considered address-banked LSQ designs, but discarded them because they failed to achieve good results [14]. There are three key differences between their proposal and ours: 1) we propose banking only the latency-tolerant verification portion of the LSQ, which can tolerate a buffer to smooth out bank conflicts, 2) our throttling mechanism can be viewed as a hybrid of their stalling and squashing mechanism, which minimizes the number of squashes required without being overconservative, and 3) our primary site of throttling is at issue rather than dispatch. With these difference we were able to achieve very positive results.

Akkary et al. observe that most forwarding occurs between relatively nearby instructions and propose building a scalable store queue by exploiting hierarchy [2]. Recent instructions are cached in a first-level store queue, with other instructions residing in a second-level structure. This approach reduces the size, but not the bandwidth of the latency critical store queue, but reduces latency predictability.

Park et al. propose reducing the required snoop bandwidth of the store queue by extending a Chrysos and Emer store set predictor [6] to predict which loads are likely to receive values forwarded by stores [12]. Our scheme achieves an equivalent reduction in snoops with a much simpler predictor.

Sethumadhavan et al. proposed using a Bloom filter to reduce store queue snoop bandwidth requirements, by eliminating those searches that the Bloom filter predicts cannot possibly match [14]. This approach has two drawbacks relative to our proposal: 1) accessing the Bloom filter is on the critical path (*i.e.*, it must be done between generating an address and accessing the store queue), and 2) an instruction's need to snoop is not known until execution time, so it is not available to the scheduler. As a result, the scheduler must either be conservative or risk over-loading the store queue ports, requiring queuing and latency mispredictions.

5. Future Work

This work was initiated in the context of designing an efficient LSQ for the Master processor in MSSP [16]. Since the MSSP Master has minimal correctness requirements—its results are used merely as value predictions by the slave processors—there stands a great opportunity for hardware simplification as structures need only work correctly "most of the time". We shifted, in this work, to a superscalar focus when we realized that our basic architecture was beneficial when guaranteed correctness is required.

With the basic organization laid out, reduced correctness requirements enable the following future work: First, sampling can be used to reduce the necessary MVQ size and bandwidth, as detected violations are rare, ex-

2 We assumed that one bit of storage is 4 transistors, and each write port requires 2 extra transistors per bit, while each CAM read port requires 3 extra transistors per bit.

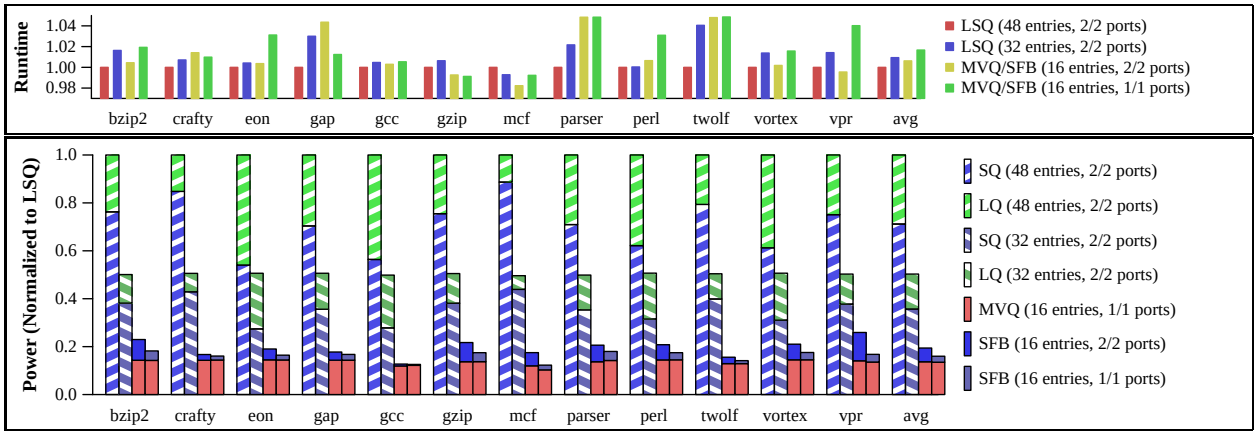

Figure 9. Designs using the SFB/MVQ can achieve similar performance using smaller, lower bandwidth structures, yielding lower power execution. *Data shown for 48- and 32-entry LSQs (striped) with 2 load and 2 store ports for each queue. The MVQs (solid) has an 16-entry buffer and 4 banks, with each bank having 16 entries and one read and one write port; the left MVQ features a 16-entry 2/2-ported SFB, while the right MVQ features a 16-entry 1/1-ported SFB.*

cept during the initial period when predictors are being trained. Second, because the Master's program is dynamically generated, we can transform the code to convert some of the memory communication into register communication, thereby reducing the load on the SFB, as well as eliminating those instructions that cause load balancing problems for the banked MVQ. Finally, we plan on exploring other mechanisms for increasing the bandwidth of the L1 memory system (*e.g.*, speculatively partitioning into two or more non-overlapping subsets).

Three potential superscalar enhancements remain, as well: 1) improving prediction accuracy of the marking mechanism, 2) reducing the number of squashes by forcing all instructions to snoop/be inserted in the SFB on their first execution (as indicated by an instruction cache miss), and 3) explore hashing the address in the MVQ to a smaller number of bits, potentially trading off a small number of unnecessary squashes for a large reduction in area and power.

6. Conclusion

Scaling traditional load-store queue (LSQ) designs presents a pressing problem for architects, as the content-addressable memories on which they are based scale poorly with regards to access time and complexity. In this paper, we have proposed an alternative for the traditional LSQ in which its several functions are decomposed and distributed so that critical value forwarding happens in a fast structure and correctness is removed from the critical path. We simplify the store forwarding logic by restricting the store queue to hold and snoop only those instructions predicted to be involved in forwarding. We simplify the checking functionality of the LSQ by implementing it in a physically distributed structure, called the Memory Validation Queue (MVQ). Having demonstrated that hashing data addresses can effectively partition memory instructions in the common case, we demonstrate how the MVQ can be banked and propose throttling techniques for dealing with load imbalance between the banks. The end result of this design is that a traditional monolithic LSQ can be replaced with a collection of small, low bandwidth structures with a negligible loss in performance. These smaller structures offer significant savings in power and modest improvements in access time, making the SFB and MVQ a practical alternative for future processors.

References

[1] V. Adve, C. Lattner, M. Brukman, A. Shukla, and B. Gaeke. LLVA: A Low-level Virtual Instruction Set Architecture. In *36^{th} Int'l Symp. on Microarchitecture*, pages 205–216, San Diego, CA, Dec 2003.

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

[3] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, Feb. 2002.

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

[5] M. F. Chowdhury and D. M. Carmean. Method, appartus, and system for maintaining processor ordering by checking load addresses of unretired load instructions against snooping store addresses. U.S. Patent Application Number 6,484,254, assigned to Intel, 2000.

[6] G. Z. Chrysos and J. S. Emer. Memory dependence prediction using store sets. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 142–153, May 1999.

[7] K. Ebcioglu and E. R. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 26–37, June 1997.

[8] B. Fahs, S. Bose, M. Crum, B. Slechta, F. Spadini, T. Tung, S. J. Patel, and S. S. Lumetta. Performance characterization of a hardware framework for dynamic optimization. In *Proceedings of the 34th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2001.

[9] C. N. Keltcher, K. J. McGrath, A. Ahmed, and P. Conway. The amd opteron processor for multiprocessor servers. *IEEE Micro*, 23(2):66–76, March/April 2003.

[10] R. E. Kessler. The alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1999.

[11] A. Klaiber. The technology behind crusoe processors. Transmeta Whitepaper, Jan. 2000.

[12] I. Park, C. liang Ooi, and T. N. Vijaykumar. Reducing design complexity of the load-store queue. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2003.

[13] A. Roth. A high-bandwidth load-store unit for single- and multi- threaded processors. Technical report, University of Pennsylvania, 2004.

[14] S. Sethumadhavan, R. Desikan, D. Burger, C. R. Moore, and S. W. Keckler. Scalable hardware memory disambiguation for high ilp processors. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2003.

[15] P. Shivakumar and N. Jouppi. Cacti 3.0: An integrated cache timing, power, and area model. Technical report, COMPAQ Western Research Lab, 2001.

[16] C. Zilles and G. Sohi. Master/slave speculative parallelization. In *Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture*, Nov. 2002.