IEEE Micro Magazine: TOP PICKS from Computer Architecture Conferences - Nov/Dec 2004

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

Power is a particular problem with scaling memory disambiguation hardware to future out-of-order architectures, since the detection of memory ordering violations requires frequent associative searches of state that is proportional to the instruction window size. This paper proposes a new class of solutions that reduces the energy required to order loads and stores properly by an order of magnitude, potentially enabling processors to scale to window sizes of hundreds or thousands of instructions.

1. Introduction

Recently, several researchers have proposed mechanisms that enable issue windows to be scaled to hundreds or thousands of instructions [1, 5, 6, 11]. In these machines, hardware must perform dynamic memory disambiguation to guarantee that a memory ordering violation does not occur. Given that these systems employ 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 typical 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. 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.

To mitigate these LSQ bottlenecks we implement low-overhead hash tables with Bloom filters [2], 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 extend LSQs in two ways. First, with *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. This scheme reduces LSQ searches by 73-98% depending on the machine configuration. (b) Second, with *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 reduces both the number of LSQ searches and the number of banks that must be searched in a banked LSQ. On average, we reduced the number of entries to be searched by 86%.

With these schemes, we show that the power and latency for maintaining sequential memory semantics can be significantly reduced. This reduction alleviates a significant scalability bottleneck to higher-performance architectures that may otherwise require large LSQs.

2. Conventional Load/Store Queues

Traditional LSQ designs 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 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 several older innovations, 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. More modern architectures, like the Power 4, have 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 loads*) 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 [15]. Later, more aggressive out-of-order processors 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 (em i.e., have the same address as) the speculative load. Processors such as the Alpha 21264 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.

2.2. LSQ Organization Strategies

We show a simplified datapath for an LSQ[1] in Figure 2(a). 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. An arriving memory instruction 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.

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 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 the slot into which 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 any 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 misspeculations.

Previous Solutions: Dynamically scheduled processors, such as those described by Intel [4], IBM [8], AMD [10] and Sun [14], 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.

An alternative strategy that can be used for organizing the LSQs is to logically break down the full associative LSQ into a set associative structure and use addresses to index into the LSQ sets. The ARB [9] used in Multiscalar and the ALAT in Itanium are examples of address-indexed LSQs. This LSQ approach often reduces performance, because the machine must stall on potential set conflicts (structural hazards), since sets *must* be able to contain every in-flight ad-

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

dress that needs to fit into them. Since most addresses are unknown statically, in the worst case, all addresses might map to one set. Furthermore, ordering and partial flushing of data become problematic when instructions to be flushed reside in different sets.

2.3. Conventional LSQ Scalability

In this section, analyze the scalability of conventional LSQs. We measured two configurations: High ILP (or HILP) which assumed an Alpha 21264-like configuration with perfect branch prediction and dependence prediction, and low ILP (LILP), the same organization with realistic branch and dependence prediction. We found that for a 512-instruction window processor, having load and store queues with 128 entries was essentially equivalent as having infinite queues. However, halving the queue sizes to 64 entries caused a large 20% performance drop for the HILP configuration, and a 5% performance drop for the LILP configuration. This result indicates that large 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.

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

Figures 1(a) and 1(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 1(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 and/or energy. 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 (an infinite window will have a matching rate of close to 100%, of course).

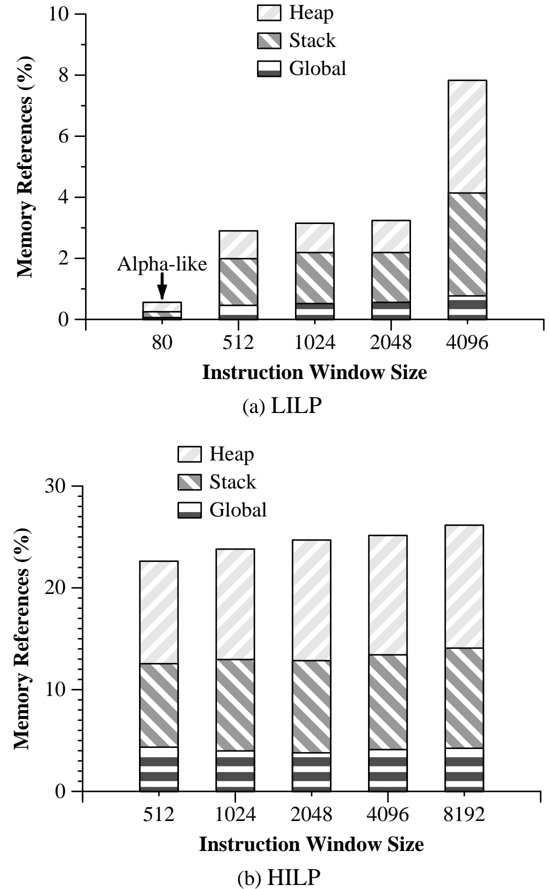


Figure 1. Percentage of matching memory instructions

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

Even though the number of matching instructions is very low the LSQs are not optimized for this case. With some simple filtering hardware the unmatching instructions can

be filtered from accessing the LSQ, thus making LSQs accesses efficient in the common case.

3.1. BFP Design for Filtering LSQ Searches

A Bloom filter predictor (BFP) can be used to eliminate unnecessary LSQ searches for many operations that will not match other entries in the LSQ. The BFP maintains an approximate and heavily encoded hardware record–as proposed by Bloom [2]–of the addresses of all in-flight memory instructions. 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 is guaranteed not to match other instructions in the LSQ, so the LSQ does not need to be searched. The instruction then sets the bit and writes it back. If a set bit is found by an address hashing into the BFP, it means either that the instruction matches an entry 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. Previously, Bloom filters were used by Pier et al. [16] used for early detection of cache misses.

The BFP evaluated in this section uses two Bloom filters: one for load addresses and another for store addresses, each of which has N locations and its own hash function. 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. In this paper, we focus on uniprocessor issues and do not evaluate an invalidation filter.

3.1.1. Deallocating BFP Entries The bit set by a particular instruction should be cleared when that 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 possible 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 count-based Bloom filters was previously proposed by Fan et al. [7].

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. When the number of frozen hash buckets grows too large, a pipeline flush can be initiated to bring down the matching rate. Our results have shown that 3-bit counters are sufficient for most table locations, for both load and store BFPs.

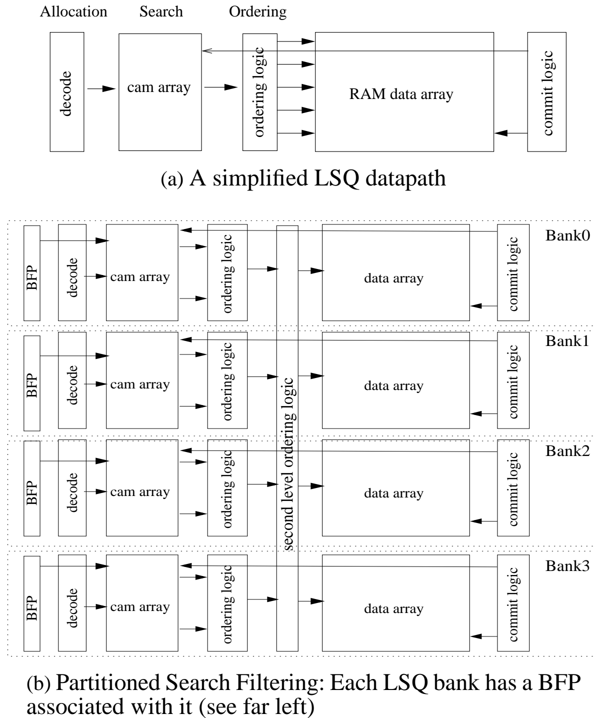
3.1.2. BFP Results Table 1 shows a sensitivity analysis of the BFP false positives for a range of parameters, varying four factors: two predictor sizes which are one and four times the size of each load and store queue, the two hash functions H_0 and H_1, flash and counter clearing, and the three microarchitectural 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.

The first hash function, H_0, uses the 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). 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. The rate of false positives is the arithmetic mean across the 19 benchmarks we used from the SPECCPU2000 suite.

As expected, the table shows that the number of false positives decreases as the size of the BFP sizes increase simply because of the reduced probability of conflicts. Flash clearing increases the number of false positives significantly over count clearing. 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. BFPs are sufficiently fast; the lookup delay of all table sizes presented herein is less than one 8FO4 clock cycle at a 90nm technology.

Configuration		Alpha 21264		LILP		HILP	
	BFP Size	32	128	128	512	128	512
Hash Type	*Clearing Method*						
H_0	Counter	93.7 (5.7)	97.8 (1.6)	88.7 (8.4)	95.1 (2.0)	63.2 (15.0)	73.3 (4.3)
H_1	Counter	95.5 (3.9)	98.0 (1.4)	91.3 (5.7)	95.0 (2.0)	67.5 (10.1)	73.4 (4.2)
H_0	Flash	39.5 (59.9)	50.2 (49.2)	66.1 (30.2)	71.1 (25.9)	n/a	
H_1	Flash	45.4 (54.0)	57.2 (42.2)	69.8 (27.2)	76.7 (20.4)	n/a	
Expected False Positives		2.8	1.0	5.7	1.7	9.6	2.8

Table 1. Percentage Activity Reduction and Percentage of False Positives (in brackets) for Various ILP Configurations and BFP Sizes



(a) A simplified LSQ datapath

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

Figure 2. Monolithic and Partitioned LSQ Organization

3.2. Partitioned BFP Search Filtering

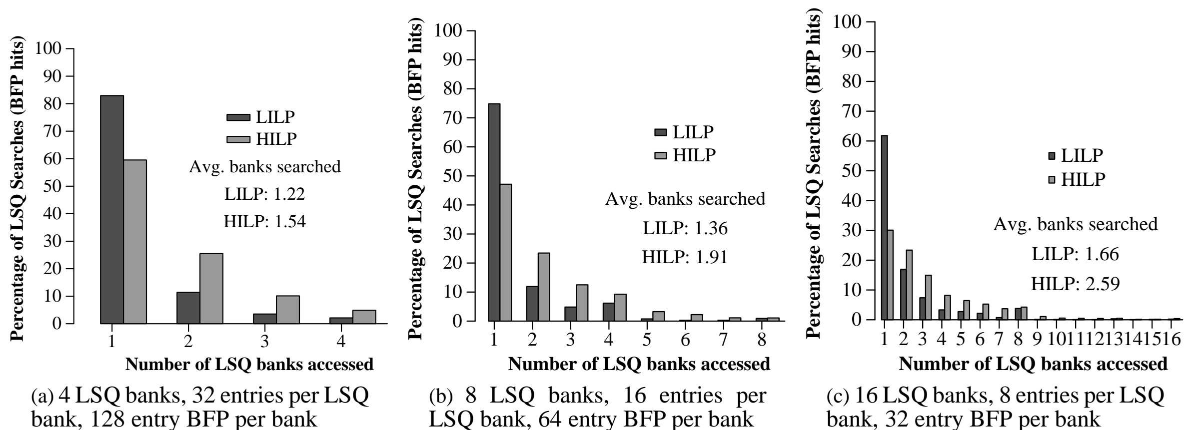
BFPs prevent most non-matching addresses from incurring expensive LSQ searches. In this section, we extend the BFPs to reduce the cost of LSQ searches when they occur. A distributed BFP (or DBFP), shown in Figure 2(b), is coupled with a physically partitioned but logically centralized LSQ. One DBFP bank is coupled with each LSQ bank, and each DBFP bank contains only the hashed state of those memory operations in its LSQ bank. 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 search all DBFP banks, which are accessed in parallel. The parallel lookup trades up-front energy for a reduced number of associative LSQ bank searches, for a net energy savings. 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. If only a subset of the banks must be searched consistently, then the power savings will be significant in a large-window machine. Figure 3 presents the distribution of number of banks that are searched on each BFP hit (the non-filtered accesses) 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.

4. 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 too much energy as reorder buffers grow. On the other hand, our analysis shows that smaller structures, which flush or stall on resource hazards, will incur significant performance 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.

In this paper, we proposed a range of schemes that use approximate hardware hashing with Bloom filters to im-



(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 3. Partitioned State Filtering for Banked LSQs and BFPs

prove LSQ scalability. These schemes fall into two broad categories: *search filtering*, reducing the number of expensive associative LSQ searches, and *state filtering* (described in an earlier version [18]), 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 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. The Bloom filters can also be placed near partitioned cache banks, preventing a slow, centralized LSQ lookup in the common case of no conflict.

Emerging Issues: 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. Third, the area occupied by the disambiguation hardware will grow forcing designers to look for area-efficient implementations.

In the past year, several researchers have proposed some solutions to the above problems. Park et al. [12] propose using the dependence predictor to filter searches to the store queue. Cain and Lipasti [3] propose a value-based approach to memory ordering that enables scalable load queues. Roth [17] has proposed combining Bloom Filters, address-partitioned store queues, and FIFO retirement queues to construct a high-bandwidth load store unit.

We foresee several promising directions towards a comprehensive solution to the above problems. First, by improving both dependence predictors and Bloom filter hash functions, effective state filtering may make distributed, area-efficient LSQ partitions coupled with cache partitions feasible. Second, 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 (as done in hardware by Moshovos and Sohi [13]). These techniques could ultimately lead to the most area, power and latency-efficient disambiguation hardware.

Memory disambiguation mechanisms are fundamental for maintaining sequential memory semantics. Approximate hardware hashing with Bloom filters provides an exciting new space of solutions for designing scalable and efficient disambiguation hardware. 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 gracefully acknowledge our sponsors: The Defense Advanced Research Projects Agency (contract F33615-01-C-1892), NSF (instrumentation grant EIA-9985991,CAREER grants CCR-9985109 and CCR-9984336), IBM (University Partnership awards), the Alfred P. Sloan Foundation and the Intel Research Council.

5. Biography

- Simha Sethumadhavan is a PhD student in CS at UT-Austin. His research interests include computer architecture, distributed systems and application of concepts developed in computer science to other sciences.
- Rajagopalan Desikan is a PhD student in ECE at UT-Austin. He is the author of the *sim-alpha* simulator, that models the

alpha 21264. His research interests include all aspects of computer architecture.

- Doug Burger is an Associate Professor of Computer Sciences at UT-Austin. Dr. Burger has a PhD in computer science from the University of Wisconsin, is an Alfred P. Sloan Foundation fellow, a member of the IEEE and a senior member of the ACM.
- Charles R. Moore is a senior research fellow at AMD. Previously, he was a research fellow at UT-Austin and the chief engineer on IBM's Power4 and PowerPC 601 microprocessors.
- Stephen W. Keckler is an Associate Professor of Computer Sciences at UT-Austin. Dr. Keckler has a PhD in computer science from MIT, is an Alfred P. Sloan Foundation fellow and a member of IEEE and ACM.

Direct questions and comments about this article to Doug Burger, Department of Computer Sciences, The University of Texas at Austin, 1 University Station C0500, Austin, TX, 78712; dburger@cs.utexas.edu.

References

[1] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. In *Proc. of the 2003 Int'l Symp. on Microarchitecture*, pages 423–433, Dec 2003.

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

[3] H. W. Cain and M. H. Lipasti. Memory ordering: A value-based approach. In *Proc. of Int'l Symp. on Computer Architecture*, pages 90–100, June 2004.

[4] 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.

[5] A. Cristal, J. F. Martinez, J. Llosa, and M. Valero. A case for resource-conscious out-of-order processors: towards kilo-instruction in-flight processors. *ACM SIGARCH Computer Architecture News*, 32(3):3–10, June 2004.

[6] Doug Burger, et.al. Scaling to the End of Silicon with EDGE Architectures. *IEEE Computer*, 37(7):44–55, July 2004.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] ll Park and Chong Liang Ooi and T.N. Vijaykumar. Reducing design complexity of the load/store queue. pages 411–422, Dec 2003.

[13] A. Moshovos and G. S. Sohi. Streamlining inter-operation memory communication via data dependence prediction. In *Proc. of the 30th Int'l Symp. on Microarchitecture*, Dec 1997.

[14] 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.

[15] 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.

[16] 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.

[17] A. Roth. A high bandwidth load store unit for single-and multi-threaded processors. Technical Report MS-CIS-04-08, University of Pennsylvania, 2004.

[18] S. Sethumadhavan, R. Desikan, D. Burger, C. R. Moore, and S. W. Keckler. Scalable Hardware Memory Disambiguation for High ILP Processors. In *Proc. of the 36th Ann. Symp. on Microarchitecture*, pages 399–410, Dec 2003.