Appears in the 36th Annual International Symposium on Microarchitecture

Scalable Hardware Memory Disambiguation for High ILP Processors

Simha Sethumadhavan Rajagopalan Desikan† Doug Burger
Charles R. Moore Stephen W. Keckler
Computer Architecture and Technology Laboratory
Department of Computer Sciences
†Department of Electrical and Computer Engineering
The University of Texas at Austin
cart@cs.utexas.edu - www.cs.utexas.edu/users/cart

Abstract

This paper describes several methods for improving the scalability of memory disambiguation hardware for future high ILP processors. As the number of in-flight instructions grows with issue width and pipeline depth, the load/store queues (LSQ) threaten to become a bottleneck in both power and latency. By employing lightweight approximate hashing in hardware with structures called Bloom filters many improvements to the LSQ are possible.

We propose two types of filtering schemes using Bloom filters: search filtering*, which uses hashing to reduce both the number of lookups to the LSQ and the number of entries that must be searched, and* state filtering*, in which the number of entries kept in the LSQs is reduced by coupling address predictors and Bloom filters, permitting smaller queues. We evaluate these techniques for LSQs indexed by both instruction age and the instruction's effective address, and for both centralized and physically partitioned LSQs. We show that search filtering avoids up to 98% of the associative LSQ searches, providing significant power savings and keeping LSQ searches to under one high-frequency clock cycle. We also show that with state filtering, the load queue can be eliminated altogether with only minor reductions in performance for small instruction window machines.*

1. Introduction

Computer architects have been improving the performance of processors by implementing deeper pipelines [28], wider issue, and with larger out-of-order issue windows. These trends produce machines in which more than one hundred instructions may be in-flight [16]. Recently, several researchers have proposed techniques for scaling issue windows to sizes of hundreds or thousands of instructions [8, 19, 22]. In all these actual and proposed machines, hardware must perform dynamic memory disambiguation to guarantee that a memory ordering violation does not occur.

For any system capable of out-of-order memory issue, the memory ordering requirements are threefold. First, the hardware must check each issued load to determine if an earlier (program order) in-flight store was issued to the same physical address, and if so, use the value produced by the store. Second, each issued store must check to see if a later (program order) load to the same physical address was previously issued, and if so, take corrective action. Third, the hardware should ensure that loads and stores reach the memory system in the order specified by the memory consistency model. In many processors, the hardware that implements the above requirements is called the load/store queue (LSQ).

One disadvantage with current LSQ implementations is that the detection of memory ordering violations requires frequent searches of considerable state. In a naive LSQ implementation, every in-flight memory instruction is stored in the LSQ. Thus, as the number of instructions in-flight increases, so does the number of entries that must be searched in the LSQ to guarantee correct memory ordering. Both the access latency and the power requirements of LSQ searches scale super-linearly with increases in the amount of state as the LSQ is typically implemented using a CAM structure [2]. As we show in the next section, simply reducing the size of traditional LSQ designs for future machines causes an unacceptable drop in performance, whereas not doing so incurs unacceptable LSQ access latencies and power consumption. These traditional structures thus have the potential to be a significant bottleneck for future systems.

The technique evaluated in this paper to mitigate these LSQ scalability limits is *approximate hardware hashing*. We implement low-overhead hash tables with Bloom filters [3], a structure in which a load or a store address is hashed to a single bit. If the bit is already set, there is a likely, but not a certain address match with another load or store. If the bit is unset there *cannot* be an address match

with another load or store. We use Bloom filters to evaluate the following LSQ improvements:

- *Search filtering:* Each load and store indexes into a location in the Bloom filter (BF) upon execution. If the indexed bit is set in the BF, a possible match has occurred, and the LSQ must be searched. If the indexed bit is clear, the bit is then set in the BF, but the LSQ need not be searched. However, all memory operations must still be allocated in the LSQ. This scheme reduces LSQ searches by 73-98% depending on the machine configuration.

- *Partitioned search filtering:* Multiple BFs each guard a different bank of a banked LSQ. When a load or store is executed, all the BFs are indexed in parallel. LSQ searches occur only in the banks where the indexed bit in the BF is set. This policy enables a banked CAM structure which reduces both the number of LSQ searches and the number of banks that must be searched. This scheme reduces the number of entries that must be searched by 86%.

- *Load state filtering:* A predictor examines each load upon execution and predicts if a store to the same address is likely to be encountered during the lifetime of the load. If so, the load is stored in the memory ordering queues. If the prediction is otherwise, the load address is hashed in a load BF and is not kept in any memory ordering queues. When stores execute, they check the load BF, and if a match occurs, a dependence violation may have occurred and the machine must perform recovery. With this scheme, the load queue can be completely eliminated, at a cost of 3% in performance for small instruction window machines. For large window machines, however, our results show that the currently used BF's hash functions cause too many unnecessary flushes and are not an effective solution.

With these schemes, we show that the area required for LSQs can be reduced marginally and, more importantly, that the power and latency for maintaining sequential memory semantics can be significantly reduced. This, in turn alleviates a significant scalability bottleneck to higher-performance architectures requiring large LSQs.

The rest of the paper is organized as follows: Section 2 surveys related work and shows that techniques proposed to date will cause unacceptable performance losses in future, large-window systems. Section 3 describes and reports the performance of the search filtering techniques. Section 4, describes load state filtering. Conclusions and a discussion of future work are provided in Section 5.

2. Conventional Load/Store Queues

Traditional methods of constructing LSQs have been effective for current-generation processors with limited number of instructions in flight. However, these traditional methods face several challenges when applied to high ILP machines of the future with large instruction windows. In this section we describe the range of organizations of memory disambiguation hardware and then show experimentally why solutions proposed to date are poor matches for future high-ILP architectures.

2.1. Historical Memory Ordering Hardware

Initially, simple sequential machines executed one instruction at a time and did not require hardware for enforcing the correct ordering of loads and stores. With the advent of speculative, out-of-order issue architectures, the buffering and ordering of in-flight memory operations became necessary and commonplace. However, the functions embodied in modern LSQ structures are the result of a series of innovations much older as described below.

Store Buffers: In early processors without caches, stores were long-latency operations. Store buffers were implemented to enable the overlap of computation with the completion of the stores. Early examples were the stunt box in the CDC 6600 [30] and the store data buffers in the IBM 360/91 [4]. More modern architectures separated the functionality of the store buffers into pre-completion and post-commit buffers. The pre-completion buffers, now commonly called store queues, hold speculatively issued stores that have not yet committed. Post-commit buffers are a memory system optimization that increases write bandwidth through write aggregation. Both types of buffers, however, must ensure that *store forwarding* occurs; when later load to the same address (henceforth called *matching*) are issued, they receive the value of the store and not a stale value from the memory system. Both types of store buffers must also ensure that two stores to the same address (*matching* stores) are written to memory in program order.

Load Buffers: Load buffers were initially proposed to temporarily hold loads while older stores were completing, enabling later non-memory operations to proceed [24]. Later, more aggressive out-of-order processors–such as IBM's Power4 [29] and Alpha 21264 [1]–permitted loads to access the data cache speculatively, even with older stores waiting to issue. The load queues then became a structure used for detecting *dependence violations*, and would initiate a pipeline flush if one of the older stores turned out to *match* (have the same address as) the speculative load. We define a memory operation that has the same address as at least one other in-flight memory operation of opposite type in the window as a *matching* address. Processors such as

the Alpha 21264 [27] and Power4 also used the load queue to enforce the memory consistency model, preventing two matching loads from issuing out of order in case a remote store was issued between them.

As window sizes increased, the probability that matching memory operations would be in-flight increased, as did the chance that they would issue in the incorrect order, resulting in frequent pipeline flushes. Memory dependence predictors [6, 21, 15] were developed to address this problem, allowing loads that were unlikely to match older stores to issue speculatively, but deferring loads that had often matched in-flight stores in the past.

2.2. LSQ Organization Strategies

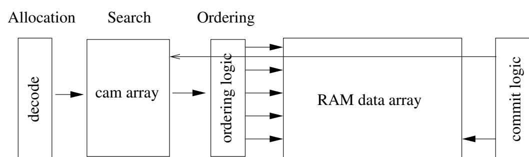


Figure 1. A simplified LSQ datapath

We show a simplified datapath for an LSQ[1] in Figure 1. The queues are divided into CAM and RAM arrays. Memory addresses are kept in the CAMs. The RAM array holds store data, load instruction targets, and other meta-information for optimizations. A memory instruction upon execution must perform two operations: *search* and *entry*. To search the LSQ, the operation searches the CAM array for matching addresses. Matching operations are emitted to the ordering logic, which determines whether a violation has occurred, or whether a value needs to be forwarded. There are two policies for entering instructions into the LSQ, which are described below.

2.2.1. Age-indexed LSQs The majority of LSQ designs have been *age indexed*, in which memory operations are *physically ordered* by age. They are entered into a specific row in the CAM and the RAM structures based on an age tag that is typically assigned at the decode or map stage and is associated with every instruction. In addition to determining into which slot a memory operation should be entered, the age tags are used to determine dependence violations and forwarding of store data as well as flushing the correct operations when a branch is found to be mispredicted. Since age-indexed LSQs act as circular buffers, they must be logically centralized and also fully associative, since every memory operation may have to search all other operations in the LSQ. Although fully associative structures are expensive in terms of latency and power, age indexing permits simpler circuitry for allocating entries, determining conflicts, committing stores in program order, and quick partial flushes triggered by mis-speculations.

1 Typically, separate structures are built for the load and store queues but to simplify the explanation we illustrate a single queue.

Related Work: Dynamically scheduled processors, such as those described by Intel [5], IBM [10], AMD [18] and Sun [23], use age-indexed LSQs. An LSQ slot is reserved for each memory instruction at decode time, which it fills upon issue. To reduce the occurrence of pipeline stalls due to full LSQs, the queue sizes are designed to hold a significant fraction of all in-flight instructions (two-thirds to four-fifths). For example, to support the 80-entry re-order buffer in the Alpha 21264, the load and store buffers can hold 32 entries each. Similarly, on the Intel Pentium 4, the maximum number of in-flight instructions is 128, the load buffer size is 48, and the store buffer size is 32.

Ponomarev et al. [26] proposed an age-indexed but segmented LSQ, in which a fully associative LSQ is broken into banks through which requests are pipelined, accessing one bank per cycle. This strategy ultimately saves little power or latency, since all entries must be searched in the common case of no match, and an operation must wait until a number of cycles equal to the number of banks has elapsed to determine that there were no conflicts. This scheme lends itself to efficient pipelining of LSQ searches for faster clock rates, but not necessarily higher performance.

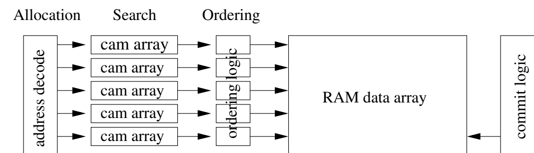


Figure 2. Address-indexed LSQ datapath

2.2.2. Address-indexed LSQs To reduce the state that must be searched for matches, partitioning of LSQs is desirable. *Address-indexed* LSQs logically break the centralized, fully associative LSQ into a set-associative structure. As shown in Figure 2, a portion of an memory instruction's address chooses the LSQ set, and then only the entries in that set are searched for a match. While this does reduce the number of entries that are searched, address-indexed organizations suffer from two major drawbacks. First, address-partitioned LSQs can have frequent overflows (since set conflicts are possible), resulting in more flushes than an age-indexed LSQ. Second, ordering and partial flushing become more difficult in an address-indexed LSQ because instructions to be flushed may exist in different sets. For the same reason, in-order commit of stores to memory is also more expensive.

To mitigate the conflict problem, sets can be made larger, in which case the latency, power, and partitioning advan-

tages diminish. Alternatively, the sets can be made more numerous, in which case the LSQ may requiring more area than a pure centralized design and the average utilization of the entries will be low. Thus, even though address-indexed LSQs can support sets residing in separate banks with localized ordering logic (enabling de-centralized LSQs), they incur both performance and complexity penalties.

Related Work: A number of proposed or implemented designs have used address-indexed LSQs to facilitate partitioning. The Itanium-1 microarchitecture uses a violation detection table called an ALAT [17], which is a 32-entry, 2-way set associative structure. Because conflicts can overflow a set in an address-partitioned LSQ, more entries can reduce the probability of conflicts; the ALAT can hold 32 entries, even though a maximum of 20 memory instructions can be in-flight. The Itanium-2 microarchitecture [20] implements a 32 entry fully associative structure reducing the probability of conflicts even more.

Both the IA-64 [13] compiler and the Memory Conflict Buffer paper [12] emphasize static disambiguation analysis to store only instructions whose addresses either have a true dependence or cannot be statically disambiguated, thus reducing the size of the hardware ALAT or MCB structures. An IA-64 study on dependence analysis [31], however, concedes that relying completely on static analysis is ineffective for programs that cannot tolerate the compile-time analysis cost (e.g. JITs) or non-native binaries for which source access is not available, and that static analysis is much less effective for many pointer-intensive codes. Static analysis can play a role but cannot address LSQ scaling issues comprehensively.

Finally, the MultiScalar processor proposed an address-indexed disambiguation table called the Address Resolution Buffer (ARB) [11]. When an ARB entry overflows (an ARB set has too many memory addresses), the MultiScalar stages are squashed and the processor rolls back. The MultiScalar compiler writers focused strongly on minimizing the probability of conflict in the ARB, trying to reduce the number of subsequent squashes.

2.3. Conventional LSQ Scalability

All of the previous schemes present one of two undesirable choices: (1) high power consumption and latency due to a large, fully associative, age-indexed scheme, or (2) increased stalls and/or rollbacks due to LSQ overflows with an address-indexed scheme. In this section, we analyze the scalability of age-indexed and address-indexed schemes to large windows, and find that neither approach provides sufficient scalability.

Experimental Infrastructure: We simulate a future large-window, 16-wide issue out-of-order processor with a 512-entry reorder buffer, using the sim-alpha simulator [7]. Table 1 summarizes the microarchitectural features of the target machine. To explore a range of in-flight instruction pressures on the LSQs, we simulate two 512-entry window configurations. The first, called low-ILP (LILP), is an Alpha 21264 microarchitecture scaled to the parameters shown in Table 1. The second, called High-ILP (HILP) is intended to emulate a more aggressive microarchitecture–assuming that other emerging bottlenecks are solved–to better stress the LSQs in our experiments. In particular, the HILP configuration assumes perfect (oracle) load-store dependence prediction and branch prediction.

Metrics: We simulated both configurations while varying the sizes and organizations of both age-indexed and address-indexed LSQ organizations, to determine the performance degradations caused by queues smaller than the instruction window. In these experiments, we optimistically assume that the LSQ structures can be accessed in one cycle, to isolate the performance effects of LSQ structural hazards. In addition to performance, the two other pertinent metrics we measure are the total number of LSQ entries required (which translates to area requirement) and the total number of entries associatively searched upon each LSQ access (which translates to LSQ energy consumption).

Age-indexed LSQ scalability: For, age-indexed structures, we measured the performance for load and store queues each with 64, 128 and 256 entries. The microarchitecture handles a full queue by throttling the map stage until LSQ entries are committed and available later for mapping.

Table 2 shows the performance of the age-indexed schemes, for the three sizes of LSQs. Almost no benchmark in either the LILP or HILP configuration had more than 128 loads or stores in flight at any time, thus the performance benefits of increasing the queue sizes from 128 to 256 entries each is negligible. Halving the queue sizes to 64 entries each causes a large 20% performance drop for HILP, but a mere 5% performance drop for LILP. This indicates that if future architectures are unable to fill their in-flight instruction windows, then centralized LSQs substantially smaller than the instruction window size can be used with negligible performance losses. However, larger performance losses will result if the processor is able to keep its window relatively full. Performance losses will also result from having centralized LSQs, since future distributed microarchitectures will be unable to access a centralized structure efficiently.

Address-indexed LSQ scalability: For the address-indexed structures, we vary the size from 128 entries total per load and store queue to 512 entries each. We simulated the queues partitioned into multiple sets, ranging from 4 to 32 partitions. The number of ways in each set ranged from 8 to 256, which is equivalent to the number of entries

Parameter	Configuration
Buffer Sizes	512-entry int, 512-entry fp issue window, 512-entry reorder buffer, separate load and store queues, 16-wide issue, 16-wide commit.
Instruction Supply	32 entry RAS, partitioned IL1 (64KB, 8 r/w ports, 2-cycle hit), 32 entry IL1 TLB, perfect and 2-level branch prediction, 16 wide fetch, fetches across branches, multiple branch prediction.
Data Supply	Partitioned DL1 (64KB, 8 r/w ports, 3-cycle hit), 64 MSHRs, 8 targets, 2MB L2, 8-cycle hit, 60-cycle main memory, 32-entry TLBs, oracle memory dependence prediction for HILP and store-wait prediction with a 2048-entry table for LILP.
Functional units	512 registers (int and fp), 16 int/fp units, 8 ld/st units, pipelined functional units.
Simulation	Single Sim-point regions of 100M for 19 SPECCPU 2000 benchmarks. The other benchmarks in the suite are incompatible with our experimental infrastructure.

Table 1. Simulation parameters for an 512-entry ROB machine

LQ/SQ Size	*Configuration*	
(Entries)	HILP (IPC)	LILP (IPC)
64/64	1.88	0.57
128/128	2.36	0.60
256/256	2.38	0.60

Table 2. Age-indexed LSQ performance for the LILP and HILP configurations

searched associatively upon each access.

Since age-indexed schemes are fully associative, there is only the possibility of a capacity overflow, and not a conflict. In address-indexed LSQ organizations, it is possible for all in-flight instructions with unknown addresses to issue and then map to the same set, causing an LSQ partition overflow even though other partitions might still have unoccupied entries.

We modeled two strategies for handling set overflows. The first is a preventive *stalling* policy in which the map stage stalls when the number of unresolved loads or stores in flight is sufficient to fill a partition completely. For example, if each load queue partition can hold N loads, and the fullest partition contains $N - 3$ loads, then the map stage must stall as soon as three additional unresolved loads are put into flight. As loads resolve to other banks, more can be permitted to pass the map stage. The second policy is a *flushing* policy that causes a pipeline flush whenever one of the partitions (sets) overflows.

LQ/SQ Organization	*Policy for Structural Hazards*	
	Stalling (IPC)	Flushing (IPC)
LQ/SQ Size: 128/128		
32 sets, 8 ways	0.45	1.97
16 sets, 16 ways	0.72	1.95
8 sets, 32 ways	1.00	1.91
4 sets, 64 ways	1.31	2.23
LQ/SQ Size: 256/256		
32 sets, 16 ways	0.71	1.92
16 sets, 32 ways	1.02	1.90
8 sets, 64 ways	1.34	2.27
4 sets, 128 ways	1.78	2.35
LQ/SQ Size: 512/512		
32 sets, 32 ways	1.01	1.88
16 sets, 64 ways	1.34	2.27
8 sets, 128 ways	1.80	2.33
4 sets, 256 ways	2.28	2.35

Table 3. Address-indexed LSQ performance for the HILP configuration with two structural hazard policies

Table 3 shows the performance across the address-indexed LSQ configurations. The stalling policy performs uniformly worse than the flushing policy across all configurations. The number of unresolved in-flight memory operations is commonly greater than the number of free spaces in the fullest partition, and pipeline flushes due to actual overflows are far less frequent.

Even though pipeline flushing upon overflow works better, 20% drops are common for all but the largest queues or those with the highest number of compares per access (on the order of 64), giving them the same problem faced by the age-indexed approaches. As can be seen from the tables, the two organizations have different tradeoffs; from the power point of view, lower associative address entry LSQs are advantageous but have lower performance. Address entry LSQs with performance comparable to age entry LSQs are twice the size and hence require more area.

2.4. LSQ Optimization Opportunities

Current-generation LSQs check all memory references for forwarding or ordering violations, since they are unable to diffentiate memory operations that are likely to require special handling from others that do not. Only a fraction of

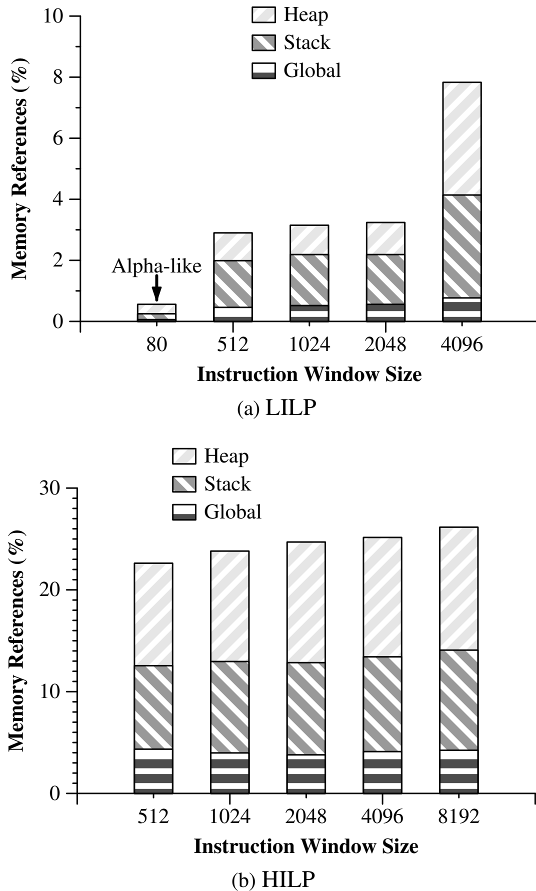


Figure 3. Percentage of matching memory instructions

memory operations match others in the LSQs, however, so treating all memory operations as worst case is unnecessarily pessimistic.

Figures 3(a) and 3(b) show the fraction of matching in-flight memory references for varying window sizes with the LILP and HILP configurations respectively. The fraction of matching addresses is extremely small for a window comparable to current processors. With an Alpha 21264-like window of 80 instructions with a realistic front end, fewer than 1% of memory instructions match. For the LILP configuration, a 512-instruction window sees 3% matching memory instructions. This rate remains essentially flat until the 4096-instruction window, at which point the matching instructions spike to nearly 8%. The HILP configuration, which has a much higher effective utilization of the issue window, has matching instructions exceeding 22% for a 512-entry window, which slowly grow to roughly 26% for an 8192-entry window.

Two results are notable in Figure 3(b). First, while the matching rates are close to two orders of magnitude greater than current architectures, three-quarters of the addresses in these enormous windows are *not* matching, indicating the potential for a four-fold reduction in the LSQ size. Second, the growth in matching instructions from 1K to 8K instruction windows is small, hinting that there may be room for further instruction window growth before the matching rate increases appreciably. Of course if the window is infinite, the matching rate will approach 100%.

Many of these matching instructions, however, are artifacts of the compilation and may be good candidates for removal. A significant fraction (approximately 50%) of the matching instructions are stack and global references. It is likely that more intelligent stack allocation and improved register allocation to remove spills and fills can eliminate many of the matching stack references.

3. Search Filtering

This section describes techniques to avoid searches for memory instructions that do not match, then reduce the LSQ power consumption and latency for instructions that do match. The first technique uses a Bloom filter predictor (BFP) to eliminate unnecessary LSQ searches for operations that do not match other operations in the LSQ. We then apply BFPs to separate LSQ partitions, reducing the number of partitions that must be searched when the BFP predicts that an LSQ search is necessary. Finally, we discuss other applications of BFPs to future partitioned primary memory systems.

3.1. BFP Design for Filtering LSQ Searches

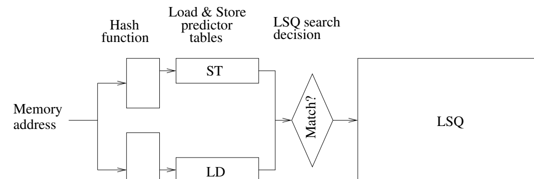


Figure 4. BFP Search Filtering: Only memory instructions predicted to match must search the LSQ; all others are filtered.

The Bloom Filter Predictor (BFP) used for filtering LSQ searches maintains an approximate and heavily encoded hardware record–as proposed by Bloom [3]–of the addresses of all in-flight memory instructions (Figure 4). Instead of storing complete addresses and employing associative searches like an LSQ, a BFP hashes each address to some location. In one possible implementation, each hash bucket is a single bit, which an memory instruction sets when it is loaded into the BFP and clears

when it is removed. Every in-flight memory address that has been loaded into the LSQ is encoded into the BFP. If a new hashed address finds a zero, it means that the address matches no other instruction in the LSQ, so the LSQ does not need to be searched. The instruction sets the bit to 1 and writes it back. If a 1 is found by an address hashing into the BFP, it means either that the instruction matches another in the LSQ or a hash collision (a *false positive*) has occurred. In either case, the LSQ must be searched. The BFP is fast because it is simply a RAM array with a small amount of state for each hash bucket. Bloom filters were used by Pier et al. [25] used for for early detection of cache misses.

The BFP evaluated in this section uses two Bloom filters: one for load addresses and other for store addresses, each of which has its own hash function and N locations. An issuing memory instruction computes its hash and then accesses the predictor of the opposite type (e.g. loads access the store table and vice versa). To detect multiprocessor read ordering violations, another Bloom filter with invalidation addresses is also checked by loads. Since our evaluation infrastructure is uniprocessor based the details of the invalidation Bloom filter are omitted.

3.1.1. Deallocating BFP Entries A bit set by a particular instruction should be unset when the instruction retires, lest the BFP gradually fill up and become useless. But if multiple addresses collide, unsetting the bits when one of the instructions retires will lead to incorrect execution, since a subsequent instruction to the same address might avoid searching the LSQ even though a match was already in flight. There are several solutions to this problem.

Counters: One solution uses up/down counters in each hash location instead of single bits. The counters track the number of instructions hashing into a particular location. Upon instruction execution the counter at the indexed location is incremented by one and upon commit the counter is decremented by one. The counters can either be made sufficiently large so as not to overflow, or they can take some other corrective action using one of the techniques described below when they overflow. The use of counter based Bloom filters was previously proposed by Fan et al. [9].

Flash clear: An alternative approach to using up/down counters, is to clear all of the bits in the predictor on branch mispredictions. A pipeline flush guarantees that no memory instructions are in flight and hence it is safe to reset all the bits. The flash clearing method has the advantage of requiring less area and complexity than the counters, but has the disadvantage of increasing the false positive rate.

Hybrid solution: A third approach that mixes the previous two involves freezing a counter when it overflows, so that all addresses that hash to that set perform LSQ searches, and then initiating a pipeline flush (or waiting for a misprediction) when the number of frozen hash buckets in the BFP grows too large. Our results have shown that 3-bit counters are sufficient for most table locations, for both load and store BFPs. The maximum number of collisions, across any benchmark with a 512-entry window, was 41. 6-bit counters should therefore be able to avoid overflows, but smaller 2- or 3-bit counters would likely be more efficient with this hybrid scheme.

3.1.2. Hash Functions To maximize the benefits of search filtering, the number of false positives must be minimized. The number of false positives depends on the quality of the hash function, the method used for unsetting the bits, and the size of the BFP tables. The BFP table must be sized larger than the number of in-flight memory operations, since the probability of a false positive is proportional to the fraction of set bits in the table.

There are two aspects that determine the efficacy of a hash function: (1) the delay through the hash function and (2) the probability of a collision in the hash table. Since the hash function is serialized with the BFP and then the LSQ search (if it is needed), we explored only two hash functions that were fast to compute, with zero or one level of logic, respectively. The first hash function, H_0, uses lower order bits of the address to index into the hash table, incurring zero delay for hash function computation. The second hash function, H_1, uses profiled heuristics to generate an index using the bits in the physical address that were most random on a per-benchmark basis. H_1 incurs a delay of one gate level of logic (a 2-input XOR gate). To determine H_1 for each benchmark, we populated a matrix by XORing each pair of bits of the address and adding the result to the appropriate position in the matrix. We then chose the bits that generated the most even number of zeros and ones, assuming that they were the most random.

3.1.3. BFP Results Table 4 presents a sensitivity analysis of the BFP false positives for a range of parameters, including varied predictor sizes ranging from one to four times the size of each load and store queue, the two hash functions H_0 and H_1, flash and counter clearing, and the three microarchitectural configurations used: the Alpha 21264, LILP, and HILP. The flash clearing results are not applicable to HILP because they rely on branch mispredictions, and HILP assumes a perfect predictor. As a lower bound, we include the expected number of false positives that would result, given the number of memory instructions in flight for each benchmark, assuming uniform hash functions[2]. The rate of false positives is averaged across the 19 benchmarks we used from the SPECCPU2000 suite.

---

2 Using probabilistic analysis the number of load (store) false positives assuming a uniform hash function can be estimated as: $\sum_i t_i \times (1 - (1 - \frac{1}{n})^i)$, where t_i is the number of store (load) searches occurring when there are i unique address in-flight loads(stores) and n is the load (store) BF.

	Configuration	Alpha 21264			LILP			HILP		
	BFP Size	32	64	128	128	256	512	128	256	512
Hash Type	*Clearing Method*									
H_0	Counter	5.7	2.6	1.6	8.4	2.8	2.0	15.0	8.8	4.3
H_1	Counter	3.9	2.3	1.4	5.7	3.3	2.0	10.1	6.1	4.2
H_0	Flash	59.9	53.3	49.2	30.2	28.6	25.9		n/a	
H_1	Flash	54.0	49.7	42.2	27.2	23.9	20.4		n/a	
Expected False Positives		2.8	1.6	1.0	5.7	3.2	1.7	9.6	5.3	2.8

Table 4. Percentage of False Positives for Various ILP Configurations and BFP Sizes

As expected, the table shows that the number of false positives decreases as the size of the BFP tables increase simply because of the reduced probability of conflicts. Flash clearing increases the number of false positives significantly over count clearing. However, the count clearing works quite effectively, especially using the H_1 hash function, showing less than a 2% false positive increase over the probabilistic lower bound. This result indicates that moderately sized BFPs are able to differentiate between the majority of matching addresses and those that have no match in flight. Furthermore, the lookup delay of all table sizes presented herein is less than one 8FO4 clock cycle at a 90nm technology. The power required to access the predictor tables is negligible compared to the associative lookup as the predictor tables are comparatively small, direct mapped RAM structures.

3.2. Partitioned BFP Search Filtering

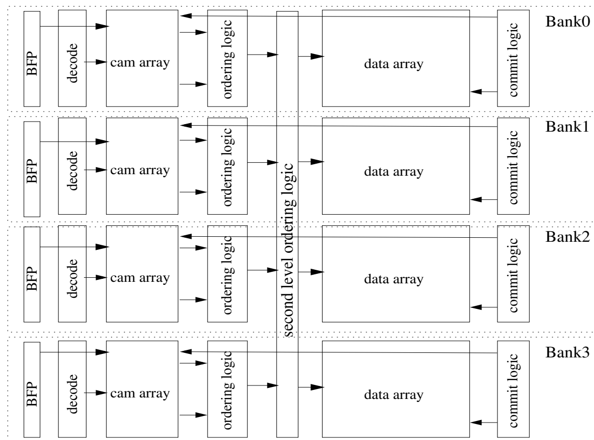


Figure 5. Partitioned Search Filtering: Each LSQ bank has a BFP associated with it (see far left)

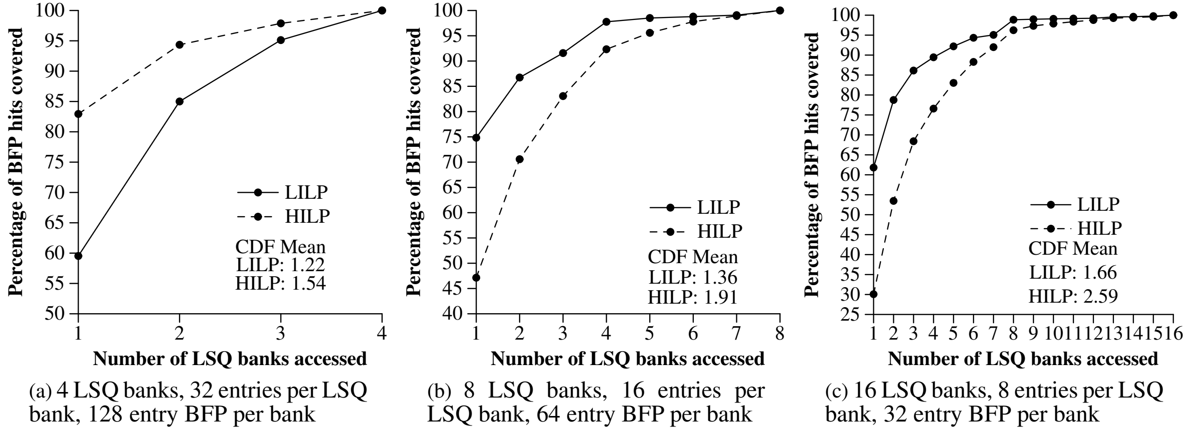
The previous section described the use of BFPs to prevent most non-matching addresses from expensive LSQ searches. In this section, we describe a BFP organization that extends the prior scheme to reduce the cost of matching address searches appreciably. A distributed BFP (or DBFP), shown in Figure 5, is coupled with a physically partitioned but logically centralized (in terms of ordering logic) LSQ. One DBFP bank is coupled with each LSQ bank, and each DBFP bank contains only the hashed state of those memory operations in its LSQ bank. Depending on the implementation of the LSQs and the partitioning strategy, some extra logic may be required to achieve correct memory operation ordering across the partitions.

Memory instructions are stored in the LSQ just as in previous sections, but an operation is hashed into the BFP bank associated with the physical LSQ bank into which it is entered instead of a larger centralized BFP as in the previous section. Before being hashed into the BFP bank, however, the address' hash is computed and used to lookup in all DBFP banks, which are accessed in parallel. Any bank that incurs a BFP "hit" (the counter is non-zero) indicates that its LSQ bank must be associatively searched. All banks finding address matches raise their match lines and the correct ordering of the operation is then computed by the ordering logic.

Depending on the LSQ implementation, the banking of the LSQ may have latency advantages over a more physically centralized structure. However, the power savings will be significant in a large-window machine if only a subset of the banks must be searched consistently. Figure 6 presents a cumulative distribution function of the number of banks that are searched on each BFP hit for both the HILP and LILP configurations, varying the number of LSQ banks from 4 to 16. The cumulative DBFP size was held at 512 entries for the different banking schemes. The results show that a DBFP can reduce the number of entries searched on a BFP hit appreciably; For the LILP configuration, 60% to 80% of the accesses result in the searching of only one bank. For the HILP configuration, 80% of the searches use four or fewer banks.

3.3. LSQ/BFP Organizations for Partitioned Cache Architectures

In high-ILP wider issue machines, as the number of simultaneously executing memory instructions increases, both the LSQs and the BFPs will need to be highly multiported. This section discusses organizations that still use a



(a) 4 LSQ banks, 32 entries per LSQ bank, 128 entry BFP per bank

(b) 8 LSQ banks, 16 entries per LSQ bank, 64 entry BFP per bank

(c) 16 LSQ banks, 8 entries per LSQ bank, 32 entry BFP per bank

Figure 6. Partitioned State Filtering for Banked LSQs and BFPs

logically centralized, age-indexed LSQ, but exploits BFPs to facilitate a disambiguation hardware organization that matches the bandwidth of the primary memory system.

The port requirements on the BFPs can be trivially reduced by banking them, using part of the memory address as an index, to select one of the BFP banks, that will hold only memory instructions mapped to its bank. Banking the BFPs lends itself naturally to a partitioned primary memory system where the L1 data caches (L1D) are also address interleaved, as shown in Figure 7a. In this organization, a portion of the DBFP guarding each physical LSQ bank is associated with each L1D bank. Upon an access to the L1D cache, its DBFP banks generate a bitmask which indicates the LSQ banks that need to be searched. If the memory operation hits in none of the DBFP banks (the common case), then the LSQ search can be avoided.

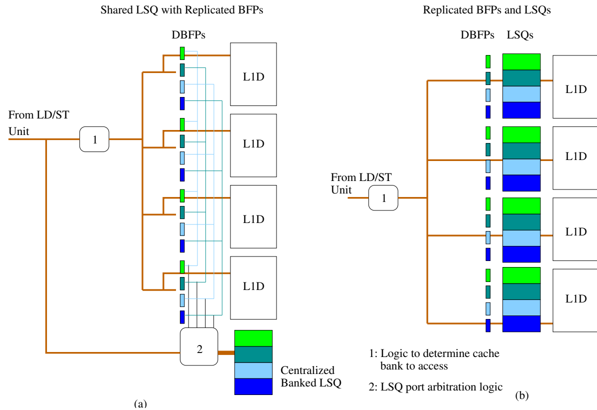


Figure 7. Replication of BFPs and LSQs to match L1D bandwidth

Even with distributed replicated BFPs, each memory instruction must still be sent and allocated in the LSQ. As long as simultaneously executing memory instructions must are targeted into different banks of the LSQ, no contention occurs. However, if the LSQ is to support parallel multi-banked accesses, extra circuitry must deal with buffering and collisions, increasing complexity. One simple solution to the problem is to replicate the banked LSQs as well. As with the DBFPs each replicated LSQ can be coupled with the statically address interleaved DL1 banks (Figure 7b), thus permitting all operations to complete locally at each partition. This scheme will also facilitate high bandwidth, low latency commit of stores to the L1D (assuming weak ordering is provided). Thus, replicated LSQs provide a complexity-effective solution but increase the area requirements significantly. In the next section we turn to schemes to reduce LSQ area, with the long-term goal being to reduce area sufficiently that completely replicated or distributed solutions become feasible.

4. Load State Filtering

Increasing instruction window sizes lead to a corresponding increase in the number of in-flight memory instructions, making it progressively less power- and area-efficient to enforce sequential memory semantics. A future processor with an 8K instruction window would need to hold, on average, between two and three thousand in-flight memory operations. Any two in-flight memory instructions to the same address (matching instructions) must be buffered for detection of ordering violations and forwarding of store values. However, as previously shown in Figure 3, a small fraction of the addresses in flight are typically matching. An ideal LSQ organization would buffer only in-flight matching instructions, permitting reductions in the LSQ area, latency, and power. Buffering fewer operations in the LSQ facilitates efficient implementation of partitioned, address-indexed schemes that are a better match

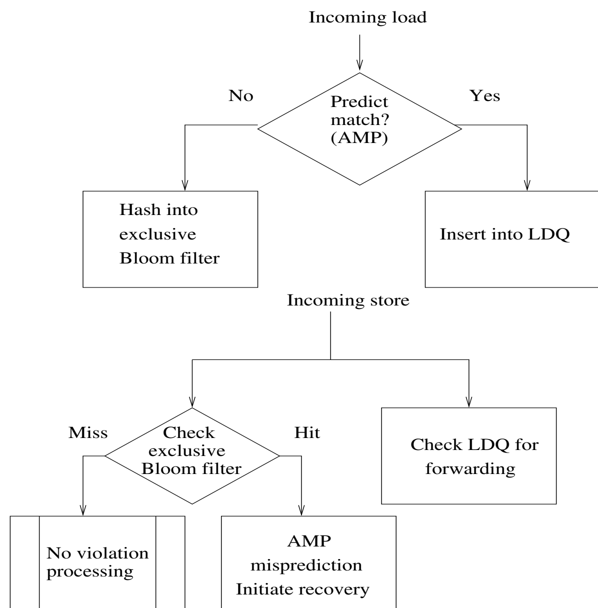


Figure 8. Load State Filtering

for future communication-dominated technologies.

In-flight store addresses and values must be buffered regardless of their interleaving with loads, since their values must only be written back to the memory system upon commit. The schemes presented in this section therefore attempt to reduce only the number of loads contained in the LSQs, by attempting to buffer only matching loads. Future work may use techniques such as the Speculative Versioning Cache (SVC) [14] to reduce the LSQs further by buffering only stores that are matching, placing non-matching stores in an SVC-like structure.

Figure 8 shows one possible scheme to reduce the loads that must be saved in the LSQ. An *address match predictor* (AMP) predicts whether a load is likely to match a store. If a load is predicted *not* to match, it is hashed into a structure called an *exclusive Bloom filter* (EBF), which contains an approximate hardware hash of all in-flight loads *not* contained in the load queue. If the load is predicted to match, it is placed into the load queue, which may be guarded by an *inclusive* Bloom filter for efficient accessing, as described in the previous section. Issued stores check the load queues as usual, but they also search the EBF. A store hashing to a set bit in the EBF indicates either a false positive hit, or a possible memory ordering violation due to incorrect hashing into the EBF by the AMP. Since the two cannot be differentiated, the processor must take corrective action, possibly culminating in a pipeline flush.

In this paper, we report only one simple, preliminary design, in which the load queue is completely eliminated and all loads are hashed into the EBF. If a load is issued before an older program-order store to the same address, every instruction past the store must be flushed when the store finds the hashed load in the EBF. This scheme thus guarantees correct execution without requiring any memory disambiguation hardware, but is likely to incur severe performance penalties due to flushes caused by dependence violations, which will grow as the window size is increased. That claim is buttressed by the results in Table 5, which show that for the Alpha-like configuration, the performance loss is a mere 3%, but for the HILP configuration, the average performance penalty is 34%. That large performance loss is due to flushes caused by three factors: false positives (EBF hash collisions of non-matching addresses), true dependence violations, and a matching load followed by a store issued in both program and temporal order (i.e. a flush caused on a *artificial* WAR hazard). Since the EBF occupies two-thirds the area of the original load queue, this particular solution saves too little area at too great a performance loss to be a viable solution for future high-ILP processors.

Two ideas that we are currently exploring eliminate the artificial WAR hazard-induced flushes and the false positive-induced flushes, respectively. The first idea stores an instruction number in an EBF-parallel structure, permitting a store that hits in the EBF to determine that the load hashed there was actually older in program order, thus requiring no corrective action. Identifying the exact conflicting load will also help reduce the cost of flushes by flushing only instructions after the conflicting load (including the load itself), instead of all instructions after the matching store, on a true dependence violation. The second idea involves placing a checker at the commit stage, to find a younger load that obtained an incorrect value which should have come from an older store. When that store hits in the EBF, it marks the checker to check all load instructions from the current newest instruction (“Y”) in the ROB to the instruction immediately following the store (“X”). This check can be done in parallel for high performance, or incrementally as instructions are committed. When instruction Y commits, if no matching load has been found, the hit in the EBF was a false positive, and no corrective action need be taken. This scheme may require multiple store values in the checker to be scanning the ROB at once, and requiring a centralized ROB, and so may not be a good match for future, more distributed architectures.

5. Conclusions

Conventional approaches for scaling memory disambiguation hardware for future processors are problematic. Fully associative load/store queues that can handle all inflight memory operations will be too slow and consume a large amount of power as reorder buffers grow. On the other hand, our analysis shows that smaller structures, that flush or stall when they fill, will incur significant performance

Benchmarks	Performance drop(%)	
	Alpha	HILP
164.gzip	0.0	35.5
171.swim	0.0	16.7
172.mgrid	0.0	46.2
173.applu	0.0	66.7
175.vpr	11.1	23.1
176.gcc	0.0	11.8
177.mesa	8.3	65.8
178.galgel	0.0	0.0
179.art	0.0	25.0
181.mcf	0.0	0.0
183.equake	0.0	30.5
188.ammp	0.0	25.0
189.lucas	0.0	25.0
197.parser	8.3	20.0
252.eon	25.0	58.1
253.perlbm	12.5	47.6
254.gap	0.0	73.1
256.bzip2	0.0	46.1
Average	3.3	34.2

Table 5. Percentage drop in performance for Alpha and HILP configuration with complete LDQ elimination

penalties. For example, in a 512-entry window machine, reducing the load and store queues (LSQs) from 128 to 64 entries each results in a 21% performance loss.

Solving this challenge by moving from fully associative to set associative, address-indexed LSQ partitions results in a different set of problems. Structural hazards occur more frequently unless each of the partitions themselves become in-feasibly large. To reduce performance losses below 10%, each of the partitions required 64 entries, resulting in a total LSQ size of twice the capacity of all in-flight instructions. For smaller set sizes, our results show that flushing the pipeline on an actual overflow is better than stalling instruction fetch on a *potential* overflow. With this scheme, 8-way partitions show a best-case performance loss of 19% due to flushing.

In this paper, we proposed a range of schemes that use approximate hardware hashing with Bloom filters to improve LSQ scalability. These schemes fall into two broad categories: *search filtering*, reducing the number of expensive associative LSQ searches, and *state filtering*, in which some memory instructions are allocated into the LSQs and others are encoded in the Bloom filters.

The search filtering results show that by placing a 4-KB Bloom filter in front of an age-indexed, centralized queue, 73% of all memory references can be prevented from searching the LSQ, including the 95% of all references that do not actually have a match in the LSQ. By banking the age-indexed structure and shielding each bank with its own Bloom filter, a small subset of banks are searched on each memory access; for a 512-entry LSQ, only 20 entries needed to be searched on average. We also proposed placing Bloom filters near partitioned cache banks, preventing a slow, centralized LSQ lookup in the common case of no conflict.

For state filtering, we coupled an address match predictor with a Bloom filter to place only predicted dependent operations into the LSQs, encoding everything else in a Bloom filter and intiating recovery when a memory operation finds its hashed bit set in the Bloom filter. These schemes were ineffective due to both false positives in the Bloom filter and dependence mispredictions, resulting in performance drops too large to justify a 37% reduction in LSQ area.

Future Directions: As instruction windows grow to thousands of instructions, hardware memory disambiguation faces severe challenges. First, the number of operations in flight with the same address will grow. Second, communication delays will force increased architectural partitioning, rendering a centralized LSQ impractical. It is possible that the search filtering methods that we have demonstrated to be effective upto 8K instruction windows may not be effective for larger window sizes.

We foresee several promising directions. First, by improving both dependence ("address match") predictors and Bloom filter hash functions, effective state filtering may make distributed, small LSQ partitions coupled with cache partitions feasible. Second, by encoding temporal information into Bloom filters rollback on artificial WAR hazards can be avoided. Third, software can help by partitioning references into classes preventing false conflicts as well, reducing in-flight address matches by renaming stack frames, and perhaps even explicitly marking communicating store/load pairs.

Approximate hardware hashing with Bloom filters provides an exciting new space of solutions for scalable LSQs. These structures may also find use in other high-power parts of the microarchitecture, such as highly associative TLBs, issue windows, downstream store queues, or other structures not yet invented.

Acknowledgments

We would like to thank members of the CART research group and the anonymous reviewers for comments on drafts of this paper. This research is supported by the Defense Advanced Research Projects Agency under contract F33615-01-C-1892, NSF instrumentation grant EIA-9985991, NSF CAREER grants CCR-9985109 and CCR-9984336, two IBM University Partnership awards, grants from the Alfred P. Sloan Foundation, Sun Microsystems, Intel Research Council and the O'Donnell Foundation.

References

[1] Alpha 21264 microprocessor hardware reference manual, July 1999. Compaq Computer Corporation.

[2] V. Agarwal, S. W. Keckler, and D. Burger. Scaling of microarchitectural structures in future process technologies. Technical Report TR2000-02, The University of Texas at Austin, February 2000.

[3] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[4] L. Boland, G. Granito, A. Marcotte, B. Messina, and J. Smith. The IBM system/360 model 91: Storage system. *IBM Journal of Research and Development*, 11(1):54–69, January 1967.

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

[6] G. Z. Chrysos and J. S. Emer. Memory dependence prediction using store sets. In *Proc. of the 25th Int'l Symp. on Computer Architecture*, pages 142–153, June 1998.

[7] R. Desikan, D. Burger, and S. W. Keckler. Measuring experimental error in microprocessor simulation. In *Proc. of the 28th Int'l Symp. on Computer Architecture*, pages 266–277, June 2001.

[8] D. Ernst and T. Austin. Efficient dynamic scheduling through tag elimination. In *Proc. of the 29th Int'l Symp. on Computer Architecture*, pages 37–46, May 2002.

[9] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area Web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[10] K. A. Feiste, B. J. Ronchetti, and D. J. Shippy. System for store forwarding assigning load and store instructions to groups and reorder queues to keep track of program order. U.S. Patent Application Number 6,349,382, 2002. Patent assigned to IBM.

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

[12] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. mei Hwu. Dynamic memory disambiguation using the memory conflict buffer. In *Proc. of the Sixth Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–193, Oct 1994.

[13] R. Ghiya, D. Lavery, and D. Sehr. On the importance of points-to analysis and other memory disambigution methods for C programs. In *Proc. of the ACM SIGPLAN 2001 conference on Programming Language Design and Implementation*, pages 47–58, 2001.

[14] S. Gopal, T. Vijaykumar, J. E. Smith, and G. S. Sohi. Speculative versioning cache. In *Proc. of the Fourth Int'l Symp. on High-Performance Computer Architecture*, pages 195–205, Feb 1998.

[15] J. Hesson, J. LeBlanc, and S. Ciavaglia. Apparatus to dynamically control the out-of-order execution of load-store instructions. U.S. Patent Application Number 5,615,350, 1995. Patent assigned to IBM.

[16] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal Q1*, 2001.

[17] J. Huck, D. Morris, J. Ross, A. Knies, H. Mulder, and R. Zahir. Introducing the IA-64 architecture. *IEEE Micro*, 20(5):12–23, Sept./Oct. 2000.

[18] W. A. Hughes and D. R. Meyer. Store to load forwarding using a dependency link file. U.S. Patent Application Number 6,549,990, 2003. Patent assigned to AMD.

[19] A. R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. A large fast instruction window for tolerating cache misses. In *Proc. of the 29th Int'l Symp. on Computer Architecture*, pages 59–70, May 2002.

[20] C. McNairy and D. Soltis. Itanium 2 processor microarchitecture. *IEEE Micro*, 23(2):44–55, Mar./Apr. 2003.

[21] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi. Dynamic speculation and synchronization of data dependences. In *Proc. of the 24th Int'l Symp. on Computer Architecture*, pages 181–193, June 1997.

[22] R. Nagarajan, K. Sankaralingam, D. Burger, and S. W. Keckler. A design space evaluation of grid processor architectures. In *Proc. of the 34th Int'l Symp. on Microarchitecture*, pages 40–51, December 2001.

[23] R. Panwar and R. C. Hetherington. Apparatus for restraining over-eager load boosting in an out-of-order machine using a memory disambiguation buffer for determining dependencies. U.S. Patent Application Number 6,006,326, 1999. Patent assigned to Sun Microsystems.

[24] Y. Patt, S. W. Melvin, W. mei Hwu, and M. Shebanow. Critical issues regarding HPS, a high performance microarchitecture. In *Proc. of the 18th annual workshop on Microprogramming*, December 1985.

[25] J.-K. Peir, S.-C. Lai, S.-L. Lu, J. Stark, and K. Lai. Bloom filtering cache misses for accurate data speculation and prefetching. In *Proc. of the 16th Int'l Conference on Supercomputing*, pages 189–198, Jun 2002.

[26] D. V. Ponomarev, G. Kucuk, and K. Ghose. Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources. In *Proc. of the 2001 Int'l Symp. on Microarchitecture*, pages 90–101, Dec 2001.

[27] R. L. Sites. *Alpha Architecture Reference Model*. Digital Press, 1992.

[28] E. Sprangle and D. Carmean. Increasing processor performance by implementing deeper pipelines. In *Proc. of the 29th Int'l Symp. on Computer Architecture*, pages 25–34, May 2002.

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

[30] J. Thornton. *Design of a Computer System: the Control Data 6600*. Scott, Foresman, and Company, 1970.

[31] Y. Wu, L.-L. Chen, R. Ju, and J. Fang. Performance potentials of compiler-directed data speculation. In *Proc. of the 2003 IEEE Int'l Symp. on Performance Analysis of Systems and Software*, pages 22–31, Mar 2003.