Memory Ordering: A Value-Based Approach

Harold W. Cain
Computer Sciences Dept.
Univ. of Wisconsin-Madison
cain@cs.wisc.edu

Mikko H. Lipasti
Dept. of Elec. and Comp. Engr.
Univ. of Wisconsin-Madison
mikko@engr.wisc.edu

Abstract

Conventional out-of-order processors employ a multi-ported, fully-associative load queue to guarantee correct memory reference order both within a single thread of execution and across threads in a multiprocessor system. As improvements in process technology and pipelining lead to higher clock frequencies, scaling this complex structure to accommodate a larger number of in-flight loads becomes difficult if not impossible. Furthermore, each access to this complex structure consumes excessive amounts of energy. In this paper, we solve the associative load queue scalability problem by completely eliminating the associative load queue. Instead, data dependences and memory consistency constraints are enforced by simply re-executing load instructions in program order prior to retirement. Using heuristics to filter the set of loads that must be re-executed, we show that our replay-based mechanism enables a simple, scalable, and energy-efficient FIFO load queue design with no associative lookup functionality, while sacrificing only a negligible amount of performance and cache bandwidth.

1. Introduction

Computer architects have striven to build faster machines by improving the three terms of the fundamental performance equation: *CPU time = instructions/program x cycles/instruction x clock cycle time*. Unless a new instruction set is in development, improving the instructions/program term is largely beyond the architect's reach, and instead depends on the application developer or compiler writer. Hardware designers have therefore focused much energy on optimizing the latter two terms of the performance equation. Instructions-per-cycle (IPC) has increased by building out-of-order instruction windows that dynamically extract independent operations from the sequential instruction stream which are then executed in a dataflow fashion. Meanwhile, architects have also attempted to minimize clock cycle time through increased pipelining and careful logic and circuit design. Unfortunately, IPC and clock frequency are not independent terms. The hardware structures (issue queue, physical register file, load/store queues, etc.) used to find independent operations and correctly execute them out of program order are often constrained by clock cycle time. In order to decrease clock cycle time, the size of these conventional structures must usually decrease, also decreasing IPC. Conversely, IPC may be increased by increasing their size, but this also increases their access time and may degrade clock frequency.

There has been much recent research on mitigating this negative feedback loop by scaling structures in ways that are amenable to high clock frequencies without negatively affecting IPC. Much of this work has focused on the instruction issue queue, physical register file, and bypass paths, but very little has focused on the load queue or store queue [1][18][21]. Load and store queues are usually built using content-addressable memories that provide an address matching mechanism for enforcing the correct dependences among memory instructions. When a load instruction issues, the store queue CAM is searched for a matching address. When a store address is generated, the load queue CAM is searched for prior loads that incorrectly speculatively issued before the store instruction. Depending on the supported consistency model, the load queue may also be searched when every load issues [7], upon the arrival of an external invalidation request [10][25], or both [23]. As the instruction window grows, so does the number of in-flight loads and stores, resulting in each of these searches being delayed by 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/store queues that reduce the size of the CAM structure through a combination of filtering, caching, or segmentation. Sethumadhaven et al. employ bloom filtering of LSQ searches to reduce the frequency of accesses that must search the entire queue [21]. Bloom filters are also combined with an address matching predictor to filter the number of instructions that must reside in the LSQ, resulting in a smaller effective LSQ size. Akkary et al. explore a hierarchical store queue organization where a level-one store queue contains the most recent *n* stores, while prior stores are contained in a larger, slower level-two buffer [1]. A fast filtering mechanism reduces level-two lookups, resulting in a common-case load queue lookup that is the larger of the level-one store queue lookup latency and filter lookup latency. Their research does not address the scalability of the load queue. Park et al. explore the use of a store-set predictor to reduce store queue search bandwidth by filtering those loads that are predicted to be independent of prior stores. Load queue CAM size is reduced by removing loads that were not reordered with respect to other loads, and a variable-latency segmented LSQ is also explored [18]. Although each of these proposals offers a promising solution to the load or store queue scalability problem, their augmentative approach results in a faster search but also adds significant complexities to an already complex part of the machine. Ponomorev et al. explore the power-saving potential of a segmented load queue design where certain portions of the load/store queue are disabled when occupancy is low, but do not address the load queue scalability problem [19].

In this paper, we study a mechanism called *value-based replay* that completely eliminates the associative search functionality requirement from the load queue in an attempt to simplify the execution core. Instead, memory dependences are enforced by simply re-executing load operations in program order prior to commit, and comparing the new value to the value obtained when the load first executed. We refer to the original load execution as the *premature load* and the re-execution as the *replay load*. If the two values are the same, then the premature load correctly resolved its memory dependences. If the two values differ, then we know that a violation occurred either due to an incorrect reordering with respect to a prior store or a potential violation of the memory consistency model. Instructions that have already consumed the premature load's incorrect value must be squashed. By eliminating the associative search from the load queue, we remove one of the factors that limits the size of a processor's instruction window. Instead, loads can reside in a simple FIFO either separately or as part of the processor's reorder buffer. In Section 3, we describe in detail value-based replay's impact on the processor core implementation.

In order to mitigate the costs associated with replay (increased cache bandwidth and resource occupancy), we evaluate several heuristics that filter the set of loads that must be replayed. These filters reduce the percentage of loads that must be replayed exceptionally well (to 0.02 replay loads per committed instruction on average), and as a result, there is little degradation in performance when using load replay compared to a machine whose load queue includes a fully associative address CAM. Section 5 presents a thorough performance evaluation of value-based memory ordering using these filtering mechanisms.

Although the focus of this work is eliminating associative lookup hardware from the load queue, the store queue also requires an associative lookup that will suffer similar scalability problems in large-instruction window machines. We focus on the load queue for three primary reasons: 1) because loads occur more frequently than store instructions (loads and stores constitute 30% and 14%, respectively, of dynamic instructions for the workloads in this paper), the load queue in a balanced-resource machine should be larger than the store queue and therefore its scalability is more of a concern; 2) the store-to-load data forwarding facility implemented by the store queue is more critical to performance than the rare-error checking facility implemented by the load queue, and therefore the store queue's use of a fast search function is more appropriate; and 3) in some microarchitectures [7][23], the load queue is searched more frequently than the store queue, thus consuming more power and requiring additional read ports.

To summarize, this paper makes the following contributions:

- **Elimination of associative search logic from the load queue via value-based replay**: We demonstrate a value-based replay mechanism for enforcing uniprocessor and multiprocessor ordering constraints that eliminates the need for associative lookups in the load queue.
- **Replay-reduction heuristics:** We introduce several novel heuristics that reduce the cache bandwidth required by the load-replay mechanism to a negligible amount.
- **Consistency model checking:** we define the constraints necessary for implementing a back-end memory consistency checker. These constraints are also useful in the domain of other checking mechanisms such as DIVA [3]. Recent work has exposed a subtle interaction between memory consistency and value prediction that results in consistency violations under different forms of weak consistency [15]. Our value-based replay implementation may be used to detect such errors.

In the next section, we describe the microarchitecture of conventional associative load queues. Section 3 presents our alternative memory ordering scheme, value-based replay, including the description of our replay filtering heuristics. Our experimental methodology is described in Section 4, followed by a detailed performance study of the value-based replay mechanism relative to an aggressive conventional load queue design.

2. Associative Load Queue Design

We believe that an inordinate amount of complexity in an out-of-order microprocessor's design stems from the load queue, mainly due to an associative search function whose primary function is to detect rare errors. In this section, we examine the source of this complexity through an

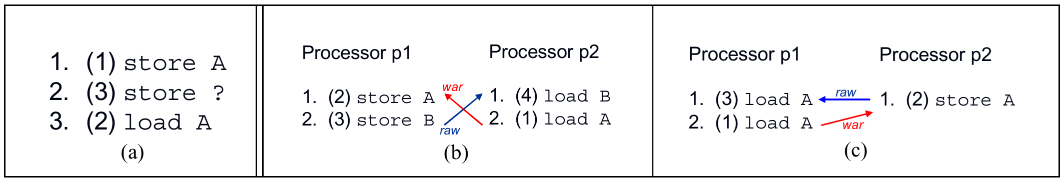

Figure 1. Correctly supporting out-of-order loads: Examples (a) uniprocessor RAW hazard, (b) multiprocessor violation of sequential consistency (c) multiprocessor violation of coherence

overview of the functional requirements of the load queue structures, and a description of their logical and physical design.

2.1. Functional Requirements and Logical Design

The correctness requirements enforced by the load queue are two-fold: loads that are speculatively reordered with respect to a prior store that has an unresolved address must be checked for correctness, and violations of the multiprocessor memory consistency model caused by incorrect reorderings must be disallowed. Figure 2(a) contains a code segment illustrating a potential violation of a uniprocessor RAW hazard. Each operation is labeled by its program order and by issue order (in parentheses). In this example, the load instruction speculatively issues before the previous store's address has been computed. Conventional memory ordering implementations enforce correctness by associatively searching the load queue each time a store address is computed. If the queue contains an already-issued prior load whose address overlaps the store, the load is squashed and re-executed. In this example, if the second store overlaps address A, the scan of the load queue will result in a match and the load of A will be squashed.

Some microarchitectures delay issuing a load instruction until all prior store addresses are known, altogether avoiding the need to detect RAW dependence violations. Unfortunately this solution is not sufficient to prevent violations of the memory consistency model, which can occur if any memory operations are reordered. Forcing all memory operations to execute in program order is too restrictive, so associative lookup hardware is still necessary even if loads are delayed until all prior store addresses are known.

In terms of enforcing the memory consistency model, load queue implementations can be categorized into two basic types: those in which external invalidation requests search the load queue, and those in which the queue is not searched. We refer to these types as *snooping load queues* and *insulated load queues*. In a processor with a snooping load queue, originally described by Gharachorloo et al., the memory system forwards external write requests (i.e. invalidate messages from other processors or I/O devices) to the load queue, which searches for already-issued loads whose addresses match the invalidation address [9], squashing any overlapping load. If inclusion is enforced between the load queue and any cache, replacements from that cache 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 loads that may have violated the consistency model. The exact load queue implementation will depend on the memory consistency model supported by the processor, although either of these two types may be used to support any consistency model, as described shortly.

Figure 1(b) illustrates a multiprocessor code segment where processor p2 has reordered two loads to different memory locations that are both written by processor p1 in the interim. In a sequentially consistent system, this execution is illegal because all of the operations cannot be placed in a total order. A snooping load queue detects this error by comparing the invalidation address corresponding to p1's store A to the load queue addresses, and squashing any instructions that have already issued to address A, in this case p2's second load. An insulated load queue prevents this error by observing at load B's completion that the load A instruction has already completed, potentially violating consistency, and the load A is subsequently squashed. Loads at the head of the load queue are inherently correct with respect to the memory consistency model, and are therefore never squashed due to external invalidation. Avoiding these squashes is important in order to ensure forward progress.

Processors that support strict consistency models such as sequential consistency and processor consistency do not usually use insulated load queues, due to the large number of operations that must be ordered with respect to one another (i.e. all loads). Insulated load queues are more prevalent in weaker consistency models, where there are few consistency constraints ordering instructions. For example, in weak ordering, only those operations that are separated by a memory barrier instruction, or those operations that read or write the same address, must be ordered. The Alpha 21264 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 [7]. Using the example in Figure 1(c), if processor 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 buffer

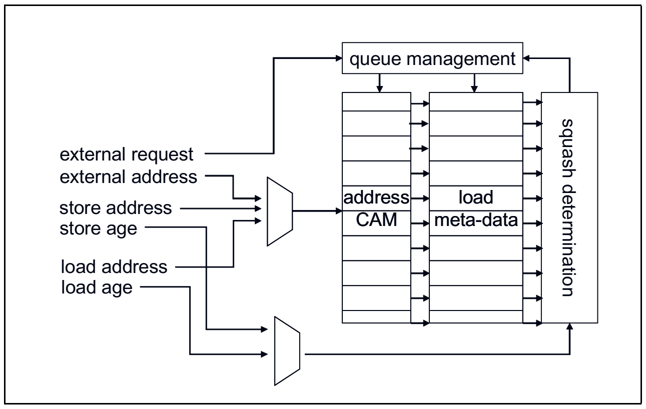

Figure 2. A simplified hybrid load queue

enforces this requirement by searching the load queue when each load issues and squashing any subsequent load to the same address that has already issued. Snooping load queues in sequentially consistent systems are simpler in this respect, because it is not necessary for load instructions to search the load queue, however this additional simplicity is offset by requiring support for external invalidation searches. In order 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 for prior loads to the same address, however only those that have been marked by a snoop hit must be squashed [23].

Obviously, both the snooping and insulated load queue implementations are conservative in terms of enforcing correctness. Due to false sharing and silent stores [14], it is not absolutely necessary to squash a load simply because its address matches the address of an external invalidation. The premature load may have actually read the correct value. Likewise, due to store value locality, when a store's address is computed, all subsequent loads to the same address that have already executed do not necessarily need to be squashed, many may have actually observed the correct value. These observations expose one benefit of the value-based ordering scheme, which has been exploited by other store-set predictor designs: a subset of squashes that occur in conventional designs are eliminated [17][26]. We quantify the frequency of unnecessary squashes in Section 5.

2.2. Physical Design

Load queues are usually implemented using two main data structures: a RAM structure containing a set of entries organized as a circular queue, and an associated CAM used to search for queue entries with a matching address. The RAM contains meta-data pertaining to each dynamic load (e.g. PC, destination register id, etc.), and is indexed by instruction age (assigned in-order by the front-end of the

Table 1: Load queue attributes for current dynamically scheduled processors

Processor	Est. # read ports	Est. # write ports
Compaq Alpha 21364 (32-entry load queue, max 2 load or store agens per cycle)	2 (1 per load/store issued/cycle)	2 (1 per load issued/cycle)
HAL SPARC64 V (size unknown, max 2 loads and 2 store agens per cycle)	3 (2 for stores, 1 for external invalidations)	2
IBM Power 4 (32-entry load queue, max 2 load or store agens 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 agen per cycle)	2 (1 for stores, 1 for external invalidations)	2

pipeline). Figure 2 illustrates a simplified hybrid load queue with a lookup initiated by each load or store address generation (agen) and external invalidation. For address generation lookups, an associated age input is used by the squash logic to distinguish those loads that follow the load or store. The latency of searching the load queue CAM is a function of its size and the number of read/write ports. Write port size is determined by the processor's load issue width; 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. A summary of their size in current generation processors with separate load queues (as opposed to integrated load/store queues) and an estimation of their read/write port requirements is found in Table 1. Typical current-generation dynamically scheduled processors use load queues with sizes in the range of 32-48 entries, and allow some combination of two loads or stores to be issued per cycle, resulting in a queue with two or three read ports and two write ports.

Using Cacti v. 3.2 [22], we estimate the access latency and energy per access for several CAM configurations in a 0.09 micron technology, varying the number of entries and the number of read/write ports, as shown in Table 2. Although this data may not represent a lower bound on each configuration's access latency or energy (human engineers can be surprisingly crafty), we expect the trends to be accurate. The energy expended by each load queue search increases linearly with the number of entries, and the latency increases logarithmically. Increasing load queue bandwidth through multiporting also penalizes these terms:

Table 2: Associative Load queue search latency (nanoseconds), energy (nanojoules)

entries	Read/Write Ports (ns, nJ)			
	2/2	3/2	4/4	6/6
16	0.6 ns, 0.03 nJ	0.68 ns, 0.04 nJ	0.72 ns, 0.07 nJ	0.79 ns, 0.12 nJ
32	0.75 ns, 0.05 nJ	0.77 ns, 0.06 nJ	0.85 ns, 0.12 nJ	0.94 ns, 0.20 nJ
64	0.78 ns, 0.12 nJ	0.80 ns, 0.15 nJ	0.87 ns, 0.27 nJ	0.97 ns, 0.45 nJ
128	0.78 ns, 0.22 nJ	0.80 ns, 0.28 nJ	0.88 ns, 0.50 nJ	0.97 ns, 0.85 nJ
256	0.97 ns, 0.37 nJ	1.01 ns, 0.48 nJ	1.13 ns, 0.87 nJ	1.28 ns, 1.51 nJ
512	1.00 ns, 0.80 nJ	1.04 ns, 1.03 nJ	1.16 ns, 1.87 nJ	1.32 ns, 3.22 nJ

doubling the number of ports more than doubles the energy expended per access, and increases latency by approximately 15%. Based on these measurements, it is clear that neither the size nor bandwidth of conventional load queues scale well, in terms of latency or energy. These scaling problems will motivate significant design changes in future machines that attempt to exploit higher ILP through increased issue width or load queue size. In the next section, we present one alternative to associative load queues, which eliminates this CAM overhead.

3. Value-based Memory Ordering

The driving principle behind our design is to shift complexity from the timing critical components of the pipeline (scheduler/functional units/bypass paths) to the back-end of the pipeline. During a load's premature execution, the load is performed identically to how it would perform in a conventional machine: at issue, the store queue is searched for a matching address, if none is found and a dependence predictor indicates that there will not be a conflict, the load proceeds, otherwise it is stalled. After issue, there is no need for loads or stores to search the load queue for incorrectly reordered loads (likewise for external invalidations). Instead, we shift complexity to the rear of the pipeline, where loads are re-executed and their results checked against the premature load result. To support this load replay mechanism, two pipeline stages have been added at the back-end of the pipeline preceding the commit stage, labeled replay and compare and shown in Figure 3. For simplicity, all instructions flow through the replay and compare stages, with action only being taken for load instructions.

During the replay stage, certain load instructions access the level-one data cache a second time. We do not support forwarding from prior stores to replay loads, so load instructions that are replayed stall the pipeline until all prior stores have written their data to the cache. Because

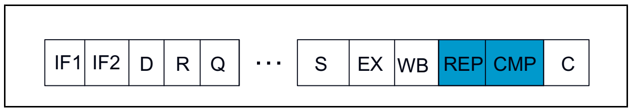

Figure 3. Pipeline diagram, replay stages highlighted

each replay load was also executed prematurely, this replay is less costly in terms of latency and power consumption than its corresponding premature load. For example, the replay access can reuse the effective address calculated during the premature load's execution, and in systems with a physically indexed cache the TLB need not be accessed a second time. In the absence of rare events such as an intervening cast-out or external invalidate to the referenced cache block between the premature load and replay load, the replay load will always be a cache hit, resulting in a low-latency replay operation. Because stores must perform their cache access at commit, the tail of the pipeline already contains datapath for store access. For the purposes of this work, we assume that this cache port may also be used for loads during the replay stage, which compete with stores in the commit stage for access to the cache port, priority being given to stores.

During the compare stage, the replay load value is compared to the premature load value. If the values match, the premature load was correct and the instruction proceeds to the commit stage where it is subsequently retired. If the values differ, the premature load's speculative execution is deemed incorrect, and a recovery mechanism is invoked. The use of a selective recovery mechanism is most likely precluded due to the variable latency and large distance between the processor's scheduling stage and replay stage, so we assume that a heavy-weight machine squash mechanism is used that squashes and re-executes all subsequent instructions, dependent and independent.

Because the replay mechanism enforces correctness, the associative load queue is replaced with a simple FIFO buffer that contains the premature load's address and data (used during the replay and compare stages), in addition to the usual meta-data stored in the load queue. To ensure a correct execution, however, care must be taken in designing the replay stage. The following three constraints guarantee any ordering violations are caught:

1) All prior stores must have committed their data to the L1 cache. This requirement ensures that RAW dependences are correctly satisfied for all loads. As a side-effect, it also eliminates the need for arbitration between the replay mechanism and store queue for access to the shared cache port; if there are prior uncommitted loads in the pipeline, the store queue will not retire stores to the cache (due to typical handling of precise exceptions), and conversely, when there are prior uncommitted stores in the pipeline, the replay stage will not attempt to issue loads.

2) All loads must be replayed in program order. To enforce consistency constraints, the local processor must not observe the writes of other processors out of their program order, which could happen if replayed loads are reordered. For the machine configuration used in this paper, we find that limiting replay to one instruction per cycle provides adequate replay bandwidth. Consequently, all loads are in-order because only one may be issued per clock cycle. In very aggressive machines, multiple load replays per cycle may be necessary. If all the replays performed in a single cycle are cache hits, their execution will appear atomic to other processors, and the writes of other processors will be observed in the correct order. In cases where a replay load causes a cache miss, correctness is ensured by forcing subsequent loads to replay after the cache miss is resolved.

3) A dynamic load instruction that causes a replay squash should not be replayed a second time after squash-recovery. This rule is necessary to ensure forward progress in pathological cases where contention for a piece of shared data can persistently cause a premature load and replay load to return different values.

One drawback to the value-based replay mechanism is its inability to correlate a misspeculated load dependency with the store on which it depends. In a conventional design, store addresses search the load queue as they are computed to find dependent loads that erroneously issued prematurely. When a match is found, there is no ambiguity concerning the identity of the store on which the load depends. However, when a load incurs a memory ordering violation in the value-based scheme, it is unclear which store caused the misspeculation, and therefore training a dependence predictor is not possible if the predictor requires knowledge of the specific store instruction that caused the replay (e.g. a store set predictor [6]).

Consequently, when evaluating value-based replay, we use a simpler predictor functionally equivalent to the dependence predictor used in the Alpha 21264, where a PC-indexed table maintains a single bit indicating whether the load at that PC has been a victim of a previous dependence misprediction [7]. If this bit is set, corresponding loads are prevented from issuing until the addresses of all prior stores are computed. For fairness, our evaluation of value-based replay in Section 5.1 includes a performance comparison to a baseline machine incorporating a store-set predictor.

Naively, all loads (except those satisfying rule 3) should be replayed to guarantee a correct execution. Unfortunately there are two primary costs associated with replaying loads, which we would like to avoid: 1) load replay can become a performance bottleneck given insufficient cache bandwidth for replays or due to the additional resource occupancy and 2) each replayed load causes an extra cache access and word-sized compare operation, consuming

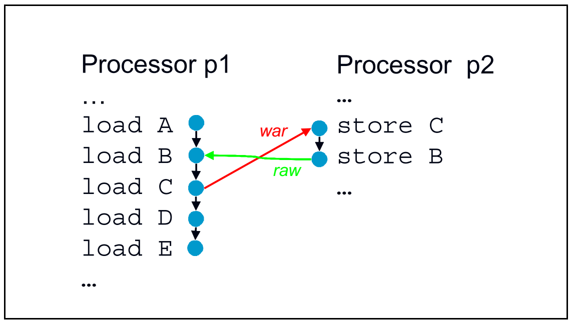

Figure 4. Constraint Graph Example

energy. In order to mitigate these penalties, we have investigated methods of eliminating the replay operation for certain load instructions. In the next two subsections, we define four filtering heuristics that are used to filter the set of loads that must be replayed while ensuring a 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. The first three filtering heuristics eliminate load replays while ensuring the execution's correctness with respect to memory consistency constraints. The final replay heuristic filters replays while preserving uniprocessor RAW dependences.

3.1. 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 of a multithreaded execution called a constraint graph [8][13]. The constraint graph is a directed graph consisting of a set of nodes representing dynamic instructions in a multithreaded execution, connected by edges that dictate constraints on the correct ordering of those instructions. For the purposes of this work, we assume a sequentially consistent system, where there are four edge types: program order edges that order all memory operations executed by a single processor, and the standard RAW, WAR, and WAW dependence edges that order all memory operations that read or write the same memory location. The key idea behind the constraint graph that makes it a powerful tool for reasoning about parallel executions is that it can be used to test the correctness of an execution by simply testing the graph for a cycle. The presence of a cycle indicates that there is not a total order of instructions, thereby violating sequential consistency. Figure 4 provides an example constraint graph for a small multithreaded execution, where processor p1 incorrectly reads the original value of C.

The following three replay filters detect those load operations that should be replayed to ensure correctness.

The first two are based on the observation that any cycle in the constraint graph must include dependence edges that connect instructions executed by two different processors. If an instruction is not reordered 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 (i.e. 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 window. Consequently, we can infer that no cycle can exist, and therefore there is no need to replay loads to check consistency constraints. Using the example from Figure 4, the load B would incur a cache miss in an invalidation-based coherence protocol, requiring p1 to fetch the block from p2's cache. When the block returns, p1 would need to replay any load instructions currently in its instruction window. This filter may be implemented as follows: each time a new cache block enters a processor's local cache, the cache unit asserts a signal monitored by the replay stage. When this signal is asserted, a "recent miss/need-replay" flag is set true and an age register is assigned the age index of the most-recently fetched load instruction in the instruction window. During each cycle that the flag is set to true, load instructions in the replay stage are forced to replay. After the flagged load instruction replays, if the age register still contains its age index, the flag is reset to zero.

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 can be detected by monitoring the occurrence of external write requests. If no blocks are written by other processors 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. Reorderings across outgoing WAW and RAW edges are prevented by the in-order commitment of store data to the cache. When the no-recent-snoop filter is used, loads are only replayed if they were in the out-of-order instruction window at the time an external invalidation (to any address) was observed by the core. In terms of implementation, a mechanism similar to the no-recent-miss filter can be used. This heuristic will perform best in systems that utilize inclusive cache hierarchies, which filter the stream of invalidates observed by the processor. Because fewer invalidates reach the processor, fewer loads will need to be replayed, although care must be taken to ensure that visibility of external invalidates is not lost due to castouts.

No-Reorder Filter: The no-reorder filter is based on the observation that the processor often executes memory operations in program order. If so, the instructions must be correct with respect to the consistency model, therefore there is no need to replay any load. We can detect operations that were originally executed in-order using the instruction scheduler, by marking loads that issue while there are prior incomplete loads or stores.

3.2. Filtering Replays While Enforcing Uniprocessor RAW Dependences

In order to minimize the number of replays necessary to enforce uniprocessor RAW dependences, we use the observation 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 did not bypass any stores with unresolved addresses when issued prematurely. These loads are identified and marked at issue time, when the store queue is searched for conflicting writes from which to forward data. A similar filter was used by Park et al. to reduce the number of load instructions that must be inserted into the load queue [18].

3.3. The Interaction of Filters

Of the four filters described above, only the no-reorder filter can be used in isolation; each of the other three are too aggressive. The no-recent-snoop and no-recent-miss filters eliminate all replays other than those that can be used to infer the correctness of memory consistency, at the risk of breaking uniprocessor dependences. Likewise, the no-unresolved-store filter eliminates all replays except those necessary to preserve uniprocessor RAW dependences, at the risk of violating the memory consistency model.

Consequently, the no-unresolved-store filter should be paired with either the no-recent-snoop or no-recent-miss filters to ensure correctness. If the no-unresolved-store filter indicates that a load should be replayed, it is replayed irrespective of the consistency filter output. Likewise, if the consistency filter indicates that a load should be replayed, it is replayed irrespective of the no-unresolved-store filter. For further improvement, the no-recent-snoop filter and no-recent-miss filter can be used simultaneously, however we find that these filters work well enough in isolation that we do not explore this option. In the next subsection, we evaluate the value-based replay mechanism using these filters.

4. Experimental Methodology

The experimental data presented in this paper was collected using PHARMsim [4], an out-of-order superscalar

Table 3: Baseline Machine Configuration

Out-of-order execution	5.0 GHZ, 15-stage 8-wide pipeline, 256 entry reorder buffer, 128 entry load/store queue, 32 entry issue queue, store-set predictor with 4k entry SSIT and 128 entry LFST (baseline only), 4k entry simple Alpha-style dependence predictor [7] (replay-based only).
Functional Units (latency)	8 integer ALUs (1), 3 integer MULT/DIV (3/12), 4 floating point ALUs (4), 4 floating point MULT/DIV (4, 4), 4 L1D load ports in OoO window, 1 commit stage L1D load/store port
Front-end	fetch stops at first taken branch in cycle, combined bimodal (16k entry)/gshare (16k entry) with selector (16k entry) branch prediction, 64 entry RAS, 8k entry 4-way BTB
Memory system (latency)	32k direct-mapped IL1 (1), 32k direct-mapped DL1 (1), 256k 8-way DL2 (7), 256k 8-way IL2 (7), Unified 8MB 8-way L3 (15), 64 byte cache lines, Memory (400 cycles/100 ns best-case latency, 10 GB/S bandwidth), Stride-based prefetcher modeled after Power4

processor model integrated into the SimOS-PPC full system simulator [11][20], which simulates PowerPC-based systems (uniprocessor and multiprocessor) running the AIX 4.3.1 operating system. The PHARMsim timing model is quite detailed, including full support for all user and system-level instructions in the PowerPC ISA, memory barrier semantics, asynchronous interrupts, I/O devices (disk, console, and network adapter) including cache-coherent memory-mapped I/O, and all aspects of address translation including hardware page table walks and page faults.

We evaluate value-based replay in the context of both a uniprocessor system and a 16-processor shared-memory multiprocessor. Details of the machine configuration used for the uniprocessor experiments are found in Table 3. We use a fairly aggressive machine model in order to demonstrate the value-based replay mechanism's ability to perform ordering checks without hindering performance.

For the multiprocessor performance data, we assume an identical machine configuration, augmented with a Sun Gigaplane-XB-like interconnection network for communication among processors and memory [5] that incurs an extra 32 cycle latency penalty for address messages and 20 cycle latency penalty for data messages. We assume a point-to-point data network in which bandwidth scales with the number of processors.

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). The three floating-point benchmarks were selected due to their high reorder buffer utilization [16], a trait with which the

Table 4: Other Benchmark Descriptions

Benchmark	Comments
barnes	SPLASH-2 N-body simulation (8K particles)
ocean	SPLASH-2 Ocean simulation (514x514)
radiosity	SPLASH-2 light interaction application (-room -ae 5000.0 -en -0.050 -bf 0.10)
raytrace	SPLASH-2 raytracing application (car)
SPECjbb2000	Server-side Java benchmark (IBM jdk 1.1.8 w/ JIT, 400 operations)
SPECweb99	Zeus Web Server 3.3.7 servicing 300 HTTP requests
TPC-B	Transaction Processing Council's Original OLTP Benchmark Benchmark (IBM DB2 v 6.1)
TPC-H	Transaction Processing Council's Decision Support Benchmark (IBM DB2 v 6.1, running query 12 on a 512 MB database)

value-based replay mechanism may negatively interact. For multiprocessor experiments, we use the SPLASH-2 parallel benchmark suite [24], SPECweb99, SPECjbb2000, and TPC-H. The SPEC integer and SPLASH-2 benchmarks were compiled with the IBM xlc optimizing C compiler, except for the C++ benchmark eon, which was compiled using g++ version 2.95.2. The SPECFP benchmarks were compiled using the IBM xlf optimizing Fortran compiler. The SPECCPU benchmarks were run to completion using the MinneSpec reduced input sets [12]. Setup parameters for the other benchmarks are specified in Table 4. 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 errors bars signifying 95% statistical confidence [2].

5. Experimental Evaluation

The experimental evaluation of value-based replay is divided into three sections. In the first, we present a detailed comparison of the value-based replay mechanism to a baseline machine containing a large unified load/store queue subject to no size-limiting physical constraints. In the second subsection we present a comparison of the best configuration evaluated in Section 5.1 (no-recent-snoop/no-unresolved-store) to a baseline machine whose conventional load queue size is constrained by clock cycle time. The third subsection presents a simple power model describing the parameters under which an implementation using value-based replay is more power-efficient than a traditional load queue implementation.

5.1. Performance Comparison

Figure 5 presents the performance of value-based

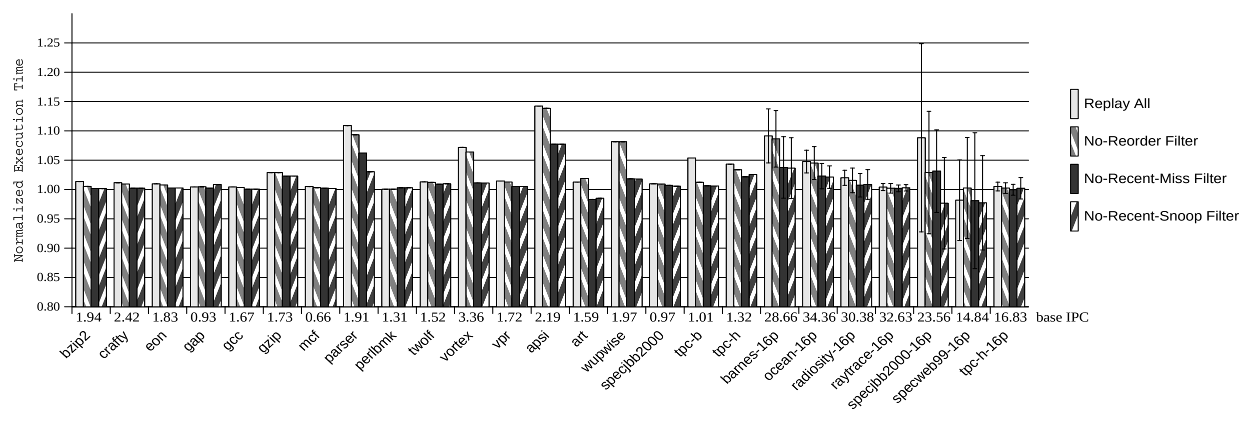

Figure 5. Value-based replay performance, relative to baseline

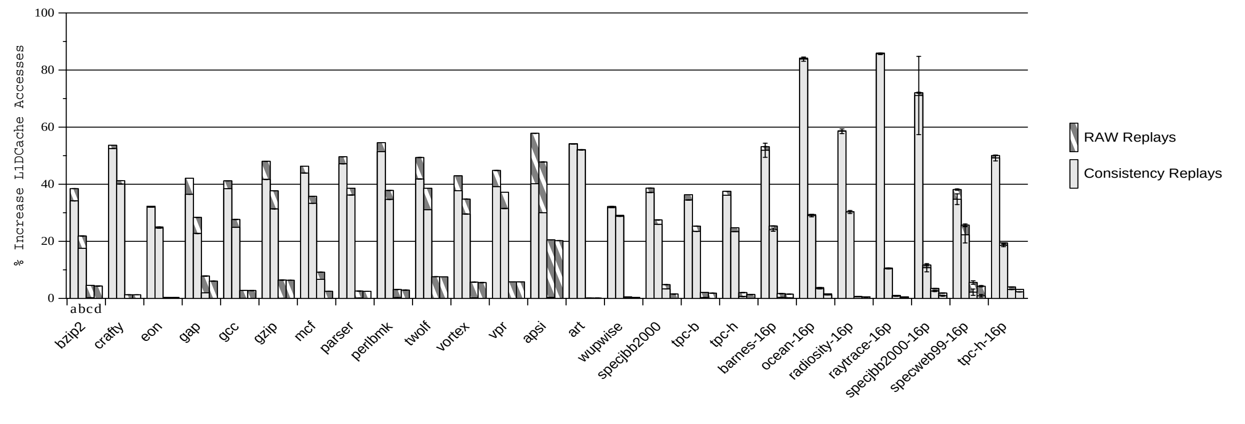

Figure 6. Increased data cache bandwidth due to replay: (a) replay all, (b) no-reorder filter, (c) no-recent-miss filter, (d) no-recent-snoop filter

replay 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. This data is normalized to the baseline IPC shown at the bottom of each set of bars.

The value-based replay mechanism is very competitive to the baseline machine despite the use of a simpler dependence predictor. Without the use of any filtering mechanism, value-based replay incurs a performance penalty of only 3% on average. The primary cause of this performance degradation is an increase in reorder buffer occupancy. Figure 6 shows the increase in L1 data cache references for each of the value-based configurations. Each bar is broken into two segments: replays that are necessary because the load issued before a prior store's address was resolved, and replays that were performed irrespective of uniprocessor constraints. Without filtering any replays, accesses to the L1 data cache increase by 49% on average, ranging from 32% to 87% depending on the percentage of cache accesses that are caused by wrong-path speculative instructions and the fraction of accesses that are stores. This machine configuration is limited to a single replay per cycle due to the single back-end load/store port, which leads to an increase in average reorder buffer utilization due to cache port contention (most dramatically in apsi and vortex, as shown in Figure 7). This contention results in performance degradation due to an increase in reorder buffer occupancy and subsequent reorder buffer allocation stalls. Although this performance degradation is small on average, there are a few applications where performance loss is significant.

When the no-reorder filter is enabled, the performance of value-based replay 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%, indicating that most loads do execute out-of-order with respect to at least one other load or store.

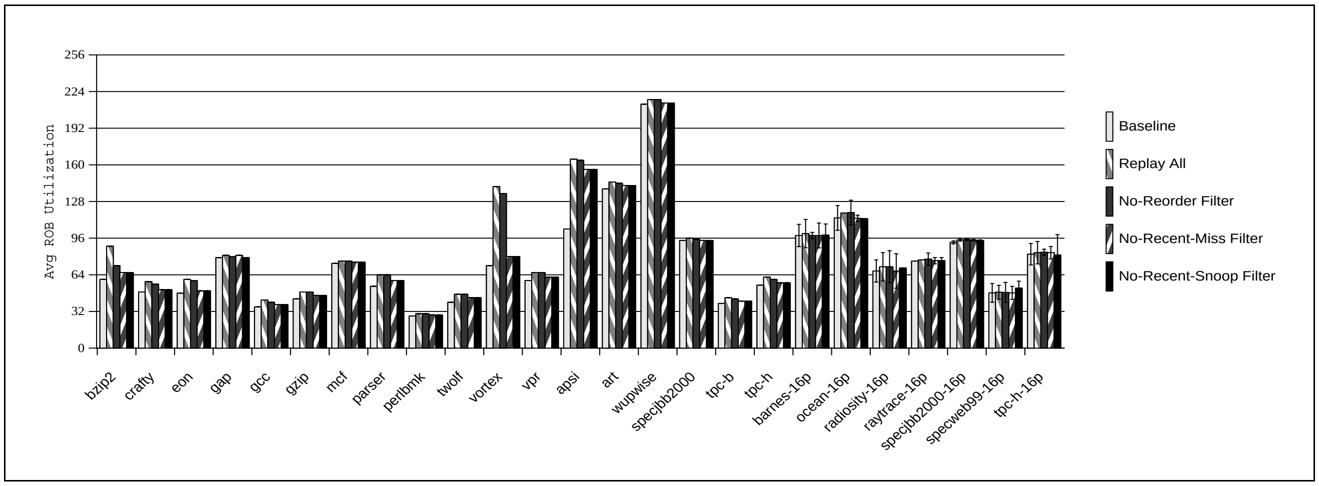

Figure 7. Average reorder buffer utilization

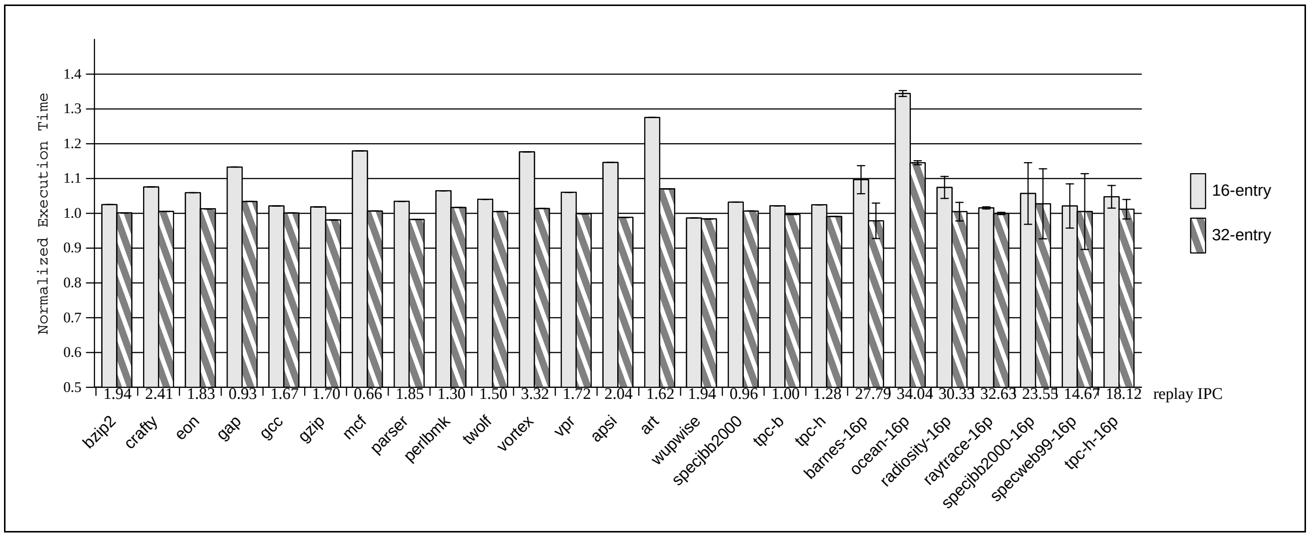

Figure 8. Performance of constrained load queue sizes, relative to value-based replay with no-recent-snoop/no-unresolved-store filters

The no-recent-snoop and no-recent-miss filters, when used in conjunction with the no-unresolved-store filter, solve this problem. For the single-processor machine configurations, there are no snoop requests observed by the processor other than coherent I/O operations issued by the DMA controller, which 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 6, the extra bandwidth consumed by both configurations is small, 4.3% and 3.4% on average for the no-recent-miss and no-recent-snoop filters respectively. The large reduction in replays leads to a reduction in average reorder buffer utilization (shown in Figure 7), which leads to an improvement in performance for those applications that were negatively affected in the replay-all configuration. For the single processor results, value-based replay with the no-recent-snoop filter is only 1% slower than the baseline configuration on average. For the multiprocessor configuration, the difference is within the margin of error caused by workload non-determinism.

The benchmark suffering the largest performance degradation (apsi), is actually less affected by resource occupancy than by the switch from a store-set predictor to a simple dependence predictor. Neither predictor incurs many memory order violations, however for this benchmark the simple predictor more frequently stalls loads due to incorrectly identified dependences, ultimately decreasing IPC. The reverse is true for the benchmark art, where the baseline machine's store-set predictor stalls a significant fraction of loads unnecessarily, resulting in a performance improvement in the value-based replay configurations. We attempted to exacerbate the negative effects of the simple dependence predictor relative to the store-set predictor by repeating these experiments using a larger 256-entry issue queue, but found that the results do not differ materially for this machine configuration.

One advantage of the value-based replay mechanism is its ability to filter dependence misspeculation squashes if the misspeculated load happens to return the value of the conflicting store. Due to this value locality, we find that the value based mechanism on average eliminates 59% of the dependence misspeculation squashes caused by RAW uniprocessor violations, because the replay load happens to receive the same value as the premature load, even though the premature load should have had its data forwarded by an intermediate store. However, the frequency of squashes for these applications is so low (on the order of 1 per 100 million instructions), this reduction has little effect on overall performance.

The results are similar for consistency violations. In the multiprocessor machine configuration, value-based replay is extremely successful at avoiding consistency squashes, eliminating 95% on average. However we once again find that such machine squashes occur so infrequently (4 per 10,000 instructions in the most frequent case, SPECjbb2000) their impact on performance is insignificant. Should consistency squashes be a problem with larger machine configurations or applications where there is a greater level of contention for shared data structures, value-based replay is a good means of improvement.

5.2. Constrained Load Queue Size

The previous 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. Figure 8 presents a performance comparison of the best value-based replay configuration (the no-recent-snoop and no-unresolved store filters) to a baseline machine configuration that uses a separate smaller load queue, for two different sizes, 16-entries and 32-entries. A 32-entry load queue is representative of current generation load queue sizes, and makes a fairly even performance comparison to the value-based replay configuration. On average, the value-based replay configuration is 1.0% faster, with art and ocean being significantly faster due to their sensitivity to load queue size (7% and 15% respectively). In future technology generations, a 32-entry load queue CAM lookup will not fit into a single clock cycle. When the load queue size is constrained to 16 entries, value-based replay offers a significant advantage, at most 34% and averaging 8% performance improvement.

5.3. A Simple Power Model

Of course, performance is not the only metric, we would also like to minimize the amount of power consumed by this memory ordering scheme. Extrapolating any quantitative power estimates would be unreliable without using a detailed underlying power model. Instead, one can get a rough estimate of the difference in dynamic energy using a simple model:

$$\Delta Energy = ((E_{cacheaccess} + E_{wordcomparison}) \times replays) \\ - (E_{ldqsearch} \times ldqsearches) + overhead_{replay}$$

The primary energy cost for the value-based replay mechanism is the energy consumed by replay cache accesses and word-sized comparison operations. This cost is multiplied by the number of replays, which we have empirically determined to be relatively small. In the equation above, $overhead_{replay}$ includes the energy cost of the two extra pipeline latches and replay filtering mechanism. Because the number of replays is quite small (on average 0.02 replay loads per committed instruction using the no-recent-snoop/no-unresolved-store filter configuration), if an implementation's load queue CAM energy expenditure per committed instruction is greater than 0.02 times the energy expenditure of a cache access and word-sized comparison, we expect the value-based replay scheme to yield a reduction in power consumption.

6. Conclusions

As transistor budgets increase, there is a great temptation to approach each new problem by augmenting an existing design with special purpose predictors, buffers, and other hardware widgets. Such hardware may solve the problem, but also increases the size and complexity of a design, creating an uphill battle for architects, designers, and verifiers trying to produce a competitive product. In this paper, 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 whose load queue size is unconstrained. When comparing the value-based replay implementation to processors whose load queues are constrained by clock cycle time, we find the potential for significant performance benefits, up to 34% and averaging 8% relative to a 16-entry load queue.

The value-based memory ordering mechanism relies on several heuristics to achieve high performance, which reduce the number of replays significantly. We believe these heuristics are more broadly applicable to other load/store queue designs and memory order checking mechanisms, and plan to explore their use in other settings. 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, as outlined in the last section. In future work, we plan to perform a more thorough evaluation of value-based replay as a low-power alternative to conventional load queue designs.

Acknowledgments

This work was made 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] H. Akkary, R. Rajwar, and S. T. Srinivasan. "Checkpoint processing and recovery: Towards scalable large instruction window processors." In *Proc. of the 36th Intl. Symp. on Microarchitecture*, December 2003.

[2] A. Alameldeen and D. Wood. "Variability in architectural simulations of multi-threaded workloads." In *Proc. of the Ninth Intl. Symp. on High Performance Computer Architecture*, February 2003.

[3] T. Austin. "DIVA: A reliable substrate for deep submicron microarchitecture design." In *Proc. of the 32nd Intl. Symp. on Microarchitecture*, pages 196–207, Haifa, Israel, November 1999.

[4] H. W. Cain, K. M. Lepak, B. A. Schwartz, and M. H. Lipasti. "Precise and accurate processor simulation." In *Proc. of the Workshop on Computer Architecture Evaluation using Commercial Workloads*, February 2002.

[5] A. Charlesworth, A. Phelps, R. Williams, and G. Gilbert. "Gigaplane-XB: extending the ultra enterprise family." In *Proceedings of Hot Interconnects V*, pages 97–112, August 1997.

[6] G. Z. Chrysos and J. S. Emer. "Memory dependence prediction using store sets." In *Proc. of the 25th Intl. Symp. on Computer architecture*, pages 142–153. IEEE Press, 1998.

[7] Compaq Computer Corporation, Shrewsbury, Massachusetts. *21264/EV68CB and 21264/EV68DC Hardware Reference Manual*, 1.1 edition, June 2001.

[8] A. Condon and A. J. Hu. "Automatable verification of sequential consistency." In *Proc. of the 13th Symp. on Parallel Algorithms and Architectures*, January 2001.

[9] K. Gharachorloo, A. Gupta, and J. Hennessy. "Two techniques to enhance the performance of memory consistency models." In *Proc. of the 1991 Intl. Conf. on Parallel Processing*, pages 355–364, August 1991.

[10] Intel Corporation. *Pentium Pro Family Developers Manual, Volume 3: Operating System Writers Manual*, Jan. 1996.

[11] T. Keller, A. Maynard, R. Simpson, and P. Bohrer. "Simos-ppc full system simulator." http://www.cs.utexas.edu/users/cart/simOS.

[12] A. KleinOsowski and D. J. Lilja. "Minnespec: A new SPEC benchmark workload for simulation-based computer architecture research." *Computer Architecure Letters*, 1, June 2002.

[13] A. Landin, E. Hagersten, and S. Haridi. "Race-free interconnection networks and multiprocessor consistency." In *Proc. of the 18th Intl. Symp. on Comp. Architecture*, 1991.

[14] K. M. Lepak and M. H. Lipasti. "On the value locality of store instructions." In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 182–191, Vancouver, BC, June 2000.

[15] M. M. K. Martin, D. J. Sorin, H. W. Cain, M. D. Hill, and M. H. Lipasti. "Correctly implementing value prediction in microprocessors that support multithreading or multiprocessing." In *Proc. of the 34th Intl. Symp. on Microarchitecture*, pages 328–337, December 2001.

[16] J. F. Martinez, J. Renau, M. C. Huang, M. Prvulovic, and J. Torrellas. "Cherry: checkpointed early resource recycling in out-of-order microprocessors." In *Proceedings of the 35th annual Intl. Symp. on Microarchitecture*, pages 3–14. November, 2002.

[17] S. Onder and R. Gupta. "Dynamic memory disambiguation in the presence of out-of-order store issuing." In *Proc. of the 32nd Intl. Symp. on Microarchitecture*, November 1999.

[18] I. Park, C.-L. Ooi, and T. N. Vijaykumar. "Reducing design complexity of the load-store queue." In *Proc. of the 36th Intl. Symp. on Microarchitecture*, December 2003.

[19] D. Ponomarev, G. Kucuk, and K. Ghose. "Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources." In *Proc. of the 34th Intl. Symp. on Microarchitecture*, December 2001.

[20] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta. "Complete computer simulation: the simos approach." *IEEE Parallel and Distributed Technology*, 3(4):34–43, 1995.

[21] 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 Intl. Symp. on Microarchitecture*, December 2003.

[22] P. Shivakumar and N. P. Jouppi. "Cacti 3.0: An integrated cache timing, power, and area model." Technical Report 2001/2, Compaq Western Research Lab Research Report, 2001.

[23] J. M. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. "POWER4 system microarchitecture." Technical white paper, IBM Server Group, October 2001.

[24] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. "The SPLASH2 programs: Characterization and methodological considerations." In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, June 1995.

[25] K. C. Yeager. "The MIPS R10000 superscalar microprocessor." *IEEE Micro*, 16(2):28–40, April 1996.

[26] A. Yoaz, R. Ronen, R. S. Chappell, and Y. Almog. "Silence is golden?" In *Work-in-progress workshop of the 7th International Symposium on High-Performance Computer Architecture*, January 2001.