MEMORY ORDERING: A VALUE-BASED APPROACH

VALUE-BASED REPLAY ELIMINATES THE NEED FOR CONTENT-ADDRESSABLE MEMORIES IN THE LOAD QUEUE, REMOVING ONE BARRIER TO SCALABLE OUT-OF-ORDER INSTRUCTION WINDOWS. INSTEAD, CORRECT MEMORY ORDERING IS MAINTAINED BY SIMPLY RE-EXECUTING CERTAIN LOAD INSTRUCTIONS IN PROGRAM ORDER. A SET OF NOVEL FILTERING HEURISTICS REDUCES THE AVERAGE ADDITIONAL CACHE BANDWIDTH DEMANDED BY VALUE-BASED REPLAY TO LESS THAN 3.5 PERCENT.

Harold W. Cain
IBM Research

Mikko H. Lipasti
University of Wisconsin

Hardware structures that extract instruction-level parallelism in out-of-order processors are often constrained by clock cycle time. Reaching higher frequency requires restricting the capacity of such structures, hence decreasing their ability to extract parallelism. Recent research on mitigating this negative feedback loop describes numerous approaches for novel structures that are amenable to high frequency without degrading instructions per cycle (IPC), but very little has focused on the load or store queues. These queues usually consist of content-addressable memories that provide an address matching mechanism for enforcing the correct dependences among memory instructions.

When a load instruction issues, the processor searches the store queue CAM for a matching address. When a store address is generated, the processor searches the load queue CAM for prior loads that incorrectly and speculatively issued before the store instruction.

Depending on the supported multiprocessor consistency model, the processor might search the load queue when every load issues or upon the arrival of an external invalidation request, or both. As the instruction window grows, so does the number of in-flight loads and stores, resulting in a delay of each search because of an increased CAM access time.

To prevent this access time from affecting a processor's overall clock cycle time, recent research has explored variations of conventional load and store queues that reduce the size of the CAM structure through a combination of filtering, caching, and segmentation. Sethumadhavan et al. employ bloom filtering of load and store queue searches to reduce the frequency of accesses that must search the entire queue.[1] Akkary et al. explore a hierarchical store queue organization where a level-one store queue contains the most recent *n* stores, while a larger, slower, level-two buffer contains prior stores.[2] Park et al. explore the use of a store-set predictor to reduce store queue search bandwidth by filtering those loads predicted to be independent of prior stores. Removing loads

Published by the IEEE Computer Society

0272-1732/04/$20.00 © 2004 IEEE

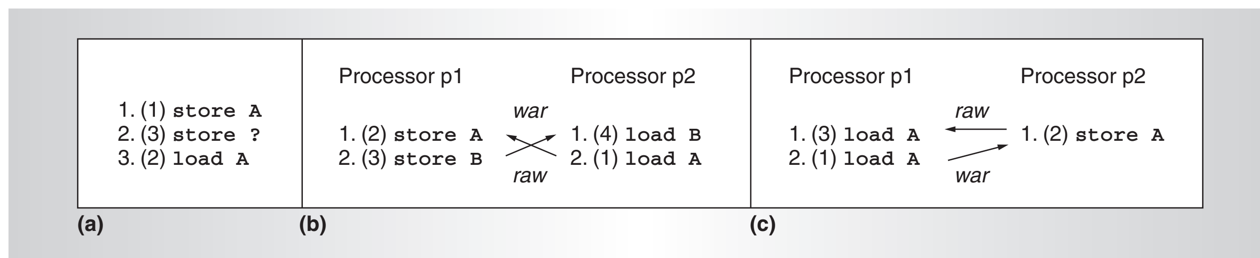

Figure 1. Examples of correctly supporting out-of-order loads: uniprocessor RAW hazard (a), multiprocessor violation of sequential consistency (b), and multiprocessor violation of coherence (c).

that the processor has not reordered with respect to other loads reduces pressure on the load queue CAM, and this work also explores a variable-latency, segmented LSQ.[3]

Ponomorev et al. explore the power-saving potential of a segmented load queue design, which disables certain portions of the load or store queue when occupancy is low; but they do not address the load queue scalability problem.[4]

Value-based replay enforces memory ordering by simply re-executing load instructions in program order prior to commit, eliminating the need for associative search functionality from the load queue. The load queue can therefore be implemented as a first-in first-out buffer, like the reorder buffer, which is fundamentally more scalable and power-efficient. In order to mitigate the bandwidth and resource occupancy costs of replay, a set of heuristics filter the set of loads that must be replayed, resulting in negligible performance degradation and data cache bandwidth increases relative to a conventional machine.

Associative load queue design

The load queue ensures that speculatively reordered loads are correct with respect to read-after-write dependences on prior stores and multiprocessor consistency constraints. Figure 1a contains a code segment illustrating a potential violation of a uniprocessor RAW (read-after-write) hazard. We label each operation with its program order and issue order (in parentheses).

In this example, the load instruction speculatively issues before the processor computes the previous store's address. Conventional machines enforce correctness by associatively searching the load queue each time they compute a store address. If the queue contains an already-issued prior load whose address overlaps the store, the processor squashes the load and reexecutes it. In this example, if the second store overlaps address A, the load queue search will match, and the processor will squash the load of A.

By considering how load queue implementations enforce the memory consistency model, we can categorize them into two basic types:

- *snooping load queues,* which are searched by external invalidation requests, and
- *insulated load queues*, which are not searched by such requests.

In snooping load queues, the memory system forwards external write requests to the load queue; these requests search for already-issued loads whose addresses match the invalidation address,[5] squashing any overlapping load. Block replacements in an inclusive cache hierarchy will also result in an external load queue search. Insulated load queues enforce the memory consistency model without processing external invalidations by squashing and replaying reordered loads. The exact implementation depends on the system's memory consistency model, although either of these two types can support any consistency model.

Figure 1b illustrates a multiprocessor code segment where processor p2 has reordered two loads to different memory locations that processor p1 writes in the interim. In a sequentially consistent system, this execution is illegal because one cannot construct a total order of execution for the instructions. A snooping load queue detects this error when it finds address A from p1's store in the load queue of p2, and squashes p2's second load. An insulated load queue prevents the error by noting that the loads to B and A issued out of

Table 1. Load queue attributes for current dynamically scheduled processors.

Processor	Estimated no. of read ports	Estimated no. of write ports
Compaq Alpha 21364 (32-entry load queue, max 2 load or store address generations per cycle)	2 (1 per load or store issued per cycle)	2 (1 per load issued per cycle)
HAL SPARC64 V (size unknown, max 2 loads and 2 store address generations per cycle)	3 (2 for stores, 1 for external invalidations)	2
IBM Power 4 (32-entry load queue, max 2 load or store address generations per cycle)	3 (2 for loads and stores, 1 for external invalidations)	2
Intel Pentium 4 (48-entry load queue, max 1 load and 1 store address generations per cycle)	2 (1 for stores, 1 for external invalidations)	2

order (potentially violating consistency) and squashing load A.

Processors that support strict consistency models do not usually use insulated load queues; they are overly conservative. Insulated load queues are a better match for weaker models with fewer consistency constraints. For example, in weak ordering, the system need only order those operations separated by a memory barrier instruction or that read or write the same address. The Alpha 21364 supports weak ordering by stalling dispatch at every memory barrier instruction (enforcing the first requirement) and uses an insulated load buffer to order those instructions that read the same address. Using the example in Figure 1c, if p1's first load A reads the value written by p2, then p1's second load A must also observe that value. An insulated load queue enforces this requirement by searching the queue when each load issues and squashing any subsequent load to the same address that has already issued. Snooping load queues are simpler in this respect because loads need not search the load queue.

To reduce the frequency of load squashes, the IBM Power4 uses a hybrid approach that snoops the load queue, marking (instead of squashing) conflicting loads. Every load must still search the load queue at issue time; however, the processor must only squash those marked by a snoop hit.

Designers usually implement load queues using a RAM structure containing a set of entries organized as a circular queue. They also use an associated CAM to search for queue entries with a matching address. The latency of searching the load queue CAM is a function of its size and the number of read and write ports. The processor's load issue width determines write port size; each issued load must store its newly generated address into the appropriate CAM entry.

The CAM must contain a read port for each issued store, each issued load (in weakly ordered implementations), and usually an extra port for external accesses in snooping load queues. Table 1 shows a summary of their size in current generation processors with separate load queues and an estimate of their read or write port requirements. Typical processors use load queues with sizes in the range of 32 to 48 entries. They allow the issue of some combination of two loads or stores per cycle, resulting in a queue with two or three read ports and two write ports.

Value-based replay

The driving principle behind our design is to shift complexity from the pipeline's timing-critical components to its back end. During a load's premature execution, it is executed the same as in a conventional machine: at issue, the load searches the store queue for a matching address, if it finds none and a dependence predictor indicates that there will not be a conflict, the load proceeds. Otherwise, it stalls. After issue, there is no need for loads or stores to search the load queue for incorrectly reordered loads. Instead, the pipeline re-executes some loads at its back end and checks their results against the premature load result. Supporting this load replay mechanism adds two pipeline stages before the commit stage; we refer to these additional stages as the replay and compare stages.

During the replay stage, certain load instructions access the level-one data cache a

second time, after all prior stores have written their data to the cache. Because each replay load also executed prematurely, this replay is inexpensive in terms of latency and power consumption. The replay load can reuse the premature load's effective address and translated physical address, and it will almost always be a cache hit. Because stores must perform their cache access at commit, the pipeline back end already contains a data path for store access; we assume that loads can also use this cache port during the replay stage.

The compare stage compares the replay load value to the premature load value. If the values match, the premature load was correct, and the instruction proceeds to the commit stage for subsequent retirement. If the values differ, the compare stage deems the premature load's speculative execution as incorrect, and it invokes a recovery mechanism.

Because the replay mechanism enforces correctness, we can replace the associative load queue with a simple first-in first-out buffer that contains the premature load's address and data, in addition to the usual metadata stored in the load queue. To ensure correct execution, however, we must take care in designing the replay stage. The following three constraints guarantee that we catch any ordering violations:

- All prior stores must have committed their data to the L1 cache. This ensures that all replay loads have correctly satisfied RAW dependences.
- The replay stage must replay all loads in program order. To enforce consistency constraints, a processor must not observe the writes of other processors out of their program order, which could happen if replayed loads are reordered. If multiple replays occur per cycle and one is a cache miss, forcing subsequent loads to replay after resolving the cache miss ensures correctness.
- After a squash recovery, the processor should not, for a second time, replay a dynamic load instruction that causes a replay squash. This rule ensures forward progress in pathological cases where contention for a piece of shared data can persistently cause a premature load and a replay load to return different values.

This replay mechanism is similar to the use of *load-verify* instructions in the Daisy binary translation system.[6] Looking at it another way, it is like an á la carte version of Diva,[7] checking only load instructions rather than all instructions.

Naively, a processor should replay all loads to guarantee correct execution. Unfortunately, replaying loads has two primary costs that we would like to avoid:

- Replay can become a performance bottleneck given insufficient cache bandwidth for replays or because of the additional resource occupancy.
- Extra cache accesses and word-sized compare operations consume energy.

To mitigate these penalties, we propose methods for avoiding most replays.

We define four filtering heuristics for filtering the set of loads that the processor must replay to ensure correct execution. Filtered loads continue to flow through the replay and compare pipeline stages before reaching commit, however they do not incur cache accesses, value comparisons, or machine squashes.

Three filtering heuristics eliminate load replays while ensuring the execution's correctness with respect to memory consistency constraints. Another replay heuristic filters replays while preserving uniprocessor RAW dependences.

Filtering replays while enforcing memory consistency

The issues associated with avoiding replays while also enforcing memory consistency constraints are fairly subtle.

To assist with our reasoning, we employ an abstract model, called a constraint graph, to model multithreaded execution.[8,9] The constraint graph is a directed graph consisting of a set of nodes representing dynamic instructions in a multithreaded execution; edges that dictate constraints on the correct ordering of those instructions connect the nodes. For the purposes of this work, we assume a sequentially consistent system with four edge types: program order edges that order all memory operations executed by a single processor, and the standard RAW, write-after-read (WAR), and write-after-write (WAW) dependence edges

that order all memory operations reading from or writing to the same memory location.

The constraint graph is a powerful tool for reasoning about parallel executions because you can use it to test the correctness of an execution by simply testing the graph for a cycle. The presence of a cycle indicates that there is no total order of instructions, which would violate sequential consistency.

Three replay filters detect those load operations that require replay to ensure correctness. Two are based on the observation that any cycle in the constraint graph must include dependence edges to connect instructions executed by two different processors. If the processor does not reorder an instruction with respect to another instruction whose edge spans two processors, then there is no potential for consistency violation.

No-recent-miss filter

One method of inferring the lack of a constraint graph cycle is to monitor the occurrence of cache misses in the cache hierarchy. If no cache blocks have entered a processor's local cache hierarchy from an external source (that is, another processor's cache) while an instruction is in the instruction window, then there must not exist an incoming edge (RAW, WAW, or WAR) from any other processor in the system to any instruction in the out-of-order window. Consequently, we can infer that no cycle can exist, and therefore there is no need to replay loads to check consistency constraints.

No-recent-snoop filter

The no-recent-snoop filter is conceptually similar to the no-recent-miss filter, only it detects the absence of an outgoing constraint graph edge, rather than an incoming edge. Outgoing edges are detectable by monitoring the occurrence of external write requests. If other processors do not invalidate a local cache block while a load instruction is in the out-of-order window, then there must not exist an outgoing WAR edge from any load instruction at this processor to any other processor. The in-order commitment of store data to the cache prevents reorderings across outgoing WAW and RAW edges. In using the no-recent-snoop filter, the processor's replay stage only replays loads if they were in the out-of-order instruction window at the time the processor core observed an external invalidation (to any address).

No-reorder filter

We base the no-reorder filter on the observation that the processor often executes memory operations in program order. If so, the instructions must be correct with respect to memory ordering, therefore there is no need to replay a load.

Filtering replays while enforcing uniprocessor RAW dependences

To minimize the number of replays needed to enforce uniprocessor RAW dependences, we observe that most load instructions do not issue out of order with respect to prior unresolved store addresses. The *no-unresolved-store filter* identifies loads that, when issued prematurely, did not bypass any stores with unresolved addresses.

The store queue can identify these loads at issue time, when loads search the store queue for conflicting writes from which to forward data. Park et al. used a similar filter to reduce the number of load instructions inserted into the load queue.

Filter interactions

Of the four filters we just described, we can only use the no-reorder filter in isolation; each of the other three are too aggressive to use in isolation. The no-recent-snoop and no-recent-miss filters eliminate all replays other than those of use for inferring the correctness of memory consistency, at the risk of breaking uniprocessor dependences. Likewise, the no-unresolved-store filter eliminates all replays except those needed to preserve uniprocessor RAW dependences, at the risk of violating the memory consistency model. Consequently, we should pair the no-unresolved-store filter with either the no-recent-snoop or no-recent-miss filters to ensure correctness. For further improvement, it is possible to use the no-recent-snoop and no-recent-miss filters simultaneously, however we find that these filters work well enough that we do not need to explore this option. In the next section, we evaluate value-based replay using these filters.

Value-based replay versus conventional load queues

We collected our experimental data using PHARMsim, an out-of-order superscalar processor model integrated into a PowerPC

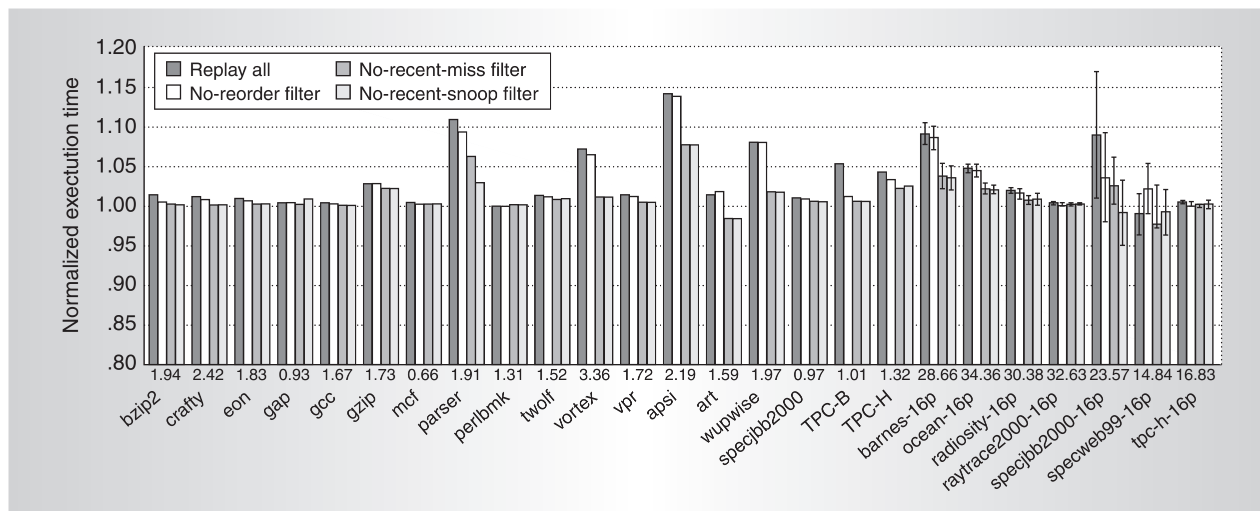

Figure 2. Value-based replay performance relative to baseline. The plot shows baseline IPC beneath each bar.

version of the SimOS full system simulator. We evaluate the value-based replay implementation in the context of both a uniprocessor system and a 16-processor shared-memory multiprocessor, using a fairly aggressive 15-stage 8-wide processor model to demonstrate value-based replay's ability to perform ordering checks without hindering performance.

For uniprocessor experiments, we use the SPECint2000 benchmark suite, three SPECfp2000 benchmarks (apsi, art, and wupwise), and a few commercial workloads (TPC-B, TPC-H, and SPECjbb2000). We selected the three floating-point benchmarks because of their high reorder-buffer utilization, a trait with which value-based replay might negatively interact. For multiprocessor experiments, we use several commercial workloads and applications from the SPLASH-2 parallel benchmark suite. Due to the variability inherent to multithreaded workloads, we use the statistical methods recommended by Alameldeen and Wood to collect several samples for each multiprocessor data point, adding error bars signifying 95 percent statistical confidence.[10]

Figure 2 presents the performance of the value-based-replay machine using four different filter configurations: no filters enabled (labeled "replay all"); the no-reorder filter in isolation; the no-recent-miss and no-unresolved-store filters in tandem; and the no-recent-snoop and no-unresolved-store filters in tandem. We normalized this data to the baseline machine's performance. Value-based replay is very competitive with the baseline machine despite the use of a simpler dependence predictor.

Without the use of any filtering mechanism, the value-based scheme incurs a performance penalty of only 3 percent on average. The primary cause of this performance degradation is an increase in reorder-buffer occupancy.

Figure 3 shows the increase in level-one data cache references for each of the value-based configurations. We break each bar into two segments: replays that occur because the load issued before the resolution of a prior store's address and replays performed irrespective of uniprocessor RAW constraints. Without filtering any replays, accesses to the level-one data cache increase by 49 percent on average, ranging from 32 to 87 percent, depending on the percentage of cache accesses from wrong-path speculative instructions and the fraction of accesses that are stores.

A single back-end load/store port limits this machine configuration to a single replay per cycle, which leads to an increase in average reorder-buffer utilization because of cache port contention. This contention results in performance degradation because of an increase in reorder buffer occupancy and subsequent reorder buffer allocation stalls. Although this performance degradation is small on average, performance loss is significant in a few applications.

When the no-reorder filter is enabled,

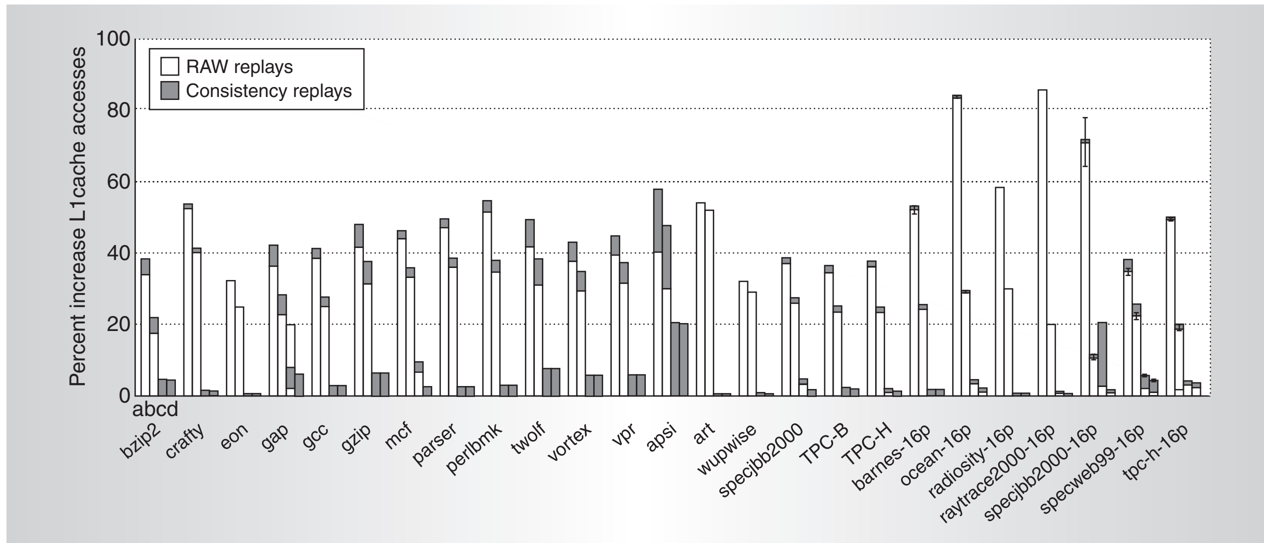

Figure 3. Increased data cache bandwidth because of replay: replay all (a), and no-reorder (b), no-recent-miss (c), and no-recent-snoop (d) filters.

value-based replay performance improves, although not dramatically. The no-reorder filter is not a very good filter of replays, reducing the average cache bandwidth replay overhead from 49 to 30.6 percent, indicating that most loads do execute out-of-order with respect to at least one other load or store. The no-recent-snoop and no-recent-miss filters, when used in conjunction with the no-unresolved-store filter, solve this problem. For single-processor machine configurations, the processor does not observe snoop requests other than from coherent I/O operations issued by the direct-memory-access controller; such operations are relatively rare for these applications. Consequently, the no-recent-snoop filter does a better job of filtering replays than the no-recent-miss filter. This is also true in the 16-processor machine configuration, where an inclusive cache hierarchy shields the processor from most snoop requests.

As shown in Figure 3, the extra bandwidth consumed by both configurations is small, 4.3 and 3.4 percent on average for the no-recent-miss and no-recent-snoop filters. The large reduction in replays leads to a reduction in average reorder-buffer utilization, which leads to an improvement in performance for those negatively affected applications in the replay-all configuration. For the single-processor results, value-based replay with the no-recent-snoop filter is only 1 percent slower than the baseline configuration on average. For the multiprocessor configuration, the difference is within the margin of error caused by workload nondeterminism.

This set of performance data uses a baseline machine configuration with a large, unified load/store queue. The primary motivation for value-based replay is to eliminate the large associative load queue structure from the processor, which does not scale as clock frequencies increase. We have also evaluated value-based replay in the context of machines with smaller clock-cycle-constrained load queues, and found that value-based replay offers a significant performance advantage, because CAM access latencies do not constrain the capacity of the load queue.

In this article, we explore a simple alternative to conventional associative load queues. We show that value-based replay causes a negligible impact on performance compared to a machine with an unconstrained load queue size. The value-based memory ordering mechanism relies on several heuristics to achieve high performance, significantly reducing the number of replays. Although we have primarily focused on value-based replay as a complexity-effective means for enforcing memory ordering, we believe that there is also potential for energy savings. Removal of the load

queue CAM should also reduce static power because of a reduction in area. It should also reduce dynamic power because of a reduction in address comparisons. In future work, we plan to more thoroughly evaluate value-based replay as a low-power alternative to conventional load queue designs. MICRO

Acknowledgments

This work was possible through an IBM Graduate Fellowship; generous equipment donations and financial support from IBM and Intel; and NSF grants CCR-0073440, CCR-0083126, EIA-0103670, and CCR-0133437.

References

1. S. Sethumadhavan et al., "Scalable Hardware Memory Disambiguation for High-ILP Processors," *Proc. 36th Int'l Symp. Microarchitecture* (Micro-36), ACM Press, 2003, pp. 399-410.
2. H. Akkary, R. Rajwar, and S.T. Srinivasan, "Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors," *Proc. 36th Int'l Symp. Microarchitecture* (Micro-36), IEEE CS Press, 2003, pp. 423-434.
3. Il Park, C.-L. Ooi, and T.N. Vijaykumar, "Reducing Design Complexity of the Load-Store Queue," *Proc. 36th Int'l Symp. Microarchitecture* (Micro-36), ACM Press, 2003, pp. 411-422.
4. D. Ponomarev, G. Kucuk, and K. Ghose, "Reducing Power Requirements of Instruction Scheduling Through Dynamic Allocation of Multiple Datapath Resources," *Proc. 34th Int'l Symp. Microarchitecture* (Micro-34), ACM Press, 2001, pp. 90-101.
5. K. Gharachorloo, A. Gupta, and J. Hennessy, "Two Techniques to Enhance the Performance of Memory Consistency Models," *Proc. Int'l Conf. Parallel Processing* (ICPP 91), CRC Press, 1991, pp. 355-364.
6. E. Altman et al., "Advances and Future Challenges in Binary Translation and Optimization," *Proc. IEEE*, vol. 89, no. 11, 2001, pp. 1710-1722.
7. T. Austin, "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design," *Proc. 32nd Int'l Symp. Microarchitecture* (Micro-32), IEEE CS Press, 1999, pp. 196-207.
8. A. Condon and A.J. Hu, "Automatable Verification of Sequential Consistency," *Proc. 13th Symp. Parallel Algorithms and Architectures* (SPAA 13), ACM Press, 2001, pp. 113-121.
9. A. Landin, E. Hagersten, and S. Haridi, "Race-Free Interconnection Networks and Multiprocessor Consistency," *Proc. 18th Int'l Symp. Computer Architecture* (ISCA 18), ACM Press, 1991, pp. 106-115.
10. A.R. Alameldeen and D.A. Wood, "Variability in Architectural Simulations of Multi-threaded Workloads," *Proc. 9th Int'l Symp. High-Performance Computer Architecture* (HPCA 9), IEEE Press, 2003, pp. 7-18.

Harold W. Cain is a research staff member at the IBM T.J. Watson Research Laboratory. His research interests include high-performance memory systems, memory consistency model implementation, and end-to-end system optimization. Cain has a BS in computer science from the College of William and Mary, and an MS and PhD in computer science from the University of Wisconsin-Madison. He is a member of the ACM and IEEE.

Mikko H. Lipasti is an assistant professor in the Department of Electrical and Computer Engineering at the University of Wisconsin, Madison. His research interests include the architecture and design of high-performance desktop and server computer systems. Lipasti has a BS in computer engineering from Valparaiso University, and an MS and a PhD in Electrical and Computer Engineering/from Carnegie Mellon University. He is a member of the IEEE.

This work was performed while Harold Cain was a student at the University of Wisconsin. Direct questions and comments about this article to him at IBM Watson Research Center, 1101 Kitchawan Road, Route 134, Yorktown Heights, NY 10598, tcain@us.ibm.com.

For further information on this or any other computing topic, visit our Digital Library at http://www.computer.org/publications/dlib.