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

Lee Baugh and Craig Zilles
Department of Computer Science
University of Illinois at Urbana-Champaign
Email: {leebaugh, zilles}@uiuc.edu

***Abstract*— Because they are based on large content-addressable memories, load-store queues (LSQs) 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 time-critical 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 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. Furthermore, we explore applying these techniques to processors requiring only statistical correctness guarantees where another factor of two power reduction is achievable.

I. 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 more 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 – that is, 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, 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:

1) **Store value forwarding 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 forwarding.**

For store forwarding, 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 require forwarding. Likewise, required bandwidth is reduced by only snooping 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, 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 and enable wider issue of memory instructions, 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.

The contributions of this work are three-fold:

1) 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.

2) 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.
3) We provide mechanisms required to achieve good utilization of banked LSQ structures while minimizing squashing (restarts due to misspeculations) and stalling. Specifically, to avoid over-subscribing one bank, we provide an execution throttling mechanism that minimizes squashes based on the availability of wait queue entries. Furthermore, we describe how deadlocks (due to out-of-program-order resource allocation) can generally be avoided and detected in the rare circumstances they occur.

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 [16]. 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 [5], 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 [18]. 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 over conservative, 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 [7] to predict which loads are likely to receive values forwarded by stores [15]. 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 [18]. 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.

More recent work by Roth further limits the proportion of loads which must re-execute by establishing a window of stores to which the load is vulnerable and employing a Bloom filter on the data address to determine whether a given load must be re-executed [17]. However, full re-execution of some loads is still required.

Complex CAM (Content-Addressible Memory) structures are not only reduced, but entirely avoided in the load-store mechanism proposed by Gandhi et al [10]. However, their technique is only applicable in the context of a "latency-tolerant processor" with an available instruction window in the thousands of instructions.

A banked, hierarchical store queue design suitable for wide-issue cores is proposed by Torres et al [19]. In this design, monolithic store queue complexity is reduced by decomposing the structure into several smaller ones, which speculatively forward values. A larger, latency-tolerant second-level store queue detects and squashes misspeculations.

This paper is organized as follows. In Section II, we describe the organization of our proposed LSQ design. In Section III, we describe our experimental methodology and results. In Section IV, we consider how to further reduce power consumption in the context of processors with reduced correctness constraints. In Section V, we look closer at the source of bank imbalance and then conclude (in Section VII), after a discussion of related work (in Section VI).

II. 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 first describe its salient details. We then describe the two components of our proposed LSQ design in Sections II-B and II-C, respectively.

A. 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 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 RAM array that holds the store's value.

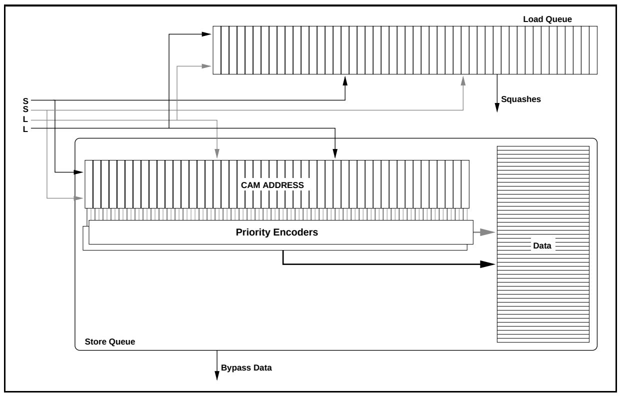

Fig. 1. Traditional Monolithic Load/Store Queue Design. *The store bypass path, which involves a CAM lookup, priority encoder, and RAM access, is generally one of a processor's critical paths. A datapath that can support up to 2 loads and/or 2 stores per cycle is shown.*

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 [12], Pentium 4: 48 loads/32 stores, 128 in-flight instructions [6]).

B. Store Forwarding Buffer

As described in Section I, 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*[1] 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 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 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% of dynamic loads) and moderately effective for stores (filtering out 40% of dynamic stores), as shown in Figure 2. As there are generally more loads than stores, it is desirable that more loads are filtered than stores.

Ideally, this prediction bit is stored in the instruction—an option when defining a new ISA or dynamically translating to an internal ISA [1], [8], [9], [13], [20]—because then

[1]Engaged in either forwarding or an ordering violation

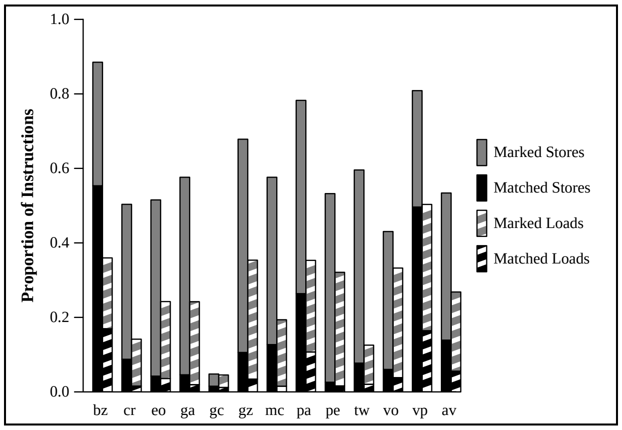

Fig. 2. Only a small, predictable fraction of memory instructions is involved in bypassing. *Fraction of all loads and stores executed, we break them down into those that* ***matched*** *in the LSQ, those that are* ***marked*** *because previous instances of their static instruction matched, and those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

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 [11].

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.

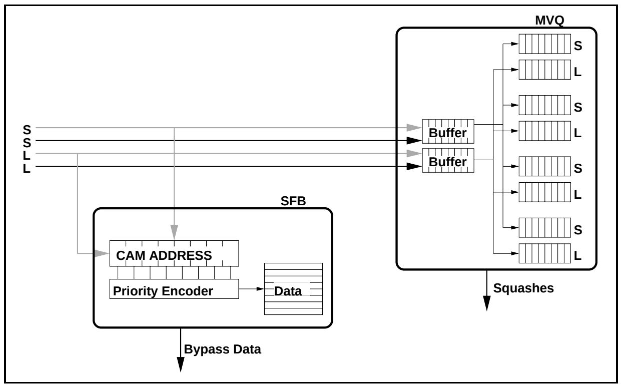

Fig. 3. Proposed Decoupled Load/Store Queue Design. *The performance critical store forwarding buffer (SFB) is used by only those instructions likely to be involved in store forwarding, reducing its access time and energy consumption. The energy consumed by the rest of the LSQ functionality is reduced 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, less SFB bandwidth can be provided than overall memory bandwidth with only modest performance loss (as shown in Section III). 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 functional units.

C. Memory Validation Queue

Of the four LSQ functions described in Section I, the SFB only provides the second (forwarding). Additional structures are required to provide the remaining functions. The first function (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 last two functions (detecting load/store ordering and consistency violations) we provide a Memory Validation Queue (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 where 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 Vt 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 Section III.

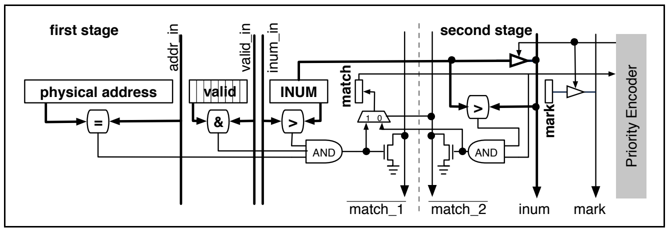

Fig. 5. A closeup of an entry of the MVQ store queue. *On the first cycle the* `match_1` *line is pulled down if any matches are found and the set of matching entries are latched in the match bit. In the second cycle, a priority encoder selects the oldest (in execution order) match, and broadcasts its INUM so that it can be verified as the oldest. It is the oldest if* `match_2` *is not pulled down and a misspeculation can be signaled if its mark bit is not set.*

The schematic diagram in Figure 3 shows the high-level organization of the MVQ. The MVQ comprises a set 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), as shown in Figure 5. Memory instructions are assigned to banks based on a hash of their memory addresses, assuring that communicating instructions will be assigned to the same bank.

In the remainder of this section, we first describe how banking the MVQ impacts its structure (Section II-C.1). We then discuss how stores and loads are handled (Sections II-C.2 and II-C.3, respectively) and then explain how entries in the MVQ are deallocated (Section II-C.4). We conclude (in Section II-C.5) with a discussion of how deadlock is handled

1) 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 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[2], 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 skewing could not 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, 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; instead we assign entries FIFO in execution order, maintaining head and tail pointers. Age ordering serves primarily two purposes: 1) simplification of the management of queue resources, and 2) 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 generally modest this has little practical impact. The execution-time allocation does improve utilization, however, by avoiding tying up resources before they are needed. Solving the priority encoding issue is more involved, but can be handled (using the INUMs stored in the MVQ) because of the latency tolerance of the MVQ and the fact that accesses to the same address are rarely reordered (see Sections II-C.2 and II-C.3).

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

[2]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 ISAs without this restriction (*e.g.*, x86) support for insertion into multiple banks would be required.

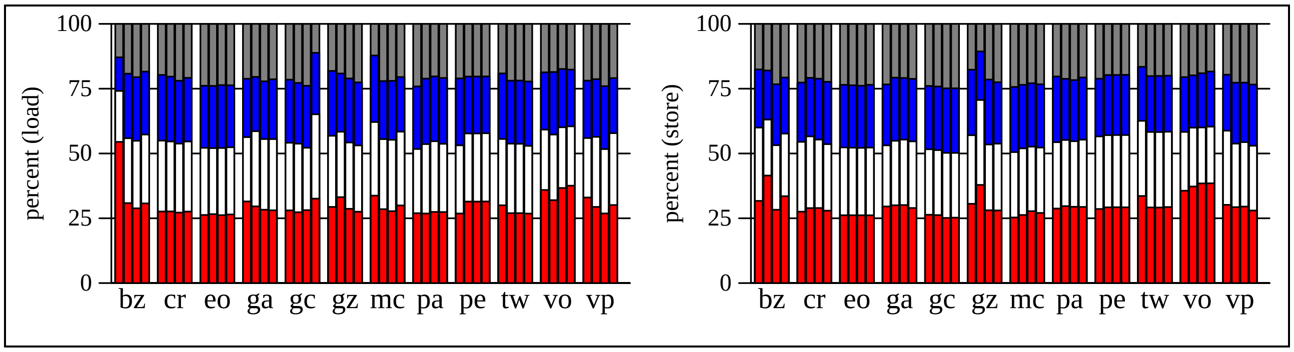

Fig. 4. Address-based hashing can be used to partition dynamic memory instructions into roughly equal groups. *For each benchmark, each column represents occupancy of 4 different hash bins over a 100 million instruction interval — from left to right, starting at 0 billion, 3 billion, 5 billion, and 8 billion instructions into the execution. Within each column, fractions of occupancy are sorted from largest to smallest, bottom to top.*

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 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, as there is a tension between fully utilizing the MVQ and not oversubscribing 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.

2) Handling Stores: : As a store is written to its MVQ bank's store queue, the entry's index (read from the head pointer) is sent to the ROB for use at retirement time. In parallel, the store's address, valid bits, and INUM are CAMed against the bank's load queue to detect ordering violations. If a load is found with a matching address, overlapping valid bits, and a younger INUM, then 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 as they are relatively infrequent.

On detecting an ordering violation, the offending load (and later instructions) is squashed, mark bits are set for the load and store instructions, and the memory dependence predictor is trained. For these last two operations, PCs 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 only add the oldest to our memory dependence predictor so as to minimally synchronize the execution. Because instructions are not necessarily stored in program order, INUMs need to 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) using a carry propagate-style priority encoder that sequences through the matched lines. Each cycle the oldest INUM is retained until all matches have been read out. Performing priority selection in this way impacts performance by less than 0.001 percent in all cases.

3) Handling Loads: : Loads are similarly entered into the MVQ bank's load queue with the position being forwarded to the ROB. In parallel, the load snoops the MVQ bank's store queue for matching (same address, overlapping valid bits, earlier INUM) stores. When such a store is detected then forwarding should have occurred. If the mark bits for either of the load or the store are not set then a value misspeculation has likely occurred and the pipeline is squashed and the mark bits are set on both instructions (again using the stored INUMs to retrieve PCs from the ROB). To avoid repeatedly replaying a load whose unmarked producing store is stalled in the reorder buffer, we squash back to the oldest unmarked instruction.

When multiple matches occur, the MVQ needs to identify the youngest (the true producer) to avoid conservatively marking all matching stores. Unlike an LSQ, a priority encoder is not guaranteed to find the oldest instruction in the MVQ, because instructions can enter out of program order. Nevertheless, stores to the same address are very rarely reordered in practice (an observation also made by Park, et al. [15]), so a priority encoder almost always (over 99.9% of the time) returns the correct value. Thus, our implementation assumes the store closest to the head is the youngest, then validates this.

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 5. In the second cycle, 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 if any of the matched entries are younger. If no entries are younger, the `match2` signal will be low, and the presumed youngest's mark bit, which is read out while its INUM is being broadcast, is checked. If there is at least one older entry, then 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.

4) 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 is lazily deallocated. 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 can be allocated in the MVQ in program order they are also deallocated in program order. Because instruction reordering in practice is modest, this only yields a small inefficiency. Likewise, 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 is available from the cache) and some of these instructions may still be in the MVQ buffer (waiting for entry in an MVQ queue). By recording at a store's retirement 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 known. If the MVQ buffer is limited to hold 6 loads, we need to keep track of at most 7 INUMs per bank: an INUM that is currently safe to retire, and INUMs that are safe to retire after 1-8 loads are processed. This functionality can be implemented with a circular buffer without a CAM port.

5) Deadlock Avoidance, Detection, and Resolution: As with any situation where the resources are limited and allocated out of program order, the MVQ has the potential for deadlock. Deadlock in the MVQ can occur in two ways. Both 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's buffer, but cannot enter its bank because the bank is full, then the late instruction can never retire — but 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's buffer is blocked on that set of banks, then it can never issue, for 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, it is vanishingly rare. To detect it, we apply a timeout on the status of the next instruction to retire — if it has not yet issued, and a timeout has elapsed , then we decide a deadlock has occurred. Resolution is easy, if costly: we flush the pipe back to the blocked instruction and resume execution. This case never occurs in our simulations.

The former case happens more frequently; 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 to remove itself from the MVQ without ever allocating it an entry in a bank. To do this, it 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 to the pipe flush otherwise required. This case is not common, but does occur, particularly in memory-intensive benchmarks like `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 fills before a stalled instruction executes.

III. 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 rewritten to perform a true execution-driven simulation of Alpha binaries. Parameters for our simulated machine can be found in the table in Figure 6. 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.

In this section, we first demonstrate (in Section III-A) 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 III-B) 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 to a conventional LSQ with a 3-to-5-fold reduction in dynamic power.

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)

Fig. 6. Simulation Parameters.

A. 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. In Figure 7, 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.

In contrast, we observe systems to be much less sensitive to the size and bandwidth of a store forwarding buffer (SFB); Figure 7 shows 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. As these results are concerned with sensativity to the SFB, they were generated with an ideal (*i.e.*, unlimited bandwidth and capacity) MVQ. We consider practical MVQ configurations next.

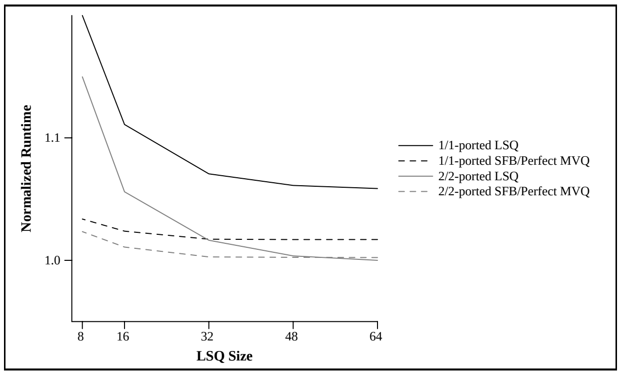

Fig. 7. A conventional system is sensitive to its store queue parameters; an MVQ-equipped system is relatively insensitive. *Data averaged across samples of the SPEC2000 integer benchmarks, normalized to a 64-entry 2/2-ported LSQ.*

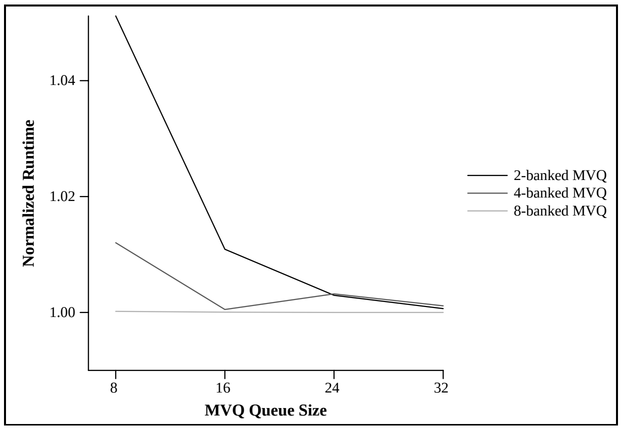

Fig. 8. The MVQ requires 4 banks and 16 entries/bank, but benefits from little more. *Data averaged across the SPEC2000 integer benchmarks, normalized to the case with a perfect MVQ.*

B. 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 and the load queue in each bank. Results in Figure 8 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 16 entries per instruction type (4x16) with minimal performance reduction, which we now show to significantly reduce power.

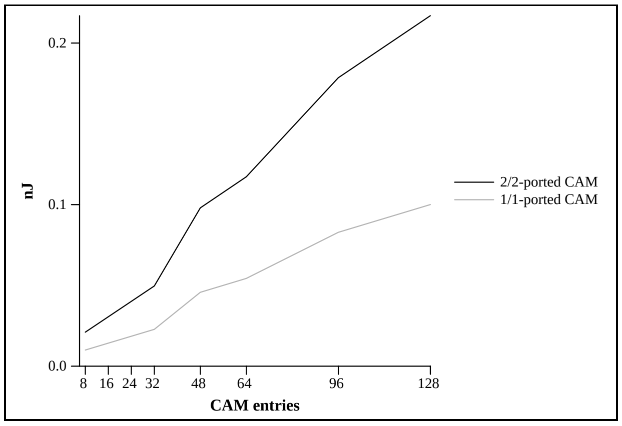

Fig. 9. Query power in CAMs varies with the number of ports and the number of elements. $.09\mu$ *technology. Produced by Cacti 3.2*

C. 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 dynamic power reduction.

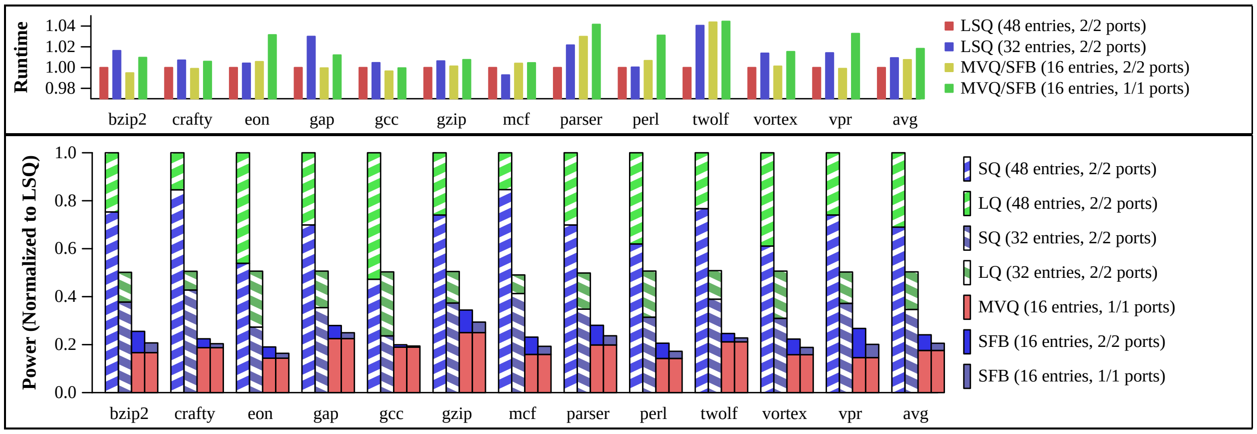

Fig. 10. Designs using the SFB/MVQ can achieve similar performance using smaller, lower bandwidth structures, yielding lower power execution. *Data shown for 48- and 32-entry LSQs (striped) with 2 load and 2 store ports for each queue. The MVQs (solid) have a 16-entry buffer and 4 banks, with each bank 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.*

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 9). 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 and limiting the queries to them 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 10 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[3]. 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. Furthermore, since the SFB and MVQ scale more slowly with instruction window size than traditional LSQs (see Figure 7), we expect this trend to continue.

[3]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.

IV. Applications for Relaxed Correctness

Given that SFB/MVQ power consumption is dominated by the MVQ, it is interesting to consider how MVQ power consumption can be reduced. Because the MVQ is only responsible for checking the work performed by a memory dependence predictor and the SFB, both of which are reliable after initial training, most of the work performed by the MVQ does not affect the program's execution. In fact, in the experiments from the previous section, only around one hundredth of one percent of dynamic instructions are involved in dependence violations detected by the MVQ. In light of the MVQ's high power consumption to utility ratio, we consider applications of the SFB/MVQ in machines that have relaxed correctness guarantees, like Diva [4] and Master/Slave Speculative Parallelization (MSSP) [20].

Master-Slave Speculative Parallelization (MSSP) is a new execution paradigm that decouples the issues of performance and correctness in microprocessor design and implementation. MSSP uses a fast, not necessarily correct, master processor to speculatively split a program into tasks; these tasks are then executed independently and concurrently on slower, but correct, slave processors. This model is most effective when the Master eliminates, in either software or hardware, corner cases – when a great deal of work is omitted at the cost of a small chance of producing an error.

Because results generated by the MSSP master are only used as value predictions, incorrect results can only affect performance and not correctness. Thus, the master only has statistical correctness requirements — necessary to ensure good performance — providing an interesting testbed for exploring the optimization of the MVQ. The key idea is that if we can accurately predict which loads and stores will not be involved in dependence violations, then these instructions need not be inserted into the MVQ, reducing both the size and activity of the MVQ. On the rare occurrence that the MVQ misses a dependence violation, it will be caught by MSSP's verify/commit unit, yielding a more expensive misspeculation, but no incorrect result.

To avoid large prediction tables, we sought instead to discover properties of instructions and their execution contexts that might predict their behavior in the MVQ. We considered four properties for making a prediction: 1) whether a squash has occurred recently, 2) how frequently an instruction has executed (a property that is tracked at a coarse grain by MSSP's iterative dynamic optimizer [14], 3) whether the instruction is marked, and 4) whether the instruction is a load or store. Somewhat disappointingly, we found no combination of these attributes to distinguish instructions (*i.e.*, separate those that should be dropped from those that should not) that was sufficiently accurate across all of the benchmarks. We did, however, find that within a benchmark, the characteristics of instructions that were safe to drop were quite stable.

Thus, the implementation that we studied tries to identify those combinations of attributes (what we call *dropsets* that are indicative of instructions that are safe to drop, and drops a fraction of those instructions. Specifically, we keep 16 counters (one for each dropset) and record the ratio of instructions allocated to that dropset to the number of dependence violations detected. If this ratio is above some threshold (1000 insertions to one violation in our experiment), then the instruction is a candidate for dropping. We found that dropping 60% of drop candidates all the time, and 90% when the MVQ buffers are close to capacity, permitted over 57% of dynamic memory instructions to be dropped from the MVQ, yielding a further factor of two dynamic power reduction with minimal impact on performance, as shown in Figure 11.

V. 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 micro-architecture independent way by breaking a program's execution into intervals of 1024 instructions; for each interval we record two statistics: 1) we count how many times each address was loaded from and record the count of the most frequent, and 2) 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 a point in two dimensions. If we aggregate these points, we can produce a three-dimensional plot, like the contour plot shown in Figure 12. 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's bank imbalance (plotted on the Y axis).

To understand the cause of this, we analyzed the assembly

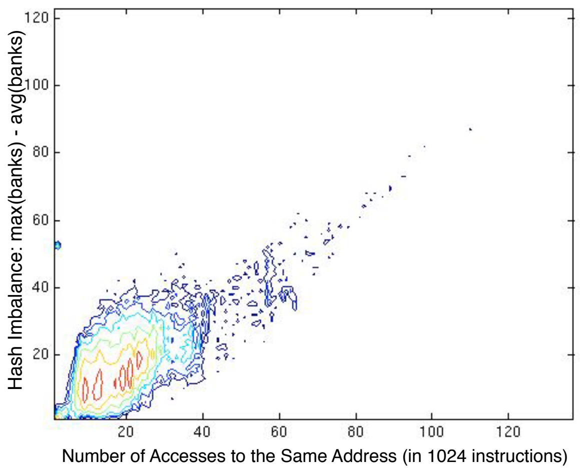

Fig. 12. A contour plot that shows the correlation between repeated accesses to the same address and load imbalance within a 1024 instruction interval. *Data shown is for loads from the benchmark `crafty` and is representative of loads and stores across all of SpecInt 2000.*

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

Fig. 13. A code example that demonstrates a common source of bank imbalance. *The compiler's alias analysis fails to detect that the store to `spec_fd[i].buf[j]` does not alias with the load of `spec_fd[i]`, so the load (to that same address) is performed every iteration of the loop. This example is responsible for the imbalance observed in the first checkpoint of `bzip2` in Figure 4.*

and source code of a few regions with frequently repeating addresses. The cases we observed were all like the code from `bzip2` shown in Figure 13. This code fragment initializes values in a large array that is reached through a level of indirection. The DEC Alpha compiler fails to promote `spec_fd[i]` to a register, due to a perceived potential alias with `spec_fd[i].buf[j]` we imagine. As a result, this load, which always loads from the same address, represents 100% of the loads during its interval.

VI. Related Work

There has been much 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 [16]. 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 [5], 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 [18]. There are three key differences between their proposal and ours: 1) we propose banking only

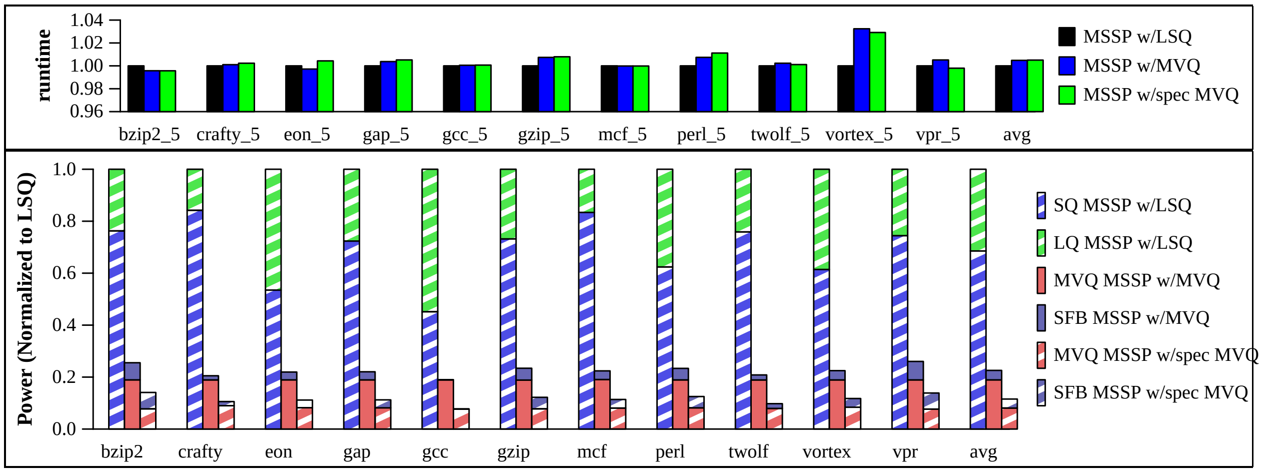

Fig. 11. When instructions are judiciously dropped from the MVQ, the MVQ produces much less dynamic power with almost no performance impact *The traditional LSQ shown here has 48 entries and is dual-ported. The MVQ has 4 ways, 16 entries per way, and is single-ported, and its SFB has 24 entries, single-ported.*

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 over conservative, 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 [7] to predict which loads are likely to receive values forwarded by stores [15]. 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 [18]. 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.

More recent work by Roth further limits the proportion of loads which must re-execute by establishing a window of stores to which the load is vulnerable and employing a Bloom filter on the data address to determine whether a given load must be re-executed [17]. However, full re-execution of some loads is still required.

Complex CAM structures are not only reduced, but entirely avoided in the load-store mechanism proposed by Gandhi et al [10]. However, their technique is only applicable in the context of a "latency-tolerant processor" with an available instruction window in the thousands of instructions.

A banked, hierarchical store queue design suitable for wide-issue cores is proposed by Torres et al [19]. In this design, monolithic store queue complexity is reduced by decomposing the structure into several smaller ones, which speculatively forward values. A larger, latency-tolerant second-level store queue detects and squashes misspeculations.

VII. 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 and a deadlock avoidance mechanism to deal with deadlocks caused by the MVQ's limited 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 SFB and MVQ a practical alternative for future processors.

VIII. Acknowledgments

This research was supported in part by NSF CCR-0311340, NSF CAREER award 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] V. Adve, C. Lattner, M. Brukman, A. Shukla, and B. Gaeke. LLVA: A Low-level Virtual Instruction Set Architecture. In *36th 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] T. M. Austin. Diva: A reliable substrate for deep submicron microarchitecture design. In *Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 196–207, Nov. 1999.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] A. Gandhi, H. Akkary, R. Rajwar, and K. L. S. Sriniva andan. Scalable load and store processing in latency tolerant processors. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 446–457, June 2005.

[11] 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.

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

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

[14] N. Neelakantam. Program orienteering. Master's thesis, Dept. of Electrical and Computer Engineering., University of Illinois at Urbana-Champaign, Urbana, IL, April 2004.

[15] 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.

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

[17] A. Roth. Store vulnerability window (svw): Re-execution filtering for enhanced load optimization. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 458–468, June 2005.

[18] 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.

[19] E. Torres, P. Ibanez, V. Vinals, and J. Llaberia. Store buffer design in first-level multibanked data caches. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 469–480, June 2005.

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