Decomposing the load–store queue by function for power reduction and scalability

L. Baugh
C. Zilles

Because they are based on large, content-addressable memories, load–store queues (LSQs) present implementation challenges in superscalar processors. In this paper, we propose an alternate LSQ organization that separates the time-critical forwarding functionality from the process of checking that loads received their correct values. Two main techniques are exploited: First, the store-forwarding logic is accessed only by those loads and stores that are likely to be involved in forwarding, and second, the checking structure is banked by address. The result of these techniques is that the LSQ can be implemented by a collection of small, low-bandwidth structures yielding an estimated three to five times reduction in LSQ dynamic power.

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: buffering store addresses and values for in-order retirement, forwarding in-flight store values to loads, detection of load and store ordering violations, and detection of memory 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 more in-flight instructions) has a significant impact on 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 (querying it for conflicts with the current memory instruction) must be performed in the same amount of time as the L1 data cache access, which is done in parallel in order to avoid further complication of the instruction scheduler.

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

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

For store forwarding, we propose using a structure—the *store-forwarding buffer* (SFB)—that 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 require forwarding. Similarly, required bandwidth is reduced by snooping only for those loads that are predicted to require forwarding. Because these predictions can be wrong, a mechanism is required to detect faulty predictions, known as *misspeculations*.

A second structure, the *memory validation queue* (MVQ), detects load–store ordering violations, consistency violations, and forwarding mispredictions. This structure must observe all in-flight loads and stores to identify violations. To efficiently implement this structure, we bank it by address. Such banking provides

0018-8646/06/$5.00 © 2006 IBM

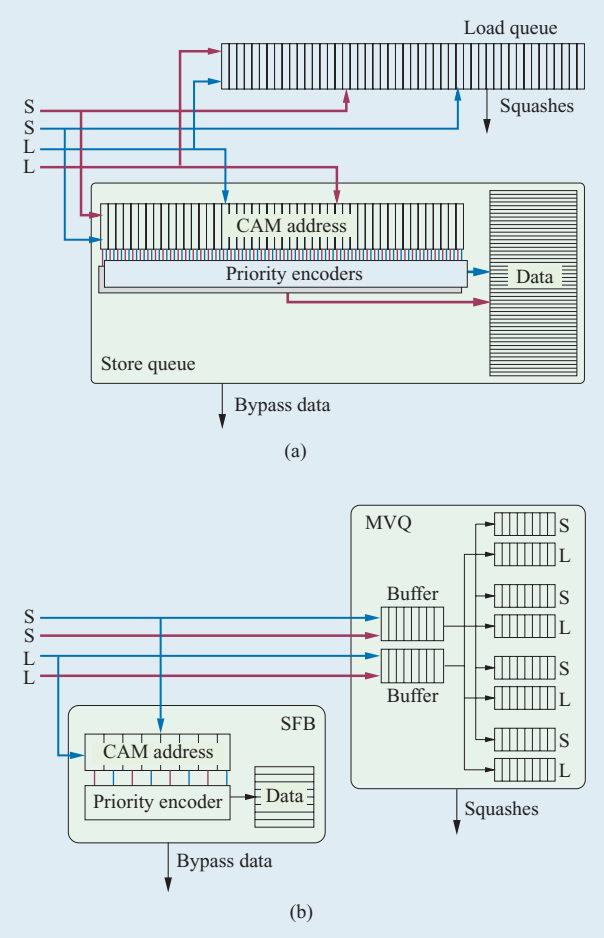

Figure 1

(a) Traditional monolithic load and store queue design. A datapath that can support up to two loads and/or two stores per cycle is shown. (b) Proposed decoupled load and store queue design. A datapath that can support up to two loads and/or two stores per cycle is shown, but only one load and one store can access the SFB. The blue and red lines indicate different memory functional units. (S = store, L = load.)

scalability and reduced energy consumption at the cost of a potential imbalance between banks. To tolerate bank conflicts and enable wider issue of memory instructions, we decouple processing in the MVQ from instruction execution by adding a small wait queue. Validation is tolerant of queuing delay because it merely has to take place before the associated instructions commit.

The contributions of this work are threefold:

- We describe a load–store queue design that decouples store forwarding from other LSQ functions, decomposing the LSQ into a small, low-bandwidth (and hence fast) SFB and a latency-tolerant MVQ, which can be made efficient and capable of high throughput by banking.
- We demonstrate that address-based hashing can be used to partition a processor address stream into four roughly balanced streams, making banking effective. Furthermore, we show that bank imbalance, when it does occur, is caused by repeated loading or storing to a single address.
- We provide the mechanisms required to achieve good utilization of banked LSQ structures while minimizing squashing (restarts due to misspeculations) and stalling. Specifically, we discuss how execution throttling can minimize resource oversubscription and how to deal with potential deadlocks.

Related work

The work most closely related to ours is that of Roth, who independently made the observation that not all loads and stores have to be considered for forwarding [1]. 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 [2], which eliminates the necessity of a load queue at the expense of re-executing a fraction of loads at retirement. This fraction can be further reduced by tracking store window vulnerabilities [3].

Other proposed approaches to filtering include the work of Park et al. [4], which extends a store set predictor [5] to predict instructions involved in forwarding, and the proposal by Sethumadhavan et al. [6] to use a Bloom filter to filter store queue snoops that can be guaranteed not to match. Park's work achieves an equivalent reduction in snoops, but with a more complex predictor. The Bloom filter approach has two drawbacks relative to our proposal. The first drawback is that accessing the Bloom filter is on the critical path (i.e., it must be done between generating an address and accessing the store queue). The second drawback is that the scheduler does not know whether an instruction will have to snoop, forcing it either to be conservative or risk overloading the store queue ports.

Banking by address was previously considered by Sethumadhavan et al. [6], but they discarded the idea because they failed to achieve good results. There are three key differences between their proposal and ours: We propose banking only the latency-tolerant verification portion of the LSQ, which can tolerate a buffer to smooth out bank conflicts; our throttling mechanism can be viewed as a hybrid of their stalling and squashing mechanism, which minimizes the number of squashes required without being overly conservative; and our primary site of throttling is in the scheduler, rather than at decode.

Hierarchy has also been proposed as a solution to scaling the LSQ. Akkary et al. [7] propose caching recent instructions in a first-level store queue, with other instructions residing in a second-level structure. This approach reduces the size (though not the bandwidth) of the latency-critical store queue, but reduces latency predictability. A hybrid of banking and hierarchy is considered by Torres et al. [8], whose design features a banked first-level store queue that speculatively forwards values, backed by a larger, latency-tolerant second-level store queue that detects and squashes misspeculations.

LSQ organization

In this section, we describe our proposed LSQ organization. Because we use a store queue similar to a traditional LSQ as a building block of our design, we begin by describing its salient details, and then describe the two components of our proposed LSQ design.

Age-ordered load and store queues

The most common implementation of an LSQ, as shown in **Figure 1(a)**, involves a pair of buffers (one for loads and one for stores) that hold instructions in program order (i.e., *age-ordered*). 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 later (in program order) load, a pipeline 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 facilitated by age ordering) and used to drive a random access memory (RAM) array that holds the store value.

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% of the maximum number of in-flight instructions (Alpha 21264: 32 loads/32 stores, 80 in-flight instructions [9]; Intel Pentium** 4: 48 loads/32 stores, 128 in-flight instructions [10]). To address the scaling challenges of monolithic LSQ, we present our decoupled LSQ [**Figure 1(b)**].

Store-forwarding buffer

As described above, we streamline the performance-critical store queue by using it only for those instructions that require it. In **Figure 2**, the yellow and blue bars respectively show the fraction of dynamic loads and stores that *matched*[1] in the LSQ and hence required forwarding, for a machine with a 256-entry instruction window. On average, only 7% of dynamic loads and 20% of dynamic stores are involved in forwarding in our runs.

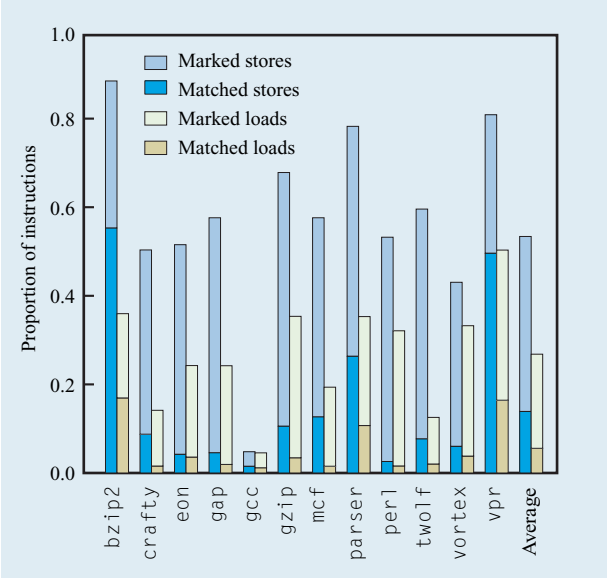

Figure 2

Proportion of dynamic instructions marked and matched for SPEC2000 integer benchmarks.

Because whether or not an instruction is forwarded is a property of the program, its program counter can be used to segregate those instructions 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 the forwarding behavior of an instruction; all bits are initially cleared and an instruction bit is set when it is first detected to require forwarding. This simple predictor is very effective for loads (filtering out 70% of dynamic loads) and moderately effective for stores (filtering out 40% of dynamic stores), as shown in Figure 2. Since there are generally more loads than stores, it is desirable that more loads be filtered than stores.

Ideally, this prediction bit is stored in the instruction—an option when defining a new instruction-set architecture (ISA) or dynamically translating to an internal ISA [11–15]—because then the behavior has to be learned only once. Alternatively, this prediction can be implemented by associating an extra bit with each instruction in the instruction cache (I-cache). To handle

[1] Engaged in either forwarding or an ordering violation.

programs with large working sets (not a problem for the SPEC2000** integer benchmarks), it may be beneficial to "page" (store when evicted from the cache) these predictions into L2 error correction control (ECC) bits, as is done in the AMD Opteron** with branch predictor information [16]. 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*—have to allocate an entry. Since only a fraction of loads and stores are marked, less SFB bandwidth can be provided than overall memory bandwidth with only a modest performance loss (as shown in the methodology and results section). Thus, only a subset of load and store units have to be provided with ports to the SFB. Marked instructions must be slotted and scheduled to execute only on those functional units.

Memory validation queue

Of the four LSQ functions numbered above, the SFB provides only the second: forwarding in-flight store values to loads. Additional structures are required to provide the remaining functions. The first function (buffering store values for in-order retirement) is relatively straightforward. Two reasonable implementations are possible: a separate (non-associative) RAM structure to hold addresses and values, or using such a structure for unmarked stores in conjunction with the SFB. To handle the last two functions (detecting load and store ordering and consistency violations), we provide a structure called the Memory Validation Queue, or MVQ.

The MVQ has two roles: to mark instructions for subsequent introduction into the SFB and to ensure that loads receive their correct value by forcing pipeline squashes when necessary. In addition to the detection of load–store ordering and consistency violations required of traditional load queues, the MVQ must detect situations in which load–store forwarding should have been performed on unmarked loads or stores.

While the MVQ acts much like a traditional LSQ, 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, though others (e.g., a lower-frequency clock domain, high-V_t transistors) are possible. 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 the next section.

Figure 1(b) shows the high-level organization of the MVQ. The MVQ comprises a set of banks, each consisting of a pair of circular queues, one to hold loads and one to hold stores. The entries in these queues contain the same fields as in the traditional load–store queue—CAM accesses to the data address, valid bits (a byte mask for supporting multiple access sizes), and instruction serial number (INUM) (see the handling loads section below). Memory instructions are assigned to banks on the basis of a hash of their memory addresses, ensuring that communicating instructions will be assigned to the same bank.

In the remainder of this section, we first describe how banking the MVQ affects its structure. We then discuss how stores and loads are handled, explain how entries in the MVQ are deallocated, and conclude with a discussion of how deadlock is handled.

Challenges due to banking

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*—i.e., accesses progressing at regular address intervals, such as array iteration—can be minimized. **Figure 3** shows that a relatively even distribution can be achieved in most cases (data shown for four banks, interleaving at the granularity of a 64-bit word and hashing bits 3 to 18 of the address). In general, the address distribution is remarkably constant over time. In the few cases in which the distribution is skewed (e.g., the first sample from `bzip2`), we can attribute it to the existence of a small number of "hot" addresses (see the section on the source of bank imbalance); thus, skewing cannot be avoided by the selection of a different hash function.

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

We address the first challenge by not using an age-ordered queue; instead, we assign entries first-in first-out (FIFO) in execution order, maintaining head and tail pointers. Age ordering primarily serves two purposes: simplification of the management of queue resources and simplification of priority encoding. Because of the simple FIFO allocation scheme we use, we cannot deallocate entries as soon as they retire when reordering has occurred; but, because the degree of reordering is

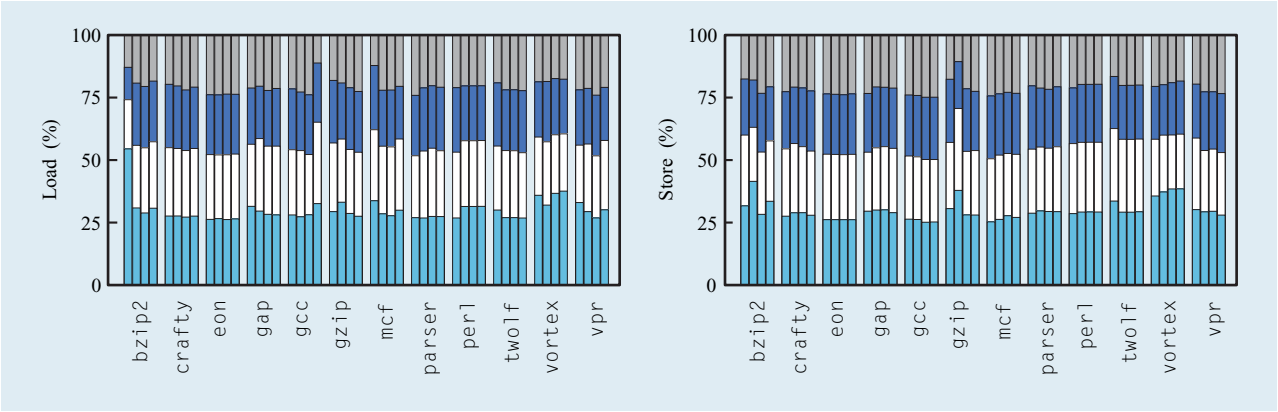

Figure 3

Address-based hashing used to partition dynamic memory instructions into roughly equal groups. For each benchmark, each column represents occupancy of four different hash bins over a 100-million-instruction interval—from left to right, starting at 0 billion, 3 billion, 5 billion, and 8 billion instructions into the execution. Within each column, fractions of occupancy are sorted from largest to smallest, bottom to top.

generally modest, this has little practical impact. The execution time allocation does improve utilization, however, because it avoids tying up resources before they are needed. Solving the priority-encoding issue is more involved, but it can be managed (using the INUMs stored in the MVQ) because of the MVQ latency tolerance and the fact that accesses to the same address are rarely reordered (see the following sections).

The second challenge, that of bank conflicts, is easily addressed by adding a buffer [Figure 1(b)] to smooth out instantaneous bank imbalance. The addition of this buffer increases the latency of an MVQ insertion, but since the MVQ is used only to signal pipeline squashes and to mark instructions involved in forwarding, it is latency-insensitive, and its latency need not be predictable.

The third challenge is the most difficult, because there is a tension between fully utilizing the MVQ and avoiding oversubscribing any one bank. Our primary mechanism is to issue memory instructions only when there is space available in the buffer. This is done by tracking, at the scheduler, the number of buffer entries that have been allocated but not freed. The MVQ buffer is sized to account for the instruction in flight between schedule and address-generation pipeline stages.

Handling stores

As a store is written to its MVQ bank store queue, the entry index (read from the head pointer) is sent to the re-order buffer (ROB) for use at retirement time. In parallel, the load queue of the bank is searched (using the CAM port) for entries with matching store address and valid bits and later INUM to detect ordering violations. If a load is found with a matching address, overlapping valid bits, and a younger INUM, the MVQ pipeline is halted. Such a match does not guarantee an ordering violation (a load may have received a value from a younger store that executed earlier than the present store), but we have found that the complexity of detecting such circumstances cannot be justified because they are relatively infrequent.

When an ordering violation is detected, the offending load and later instructions are squashed, mark bits are set for the load and store instructions, and the memory dependence predictor is trained. For these last two operations, program counters are retrieved from the ROB using the load and store INUMs available from the MVQ.

When multiple matches occur, we need to squash back to the oldest; also, we choose to add only the oldest to our memory dependence predictor so as to minimally synchronize the execution. Because instructions are not necessarily stored in program order, INUMs must be compared to identify the oldest. Our solution to this problem is a low-cost, low-performance one, because ordering violation squashes (particularly those involving multiple matches) are rare. When processing a store, the MVQ load queue sets match bits on all matching entries. The INUMs of the matching lines are then read out one per cycle (while the MVQ is otherwise stalled), retaining the oldest INUM. This approach affects performance by less than 0.001% in all cases observed.

Handling loads

Loads are similarly entered into the MVQ bank load queue, with the position being forwarded to the ROB. In parallel, the load snoops the MVQ bank store queue for matching (same address, overlapping valid bits, earlier INUM) stores where forwarding should have occurred. If the mark bits for either the load or the store are not set, a

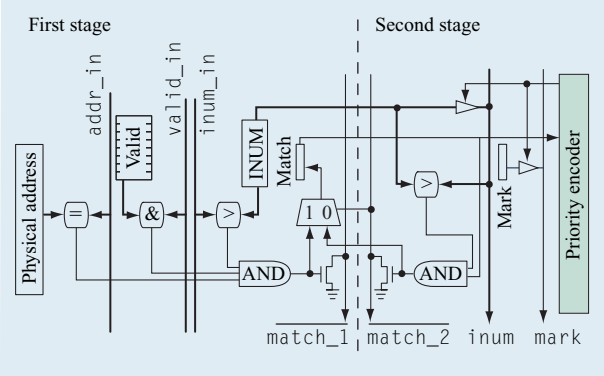

Figure 4

Closeup of an entry of the MVQ store queue.

value misspeculation has likely occurred; the pipeline is squashed and the mark bits are set on both instructions (again using the stored INUMs to retrieve program counters from the ROB).

When multiple matches occur, the MVQ must identify the youngest (the true producer) in order to avoid conservatively marking all matching stores. In contrast to an LSQ, the instructions in the MVQ cannot be relied on to be in program order. Nevertheless, stores to the same address are very rarely reordered in practice (an observation also made by Park et al. [4]), so a priority encoder almost always (more than 99.9% of the time) returns the correct value. Thus, our implementation assumes that the store closest to the head is the youngest, then validates this assumption.

To prevent this process from affecting the throughput of the MVQ, we pipeline the store queue access over two cycles. In the first cycle, we identify all matches, setting the match bits shown in **Figure 4**. In the second cycle, if any matches have occurred (when `match_1` is pulled down), we prioritize the matches, select the presumed youngest (using a priority encoder) and (attempt to) verify that all other matches are older. The verification is performed by broadcasting the INUM of the entry selected by the priority encoder on a second INUM CAM port to see whether any of the matched entries are younger. If no entries are younger, the `match_2` signal will be low, and the presumed youngest mark bit, which is read out while its INUM is being broadcast, is checked. If there is at least one older entry, we must iterate; the match bits are updated so that only those matching entries younger than the presumed youngest are set, and the process is repeated. If another load was in the first stage of the pipeline and had a match, it would have to be replayed on the following cycle, but, as previously noted, this almost never happens.

Deallocating entries

Loads and stores are not allowed to retire until they have been processed by the MVQ. Once an instruction has been committed, its MVQ entry can be deallocated in the background. When the tail instruction in an MVQ load queue has an INUM older than the oldest retired instruction, the instruction is invalidated and the tail pointer is incremented. Because instructions are allocated in the MVQ in execution order, they must also be deallocated in execution order. Because instruction reordering in practice is modest, this yields only a small inefficiency. Similarly, squashed instructions are invalidated, but the "holes" created in the queues are not collapsed.

A similar process happens with the store queue, but stores cannot be deallocated immediately at retirement. Before a store entry can be deallocated, it must be snooped by all loads that executed before its retirement (i.e., before it became available from the cache), and some of these instructions may still be in the MVQ buffer waiting for entry into an MVQ queue. By recording, at the retirement of a store, the number of buffered loads destined for the same bank and decrementing this count each time a load is processed, the safe time for deallocating a store can be determined. If the MVQ buffer is limited to hold six loads, we need to keep track of, at most, seven INUMs per bank—an INUM that is currently safe to retire and INUMs that are safe to retire after one to six loads are processed. This functionality can be implemented with a circular buffer without any CAM logic.

Deadlock avoidance, detection, and resolution

As with any situation in which the resources are limited and are allocated out of program order, the MVQ has the potential for deadlock. Deadlock in the MVQ can occur in two ways. Both cases begin with an instruction being scheduled later than its program order and becoming next-to-retire while some set of MVQ banks is full:

- If the late instruction issues and enters the MVQ buffer but cannot enter its bank because the bank is full, the late instruction can never retire; however, until it does so, no other instructions can retire.
- If the late instruction becomes the next to retire when some set of banks in the MVQ is full and the MVQ buffer is blocked on that set of banks, it can never issue, since there is no room for it in the MVQ, and the MVQ will remain full, since none of the instructions in it can retire until the late instruction does so.

The latter case requires a full MVQ bank, an MVQ buffer full of instructions waiting to enter the full bank,

and a memory instruction, scheduled out-of-order, bypassed by every instruction in the MVQ. In practice, for reasonable MVQ sizes, this case is exceedingly rare; we have never observed it. To detect it, we cause a timeout to occur if the next-to-retire instruction remains unissued for long. If it has not yet issued and a timeout has elapsed, we determine that a deadlock has occurred. Resolution is easy, if costly: We flush the pipeline back to the blocked instruction and resume execution.

The former case happens more frequently, but fortunately it can be avoided. When the MVQ detects that an instruction in the buffer is the next to retire, it can allow that instruction to snoop and remove itself from the MVQ without ever allocating the instruction entry in a bank. To do this, the MVQ permits the instruction to snoop its bank as usual, but also requires the instruction to snoop backward in the buffer, examining all later-issued instructions for matches. The extra time required for such an operation is small compared with the pipe flush otherwise required. This case is not common, but it does occur, particularly in memory-intensive benchmarks such as `mcf`.

To reduce the likelihood of either situation, 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 may fill before a stalled instruction can execute.

Experimental method and results

We evaluated our proposed load–store queue design using timing simulations of the SPEC2000 integer benchmarks. Our timing simulator uses the loader and system call functionality from SimpleScalar [17], but the pipeline model has been rewritten to perform a true execution-driven simulation of Alpha binaries. Parameters for our simulated machine are as follows:

- *Scheduler and pipeline:* Four-issue, twelve-stage pipeline, 256-entry instruction window, 4k gshare predictor with 8 bits of history.
- *Memory:* 64-KB two-way associative L1 instruction and data caches with one-cycle latency, 1-MB eight-way associative L2 cache with 20-cycle latency, 80-cycle memory latency.
- *Functional units (latency in clock cycles):* Four integer arithmetic logic units (1), one integer multiply/divide (3/12), two memory ports (three loads/two stores), two floating-point arithmetic logic units (2), one floating-point multiply/divide (4/12).

Benchmarks are compiled with the Alpha compiler at the highest level of optimization, but without profile information. The results presented in this paper are for 200 million instruction runs started after skipping the first five billion instructions.

In this section, we demonstrate that filtering based on previous forwarding behavior significantly reduces the required number of entries and ports on the SFB compared with a traditional LSQ. We then show that our throttling mechanisms are sufficiently effective to enable the performance of an MVQ with four banks and 16 entries per bank to approximate that of an ideal MVQ. We conclude this section by demonstrating that an SFB MVQ design enables performance equivalent to that of a conventional LSQ with a three to five times reduction in dynamic power.

Varying store queue size and ports

In a system using a conventional LSQ, performance is closely related to the LSQ parameters. In particular, reducing the queue capacity or the number of ports severely limits performance. In **Figure 5(a)**, the LSQ capacity is varied between eight and 64 entries with both single-ported and double-read/write-ported queues. With the conventional LSQ (solid curves), single-ported performance trails double-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 eight elements.

In contrast, we observe systems to be much less sensitive to the size and bandwidth of an SFB. Figure 5(a) shows that 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 eight-entry queue. These SFB sensitivity results were generated with an ideal (i.e., unlimited bandwidth and capacity) MVQ. We consider practical MVQ configurations next.

Varying MVQ parameters

We explore two MVQ parameters that potentially affect performance: the number of banks in the MVQ and the capacity of both the store queue and the load queue in each bank. Our results, shown in **Figure 5(b)**, 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. This significantly reduces power, as we show below.

MVQ power

Our previous results show that our SFB MVQ combination can achieve performance comparable to that of 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 dynamic power reduction. According to CACTI 3.2 [18],

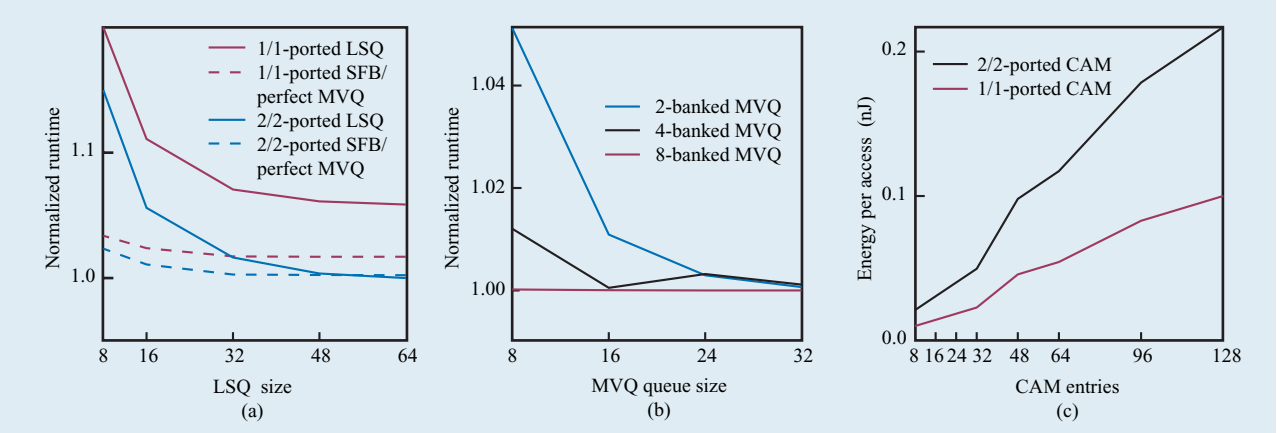

Figure 5

(a) Data averaged across samples of the SPEC2000 integer benchmarks, normalized to a 64-entry 2/2-ported LSQ. A conventional system is sensitive to its store queue parameters; an MVQ-equipped system is relatively insensitive. (b) Data averaged across the SPEC2000 integer benchmarks, normalized to the case with a perfect MVQ. (c) Query power in CAMs varies with the number of ports and the number of elements. All simulations were of 0.09-μm technology. (Data produced by CACTI 3.2.)

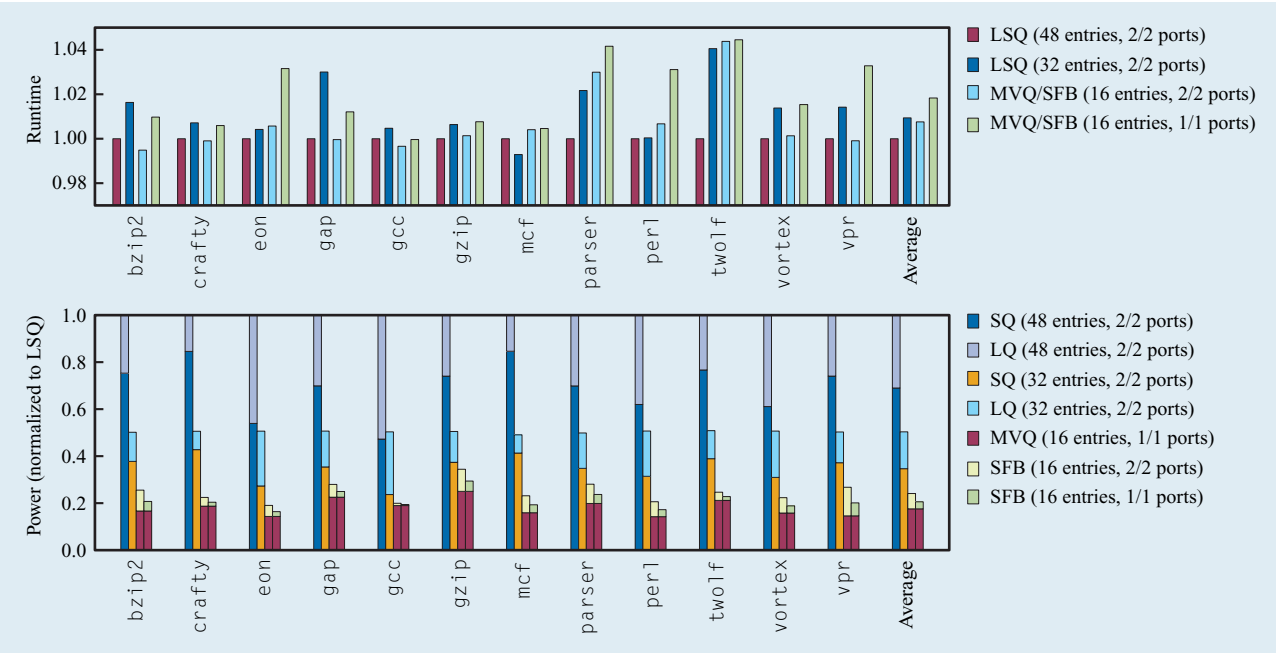

Figure 6

Data shown for 48- and 32-entry LSQs with two load and two store ports for each queue. The MVQs have a 16-entry buffer and four banks, with each bank having 16 entries and one read and one write port.

energy per access for a CAM scales roughly linearly with both the number of entries and the number of ports [**Figure 5(c)**]. Thus, querying a 48-entry 2-read/2-write-ported CAM takes almost seven times the power of a query to a 16-entry 1-read/1-write-ported CAM.

As a result, using smaller structures and limiting the queries to them translates into as much as a five times 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 4×24-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 4×16-entry MVQ. **Figure 6** shows that the SFB/MVQ achieves roughly a five times and three times reduction of dynamic power, respectively.

A quick calculation suggests that our design compares equivalently or favorably with conventional LSQ designs in terms of static power and area. To analyze both the static power consumption and area requirements of our design, we estimated the transistor count of both the traditional design and the SFB/MVQ. 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. Since the MVQ dominates the transistor count of our architecture, we expect this to have a favorable influence on static power and area. Furthermore, since the SFB and MVQ scale more slowly with instruction window size than do traditional LSQs [Figure 5(a)], we expect this trend to continue.

Source of bank imbalance

In this section, we demonstrate that when bank imbalance occurs in the MVQ, it is due to the repetition of a single or small set of addresses. We observe this correlation in a microarchitecture-independent way by breaking the execution of a program into intervals of 1,024 instructions; for each interval we record two statistics. First, we count the number of times there was a load from each address and record the count of the most frequent. Second, we hash all of the load addresses and record the amount of imbalance (i.e., we subtract the average bank occupancy from the maximum bank occupancy). In this way, each interval provides us with a point in two dimensions. If we aggregate these points, we can produce a three-dimensional plot such as that shown in **Figure 7**. It can be seen that there is a strong linear correlation between a value being repeated within the interval (plotted on the x-axis) and the interval bank imbalance (plotted on the y-axis).

To understand the reason for this, we analyzed the assembly and source code of a few regions with frequently repeating addresses. The cases we observed all resembled this code from `bzip2`:

```
for (j = 0; j < limit; j+=1024) {
    spec_fd[i].buf[j] = 0;
}
```

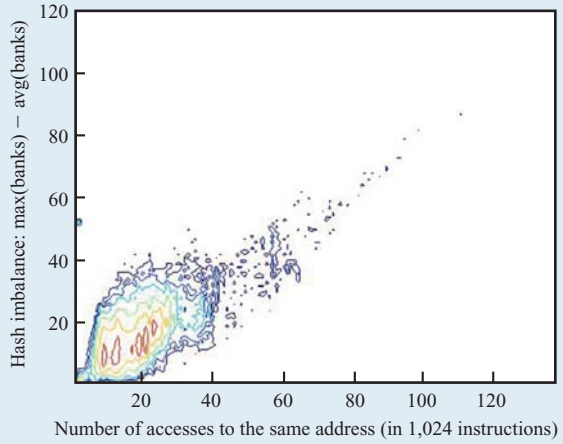

Figure 7

Correlation between repeated accesses to the same address and load imbalance within a 1,024-instruction interval. Data shown is for loads from the benchmark `crafty` and is representative of loads and stores across all of SPECint2000**.

This code fragment initializes values in a large array that is reached through a level of indirection. We imagine that the Alpha compiler fails to promote `spec_fd[i]` to a register because of a perceived potential alias with `spec_fd[i].buf[j]`. As a result, this load, which always loads from the same address, represents 100% of the loads during its interval.

Conclusion

Scaling traditional LSQ designs presents a pressing problem for architects because the content-addressable memories on which such designs are based scale poorly with regard 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 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 we propose throttling techniques for dealing with load imbalance between the banks and a deadlock-avoidance mechanism to deal with deadlocks caused by the limited MVQ resources. 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 combination of the SFB and MVQ a practical alternative for future processors.

Acknowledgments

This research was supported in part by NSF Grant No. CCR-0311340, NSF CAREER Award No. CCR-03047260, and a gift from the Intel Corporation. We thank the members of the MSSP group and the anonymous reviewers for feedback on previous drafts of this paper.

References

1. A. Roth, "A High-Bandwidth Load-Store Unit for Single- and Multi-Threaded Processors," *Technical Report MS-CIS-04-09*, University of Pennsylvania, Philadelphia, PA 19104, 2004.
2. H. W. Cain and M. H. Lipasti, "Memory Ordering: A Value-Based Approach," *Proceedings of the 31st Annual International Symposium on Computer Architecture,* 2004, pp. 90–101.
3. A. Roth, "Store Vulnerability Window (SVW): Re-Execution Filtering for Enhanced Load Optimization," *Proceedings of the 32nd Annual International Symposium on Computer Architecture,* 2005, pp. 458–468.
4. Il Park, C.-L. Ooi, and T. N. Vijaykumar, "Reducing Design Complexity of the Load-Store Queue," *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, 2003, pp. 411–422.
5. G. Z. Chrysos and J. S. Emer, "Memory Dependence Prediction Using Store Sets," *Proceedings of the 25th Annual International Symposium on Computer Architecture,* 1998, pp. 142–153.
6. S. Sethumadhavan, R. Desikan, D. Burger, C. R. Moore, and S. W. Keckler, "Scalable Hardware Memory Disambiguation for High-ILP Processors," *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture,* 2003, pp. 399–410.
7. H. Akkary, R. Rajwar, and S. T. Srinivasan, "Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors," *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture,* 2003, pp. 423–434.
8. E. F. Torres, P. Ibanez, V. Vinals, and J. M. Llaberia, "Store Buffer Design in First-Level Multibanked Data Caches," *Proceedings of the 32nd Annual International Symposium on Computer Architecture,* 2005, pp. 469–480.
9. R. E. Kessler, "The Alpha 21264 Microprocessor," *IEEE Micro* **19**, No. 2, 24–36 (1999).
10. M. F. Chowdhury and D. M. Carmean, "Method, Apparatus, and System for Maintaining Processor Ordering by Checking Load Addresses of Unretired Load Instructions Against Snooping Store Addresses," U.S. Patent Application 6484254, November 2002.
11. V. Adve, C. Lattner, M. Brukman, A. Shukla, and B. Gaeke, "LLVA: A Low-Level Virtual Instruction Set Architecture," *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture,* 2003, pp. 205–216.
12. K. Ebcioglu and E. R. Altman, "DAISY: Dynamic Compilation for 100% Architectural Compatibility," *Proceedings of the 24th Annual International Symposium on Computer Architecture,* 1997, pp. 26–37.
13. 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," *Proceedings of the 34th Annual IEEE/ACM International Symposium on Microarchitecture,* 2001, pp. 16–27.
14. A. Klaiber, "The Technology Behind Crusoe Processors," white paper, Transmeta Corporation, 3990 Freedom Circle, Santa Clara, CA 95054, January 2000.
15. C. Zilles and G. Sohi, "Master/Slave Speculative Parallelization," *Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture,* 2002, pp. 85–96.
16. C. N. Keltcher, K. J. McGrath, A. Ahmed, and P. Conway, "The AMD Opteron Processor for Multiprocessor Servers," *IEEE Micro* **23**, No. 2, 66–76 (2003).
17. T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An Infrastructure for Computer System Modeling," *IEEE Computer* **35**, No. 2, 59–67 (2002).
18. P. Shivakumar and N. Jouppi, "CACTI 3.0: An Integrated Cache Timing, Power, and Area Model," *Technical Report 2001/2*, COMPAQ Western Research Laboratory, Palo Alto, CA 94301, 2001.

Received June 21, 2005; accepted for publication July 22, 2005; Internet publication March 14, 2006

Lee Baugh *University of Illinois at Urbana–Champaign, Department of Computer Science, 201 N. Goodwin Avenue, Urbana, Illinois 61801 (leebaugh@uiuc.edu)*. Mr. Baugh is a doctoral candidate in computer science at the University of Illinois at Urbana–Champaign, under the guidance of Dr. Craig Zilles. He holds B.S. and M.S. degrees from the University of Texas at Dallas. Mr. Baugh's research interests include speculative multithreading processors, dynamic optimization, and microarchitecture.

Craig Zilles *University of Illinois at Urbana–Champaign, Department of Computer Science, 201 N. Goodwin Avenue, Urbana, Illinois 61801 (zilles@cs.uiuc.edu)*. Dr. Zilles is an assistant professor of computer science at the University of Illinois at Urbana–Champaign. He holds B.S. and M.S. degrees in mechanical engineering from the Massachusetts Institute of Technology, and an M.S. degree in electrical and computer engineering and a Ph.D. degree in computer science, both from the University of Wisconsin at Madison. Dr. Zilles's current research focuses on cooperation between compilers, runtime systems, architecture, and microarchitecture, often motivated by work on program characterization.