Replacing Associative Load Queues: A Timing-Centric Approach

Fernando Castro, *Member*, *IEEE*, Regana Noor, Alok Garg, *Student Member*, *IEEE*, Daniel Chaver, Michael C. Huang, *Member*, *IEEE*, Luis Piñuel, *Member*, *IEEE*, Manuel Prieto, *Member*, *IEEE*, and Francisco Tirado, *Senior Member*, *IEEE*

Abstract—One of the main challenges of modern processor design is the implementation of a scalable and efficient mechanism to detect memory access order violations as a result of out-of-order execution. Traditional age-ordered associative load queues are complex, inefficient, and power hungry. In this paper, we introduce two new dependence checking schemes with different design tradeoffs, but both explicitly rely on timing information as a primary instrument to rule out dependence violation. Our timing-centric designs operate at a fraction of the energy cost of an associative LQ and achieve the same functionality with an insignificant performance impact on average. Studies with parallel benchmarks also show that they are equally effective and efficient in a chip-multiprocessor environment.

Index Terms—LSQ, memory disambiguation, energy efficiency.

1 Introduction

With high operating frequency, modern out-of-order processors often need to buffer a very large amount of instructions to be able to overlap useful processing with relatively long latencies associated with accesses to lower levels of the memory hierarchy. Processor features such as multithreading further increase the demand on the instruction buffering capability. However, increasing the number of in-flight instructions requires scaling up different microarchitectural structures, which has a significant impact on energy consumption, especially if the structure is accessed associatively.

One such example is the logic that enforces correct memory-based dependences, commonly referred to as the load-store queue (LSQ), and typically implemented as two separated queues: the load queue (LQ) and the store queue (SQ). Conventional implementations of these queues contain complete addresses and their entries are allocated in program order. To enable early execution of loads without compromising program correctness, memory instructions are tracked by the two queues and associative searches are used to find the correct producer or to detect dependence violations.

These associative search operations are a major concern for the scalability of these queues. Not only energy consumption increases with the size of the queue, the latency of accesses also worsens and may present complications in the logic design. As such, a range of implementations that avoid associative searches has been explored recently. The main observation behind these designs is that memory-based dependencies are very infrequent, and hence, through clever filtering or prediction, it is possible to reduce the number of associative accesses.

In this paper, we introduce an alternative *timing-centric* dependence violation detection strategy. The central observation is that memory instructions very often execute in such a manner that dependence violations can be ruled out solely based on the instruction execution timing. For these cases, conventional approach designed to capture all possible violations becomes an overkill and a source of energy waste as expensive associative searches are performed unconditionally. We show that a timing-centric strategy can be implemented with simple circuitry and with virtually no performance degradation compared to conventional design. In this paper, we discuss two realizations of this strategy. One performs violation detection early at the execution time, the other delays the action to the commit time. In both cases, the conventional associative LQ is replaced with a more efficient implementation.

The rest of this paper is organized as follows: Section 2 recaps the conventional design of the LQ. Section 3 motivates the timing-centric approach. Sections 4 and 5 present two alternative implementations. Section 6 presents experimental results and analyses. Section 7 discusses related work. Finally, Section 8 concludes.

2 Conventional Design

2.1 Memory Dependence

Out-of-order microprocessors typically allow early execution of loads for high performance, even if some older

- *F. Castro, D. Chaver, L. Piñuel, M. Prieto, and F. Tirado are with the ArTeCS Group, Departamento de Arquitectura de Computadores y Automática, Facultad de C.C. Físicas, Universidad Complutense de Madrid, Ciudad Universitaria, 28040 Madrid, Spain.*
  E-mail: fcastror@fis.ucm.es,
  {dani02, lpinuel, mpmatias, ptirado}@dacya.ucm.es.
- *R. Noor, A. Garg, and M.C. Huang are with the University of Rochester, 160 Trustee Rd, Rochester, NY 14627.*
  E-mail: {noor, garg, huang}@ece.rochester.edu.

Manuscript received 26 July 2007; revised 31 Dec. 2007; accepted 23 July 2008; published online 6 Aug. 2008.
Recommended for acceptance by M. Yousif.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2007-07-0386.
Digital Object Identifier no. 10.1109/TC.2008.146.

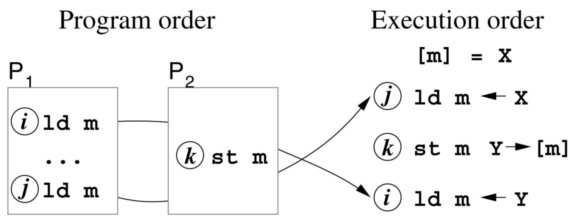


Fig. 1. Example of violation of WS.

stores have not yet been resolved.[1] Thus, when an earlier store does access the same memory location, the data returned by the earlier speculative load becomes incorrect and the processor must take a corrective action to ensure the sequential semantics. This dependence enforcement is achieved using age-ordered LQ and SQ.

When a load executes, in addition to the cache access, its address is checked with all older stores in the SQ. If a match is detected, the youngest store forwards the data to the load (load forwarding). Conversely, when a store executes, it must check the LQ looking for younger loads to the same address that have executed prematurely. When matches are found, the processor needs to reexecute (or replay) premature loads and their dependents. To simplify implementation, processors typically replay many more instructions (such as all instruction groups following the store [1]), as these premature loads are rare in general and sometimes extra logic is employed to further reduce their occurrence [2].

2.2 Cache Coherence

The LQ also serves the purpose of maintaining load-load ordering for coherence. A cache-coherent design requires *write serialization (WS)*: all writes to the same location have to appear in the same order to all processors. Even in a relaxed consistency model [3], [4], where a load from one processor need not have a defined order with a store from another processor, this following sequence of events (all to the same memory location) is illegal as it violates WS (as shown in Fig. 1): 1) load j issues, obtaining some data X; 2) an invalidation message then arrives due to a store from another processor; 3) finally, load i, older in program order than j, issues and obtains the new data Y. This effectively makes the store of Y appear earlier than the store of X. In this case, the common practice is to replay from load j to ensure that it gets newer data [1], [2].

Detecting this sequence is not trivial, however. This is because none of the individual ordering is illegal, not even the reordering of i and j. It is the combination that is not allowed. In [1], every invalidation will search the entire LQ to mark a bit for any matching loads. Every load will also search the entire LQ. If a matching younger entry is found and the aforementioned bit is set, the sequence has occurred and the corrective action is taken. As modern processors are almost all cache-coherent, at least with respect to DMA operations, WS becomes a standard feature even for uniprocessors.

2.3 Sequential Consistency

When a system implements sequential consistency (SC), it needs to present an image of a global ordering: all processors issue memory requests as if following some sequential ordering and the requests from each processor follow the program order. Presenting this image allows the programmer to argue about program behavior relatively easily. It does not necessarily limit the implementation to sequentially performing memory operations. Indeed, high-end microprocessors supporting SC allow load instructions to speculatively issue early and perform corrections when such speculation may violate the SC model. For instance, in MIPS R10000 [5], loads and stores have the appearance of being performed at the time of commit, but a load can issue early because so long as the memory location is not changed in between issue and commit, the result would be the same. Hence, if the location is changed—manifested as an invalidation received before the commit of the load—then the result from the speculatively issued load cannot be trusted. Thus, during an invalidation, the LQ is searched to find any loads that have been issued to the cache line. If so, the load and all subsequent instructions are basically squashed and reexecuted.

In summary, in a typical microarchitecture, the LQ is being searched very frequently, by all stores, loads, and external invalidation messages. As current processors are routinely designed to have a capacity of hundreds of in-flight instructions, the LQ also needs to have a fairly large size to support that capacity. Clearly, large associative queues with wide entries (full address) and multiple ports are undesirable in wide-issue, high-frequency designs. In addition to creating timing challenges, they consume significant energy.

3 A Timing-Centric Approach

An important inefficiency in conventional design is that it does not fully exploit the common case: actual order violations are rare; and ruling out a violation can often be done very inexpensively, with partial information: timing.

A memory ordering violation happens only when both temporal and spatial conditions are satisfied. For instance, only when a load executes before an older store and they overlap in address will there be a memory dependence violation. Although processors allow out-of-order issue of memory instructions, in typical programs, the execution order of memory instructions is often not wildly different from the program order. Thus, the timing condition for a violation is quite hard to establish. Therefore, observing the timing condition alone can be sufficient for ruling out violation. This point is easily illustrated by an extreme example: in a uniprocessor, if all memory instructions happened to be issued all in program order, the forwarding logic in the SQ already ensures memory dependence and without any knowledge about the addresses used, we can still be certain of the absence of any memory dependence violation.

1. A store has two operands—the address and the data—which can be separately handled [1]. The store is *resolved* when the address is ready. If only the address of a store is ready, the SQ cannot perform forwarding and instead will *reject* the consumer load to issue later [1]. We assume such an implementation in this paper.

Based on this observation, we propose a timing-centric approach to violation detection where we focus on memory instructions' ordering and rely on execution timing and the instruction's program order (or *age*) to rule out violation and eliminate the need for any further action in the common case. In contrast, a conventional LQ performs the (expensive) associative address comparisons unconditionally leading to energy waste.

Even though timing information is not always enough to lead to definitive conclusions by itself, it can still serve as a first-cut filtering mechanism, which allows a more efficient implementation to perform further inspection of the addresses information. In the following, we will show two different flavors of implementation. Both use very simple indexing tables for address inspection and no associative LQ is needed.

4 Issue-Time Memory Dependence Checking

The first timing-centric implementation we propose is similar to the conventional design in that the memory dependence violation checking is still performed at the issue time of a store instruction [6] and the goals of individual actions are all the same. The only difference is the circuits used to perform the actions.

4.1 Hash Table-Based Tracking

The key difference between our timing-centric design and the conventional design is that we *explicitly* assign and track the age of loads. Recall that determining whether a store-load replay is needed requires two pieces of information: address and age. Conventional design allocates an LQ entry for each load at dispatch time in program order thereby *implicitly* encoding the age information within the position of the entry.

By explicitly encoding and tracking the age, we no longer need to allocate an entry for every load at the early stage of dispatch. Instead, we use the simple, oft-used indexing table or a hash table. We refer to this table as the *load table*. Upon execution, each load will use the address to hash into the table and record its age. When a store executes, the address is also used to hash into the load table. If the age recorded in the entry is younger than that of the store, then the load and store are executed out of order with respect to each other. Since they map into the same entry, their addresses are conservatively assumed to be the same, and a replay is triggered to correct the (possible) dependence violation.

Clearly, multiple loads can hash into the same entry of the load table. We simply keep the age of the *youngest* load. This is because the *existence* of an order violation is all-important, whereas the identity of the load(s) involved is dispensable. Keeping the age of the youngest load is sufficient to detect the existence of order violation. When a replay is needed, we can simply replay from the instruction following the store in program order—rather than starting from the oldest load among all triggering loads. In fact, trying to identify the oldest load that needs to replay requires additional circuit complexity, which existing processors such as the IBM POWER 4 chose to avoid [1]. It, too, replays from the store onward.

4.1.1 Representing Age

In the conventional design, load and store ages are recorded largely implicitly—by their relative position in the associative queues. For timing-centric dependence checking to work, we need to *explicitly* track and compare instructions' age to determine if they have issued out of program order. As long as comparison is possible and there is no ambiguity, any age representation will work. In typical microprocessor implementations, ROB entry ID already serves as a convenient age representation[2] and is used by other microarchitectural structures and thus is an obvious choice in our design. Later in Section 4.3, we discuss a simple additional mechanism to make the representation more suitable for our tasks.

4.1.2 Ensuring Write Serialization

The guarantee of WS can be supported by the hash table implementation as well. For load instructions, we already need to perform a read of the hash table to determine whether the recorded age needs to be updated. Thus, checking of load-load replay is essentially for free in our design. In fact, we can simply replay when loads mapping to the same hash table entry execute out of order, similar to the way a load-load replay is triggered in Alpha 21264 [2]. However, we choose to make a simple addition to the circuitry that would reduce unnecessary false replays. Each entry has an "INV" bit to record invalidations. An invalidation will also hash into the load table and mark the INV bit of the entries corresponding to the invalidated addresses. When a load executes, we inspect the INV bit in the entry. We only replay when the bit is set and the age recorded in the table is younger than that of the current load (L_c).

Similar to the store-load replay, we cannot pinpoint the identity of the younger loads and have to replay from the instruction following L_c. Again, such a more conservative range of replay is already adopted in existing processors for circuit simplicity [1]. After the replay, we also set the entry's age to that of L_c, since all instructions younger than that have been squashed.

Finally, the invalidation mark needs to be cleared at some point. Specifically, when an invalidation arrives, the youngest in-flight load will be recorded by a pointer ($P_{youngest}$). If another invalidation arrives, the pointer will be pushed to point to the then youngest load. After the load identified by $P_{youngest}$ retires, all the INV bits will be flash-cleared. In the extremely unlikely case where invalidations are so frequent that clearing never happens, all INV bits will probably be set. Then, the circuit essentially falls back to ensuring all loads mapping the same location always happen in program order [2].

4.1.3 Supporting Sequential Consistency

SC can also be supported with the load table. Recall that in R10000 [5], if an invalidation overlaps with an in-flight load instruction that has issued, the instructions from the load onward will be squashed. Similarly, if an invalidation

2. Strictly speaking, ROB entry ID is a representation of "date of birth," not age in the conventional sense. So, the larger the number, the younger the age.

covers a load table entry with a valid age, we need to conservatively replay. Note that because the load table only records the age of the youngest load, we cannot identify the oldest load that potentially violates SC. Thus, we can only conservatively replay from the oldest in-flight load instruction.

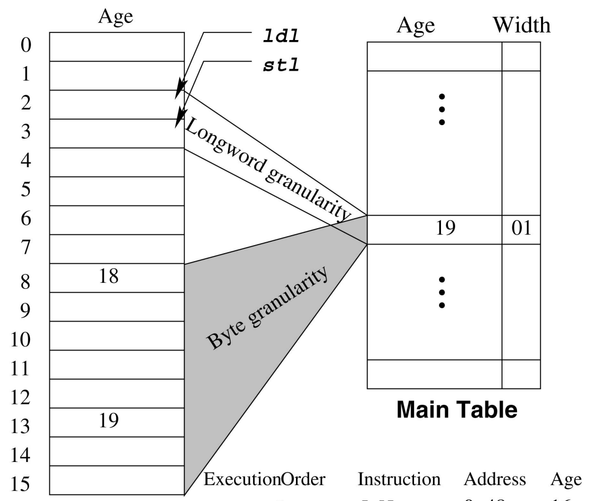
4.2 Handling Multiple Data Sizes

Unfortunately for memory order tracking, accesses come at different sizes: from 1 to 8 bytes or even larger. This creates a challenge to accurately and efficiently track accesses: If tracking happens at a fine granularity (e.g., byte), a wider access needs to simultaneously mark multiple entries. This creates the need for a bigger tracking table and increases circuit complexity. If, on the other hand, tracking is done at a coarse granularity, e.g., 8-byte quadword,[3] we lose the ability to distinguish two finer grain accesses to two different portions within the same quadword. In general, we found this to be an acceptable approximation in our test environment. However, for certain applications, this can result in pathological scenarios and can create numerous spurious replays. Consider a conceptual example: If a loop performs a read-modify-write cycle on an array of bytes and the issue logic favors loads over stores, then there can be many reorderings of stores and loads from consecutive iterations to neighboring bytes. These would be incorrectly construed by the coarse-grain tracking logic as order violations to the same memory location, resulting in unnecessary replays and performance losses.

4.2.1 Tracking Loads

One approach to handle multiple access widths efficiently and without complicated circuitry is to use a two-tiered system: a main table to track the predominant access width (64 bits, or quadword in our experimental system) and a "side" table to keep *additional* fine-grain information. For the programs we studied, fine-grain accesses are less numerous and the side table need not be big in size. If a quadword is being accessed at the granularity of byte, the main table only tracks the age of the youngest instruction accessing any part of the quadword whereas the side table can tell us the age of instructions accessing every one of the 8 bytes within the quadword (Fig. 2). At any moment, the fine-grain tracking of any quadword is done at a fixed granularity. This granularity, however, can be different for different quadwords and can change over time for the same quadword. The mechanism works as follows:

Each entry in the main table contains 2 bits to encode the current tracking width. When a load-quadword instruction executes, it only accesses the main table. When a load instruction with a narrower width executes, we may start to track at a finer granularity. This happens when there are no other pending loads accessing the quadword, i.e., the entry's age is invalid. At this time, the load will set the entry's width field to its own width and proceed to the side table to update it. It will also set the age of the main table. This way, the main table *always* contains the age of the



ExecutionOrder	Instruction	Address	Age
③	`ldl`	0x48	16
④	`stl`	0x4C	17
①	`ldb`	0x48	18
②	`ldb`	0x4D	19

Fig. 2. Example of table update with width and age upgrades. *ldl*, *stl*, and *ldb* stand for load-longword, store-longword, and load-byte, respectively. For illustration purposes, the example shows a side table with only 16 entries and takes the four least significant bits of byte, word, or longword addresses, depending on the access width. The state shown is after the two *ldb* instructions have finished execution. The current tracking width (01) of the quadword is byte.

youngest load that accessed *any* part of the quadword. (This allows us to avoid the circuit complexity of having to scan multiple entries as will be explained later.) If, when a load executes, the main table's entry is valid, then we compare the entry's existing width and that of the load. There are three possible cases: the entry's width is 1) the same as, 2) wider than, or 3) narrower than that of the load.

Case 1 is the most straightforward to handle. We simply update the main table and, if the width is less than quadword, the side table.

In case 2, because we are already tracking at a coarser granularity, finer grain information is useless. So, we simply "upgrade" the width of the load to the same as the entry's and proceed as in case 1.

In case 3, we can potentially treat the wider load as a collection of loads with narrower width and keep on maintaining fine-grain information currently tracked by the system. However, we want to avoid the complexity of having to access and update multiple entries. Thus, we forgo the fine-grain information we have accumulated so far and upgrade the entry's width. It is nontrivial to determine the correct age to keep at the new granularity as can be explained by an example.

In Fig. 2, we show a sequence of four instructions in which the two younger *ldb* instructions execute first. The figure shows the state of the two tables after their execution. When *ldl* executes, we will upgrade the tracking width to longword. Note that the way the side table is indexed depends on the width. For instance, to track words, the least significant bit is ignored and a quadword maps to eight consecutive entries. When the tracking width changes from byte to longword, the same quadword maps to an entirely different region of two consecutive entries. To get the most

3. Our study uses Alpha ISA and we follow their terminology to refer to 8-, 16-, 32-, and 64-bit quantities as *byte*, *word*, *longword*, and *quadword*, respectively [2].

accurate information, we should scan the old entries to update the new ones. Specifically, find the youngest age among entries 8 to 11 and update entry 2 unless it has an even younger age. Similarly, we need to scan entries 12 to 15 and update entry 3. Hard-wired logic is unsuited for iterative tasks such as scanning different entries to consolidate information.

Our solution, therefore, is simply to always upgrade all the way to quadword whenever an upgrade is required. This way, the side table is not accessed at all and the main table entry already contains the age of the youngest load accessing any part of the quadword. With this policy, there is never any need to either read or update more than one entry in the side table. Though this loses some information and is conservative, the only impact is on performance as it might generate spurious replays. In our testbed, we found that width upgrades are extremely rare to begin with, and that spurious replays due to width upgrades are even rarer. This justifies using the simple circuit. In other environments such as x86, width upgrades may not be as rare and a more involved circuit to scan and update multiple entries may be warranted.

4.2.2 Checking the Tables

When a store executes, it compares its age to that in the corresponding entry in the main table. If the store's age is younger, then no replay is needed. Otherwise, we may need to replay. However, if the store has a narrower width, we may have more fine-grained tracking information from the side table that can rule out violation.

If the store's width is the same or narrower than the corresponding entry's width, we simply treat the store as a wider access (with the same width as the entry) and consult the side table to determine whether we replay. However, if the store's width is wider that the entry's, the side table still contains more information about access ordering, but it does so in a "fragmented" way, and we need to piece together the information from multiple entries, which we avoid as before. In this case, we simply ignore the side table.

In summary, the side table essentially provides some extra space to allow us to "zoom in" and track select quadwords at a finer granularity. At any time, a single quadword is tracked with only one data width. However, that width differs from quadword to quadword. The entries in the side table, therefore, are tracking different widths. For circuit simplicity, the side table never requires update or checking of multiple entries. In general, we found the side table does not need to be as big as the main table. Hashing conflicts may incur spurious replays but do not affect correctness.

4.2.3 Handling Unaligned Accesses

A memory access is unaligned when the starting address is not a multiple of the size of the access. An unaligned access of, say, a word, straddles the "natural" boundary of two words. Depending on the exact location, this natural word boundary can also be the boundary of two longwords or even two quadwords. Unless the unaligned word straddles quadword boundary, the unaligned word access can be treated as an aligned longword or quadword access. In the cases when the unaligned data straddle a quadword boundary (regardless of the size of the access), it has to be treated as an access to both quadwords.

4.3 Pollution-Resistant Age Representation

Using ROB entry ID as an age representation is a source of spurious replays due to a branch misprediction recovery or a replay. Recall that updates to the load tables (main table and side table) occur at execution time when the instructions are speculative and may be on the wrong path. Later on, when a misprediction recovery or replay takes place, the processor rolls back and starts to reuse some ROB entries for newly fetched instructions and, as a side effect, reassigns older ages to memory instructions on the right path. The age recorded in the load tables by the wrong-path instructions can easily be younger and create the appearance of order violation, which in turn triggers unnecessary replays. As it turns out, this is actually a very serious source of false replays: the number of false replays triggered this way is about 10 times that of true replays.

If we can ensure age representation to monotonically increase—even during misprediction recovery or replay, we continue to assign age IDs younger than all squashed instructions—we can eliminate most of the age confusion and the resulting false replays. This can be achieved with a very simple mechanism. We augment the ROB entry ID with a few bits of *prefix*. When we squash instructions during a replay or a recovery, we increment the prefix. This way, even though the least significant bits of the age representation (the ROB entry ID) is being rolled back, the overall age number is still increasing and is guaranteed to be younger than any squashed instruction. Note that a prefix is already needed to handle ROB wraparounds [7]. We only increase the number of bits needed. Similar to [7], if an ROB wraparound demands a prefix increment, but the new prefix is still being used by in-flight instructions, we stall the dispatch until the new prefix is freed up. However, if after a recovery or a branch misprediction we cannot increment the prefix anymore, we do not stop dispatch but simply keep using the original prefix.

Finally, it is worth noting that the increment of prefix cannot prevent certain false replays. Consider a series of instructions (and their age representation): load l_1 (4), store s (5), branch b (6), and load l_2 (7). Suppose all three memory instructions map to the same load table entry and the execution order is l_1, l_2, b followed by a misprediction recovery, and then s. The execution of l_2 will irrevocably increment the age recorded in the load table's entry to 7 and create the impression of an order violation when later s executes, no matter what age we assign to the newly dispatched right-path instructions.

5 Delayed Memory Dependence Checking

We now discuss a second timing-centric implementation. In this design, we separate the use of timing and address information and exploit the timing information to quickly cut down the dependence checking workload necessary, which in turn allows us to delay the checking to commit time so that a simpler, area-, and energy-efficient circuitry suffices.

5.1 YLA-Based Filtering

As explained earlier, when memory instructions happen to issue in program order, certain dependence checking becomes unnecessary. For instance, when a store searches the LQ, the intention is to identify any *younger* load to the same address that has already issued. The knowledge that no younger load has ever issued can help avoid accessing the queue for dependence violation checking altogether. To know that, we can track explicitly the *age* of issued loads, specifically, the age of the youngest one.

5.1.1 The YLA Register

We keep a dedicated register to hold the age of the youngest issued load. We call the register *YLA* for Youngest issued Load's Age. When a load executes, the YLA register is updated with the load's age if it is younger. When a store is resolved, it compares its age with that recorded in YLA. If the store is younger, no violation has occurred and no further action is needed. We call that a YLA hit. Otherwise (a YLA miss), a potential violation of memory ordering can exist and additional checking must be done.

5.1.2 Multiple YLA Registers

Dependence violations are only possible if both the store being executed and the younger issued loads access the same memory location. Therefore, the YLA-based filtering can be enhanced by taking into account *some* address information. Specifically, we can use multiple YLA registers to cover different address banks and to spread loads and stores among them according to their memory addresses (a few bits are enough). This can increase the probability of YLA hits, thereby reducing LQ searches.

5.2 Delayed Memory Dependence Checking

Although the simple age-based filtering can be a stand-alone optimization applied to reduce the associative searches for the LQ and is in fact very effective, its main appeal really lies in the new opportunities of more effective designs it enables. As only a small portion of stores need to check for possible premature loads, the associative searching can be substituted with energy-efficient albeit slower processes. Specifically, for every store, 1) instead of using associative searches of the LQ, we employ a sequential process to inspect each possible premature load (we keep a FIFO LQ) and 2) we delay the process to commit time. We call this scheme *Delayed Memory Dependence Checking* (DMDC) [8].

The high-level procedure consists of three steps as follows:

1. The YLA-based filtering logic classifies stores into two disjoint sets at issue time: those that do not need to check for premature loads as no younger loads have issued earlier and those that do need additional checking. For convenience, we call them *safe* and *unsafe* stores, respectively. This safety information is recorded in the SQ.
2. As an unsafe store commits, it triggers a special *checking mode* to start the sequential checking process.
3. During the checking mode, as a load completes, we test to see if a memory dependence violation has happened at execution. Note that the responsibility of checking conventionally associated with stores is shifted to loads.

From a high-level point of view, the reason such a process is viable is twofold. First, although a sequential process gets less done per cycle (e.g., only checking one load against a store), the throughput is sufficient as only a small portion of stores need to enable the delayed checking. Second, as actual memory dependence violation is typically very rare, the small delay incurred in identifying dependence violation (by postponing the checking to commit time) will not result in a significant slowdown.

5.3 Implementation

From an implementation point of view, delaying memory dependence violation checking is a challenging task as we need to propagate information through some media to be used at a later time. The information includes address and execution timing. Conventionally, the timing information is implicitly embedded in the queue: When a store issues, by inspecting the LQ, it is easy to know which loads have already executed. When we delay the checking, the LQ no longer provides the correct timing information.

A naive implementation of DMDC would call for explicit recording of execution time of loads and stores in the LQ and SQ. Additionally, since the delayed checking only starts as an unsafe store commits, we would have to move information kept in the SQ to some other place as the SQ entry is recycled. Instead of introducing these complexities, we choose a much simpler circuit. With this design, we introduce approximations in both timing and address information.

The algorithm is simple. When a store's address is resolved, the YLA registers are consulted to determine whether the store is safe. If not, some younger load may have executed prematurely. The age can be as young as that indicated in the YLA register corresponding to the store's address bank. We will remember that age and later check every load in between. These loads effectively form a *checking window*. The timing approximation is that only some loads in this window actually executed prior to the store, but all are treated as potentially premature. The architectural support needed to encode this checking window is a single global *end check* register.

As for the address information, we use a hash table called the *checking table* to mark the location of the store. Two memory accesses are considered overlapping when they hash to the same entry. At commit time, loads in the checking window are inspected to see if they map to an entry marked by the unsafe store. With these supports, we no longer need any associative LQ. A FIFO-allocated queue to record the address is enough. In fact, as we will only use the address to index the checking table, we only need a queue to record the hash keys of the loads. The operation procedure is given as follows:

- **Issue—store:** If a store is unsafe, the *end check* register is updated with the content of the YLA register corresponding to the store's address bank—unless the *end check* register already records a younger age.

- **Issue—load:** When a load issues, we record its hash table entry in the FIFO LQ.
- **Commit—store:** As an unsafe store commits, it sets the corresponding entry in the checking table and triggers the *checking mode* if not already active.
- **Commit—load:** During the checking mode, as a load commits, it indexes the checking table. A marked entry indicates a *possible* dependence violation and a replay is performed. Otherwise, no further action is required. After the load pointed to by the *end check* register commits, the checking mode is terminated and the checking table is cleaned up.

Local versus global DMDC. Note that in this implementation described above, the *end check* register is a global register in that there can be multiple unsafe stores in-flight and they all update the register as they *issue* (can only increase the checking window size). Thus, when an unsafe store commits and activates the checking mode, the register may have been pushed forward by an unrelated store to the end of its own checking window. In the pathological case, the end of the checking period can be perpetually pushed forward and never reached, creating an endless checking period. Without the chance to clean the table, the probability of a false replay will become higher and higher. Note that even in this case, a replay (false or true) will clean the table. As we will show later, this does not lead to runaway performance degradation—the worst-case performance impact is 5.8 percent in our experiments.

An alternative to using the global register is to use "local" information: Each unsafe store can remember its boundary of checking period and only update the register at commit time. We call these two options *global* and *local* DMDC.

5.4 Exploiting Safe Loads

A very important and cost-effective optimization to the above design is to identify *safe loads*. When a load issues, if all older stores have resolved their addresses, then we can already rule out the possibility of a store-load replay concerning this load and mark it as safe. During commit, a safe load bypasses the checking process even in checking mode. This not only saves energy but, more importantly, avoids false replays due to the approximations.

The circuit support to identify safe loads is straightforward and will not affect SQ timing as it is much simpler than the forwarding logic: Typically, the forwarding control signal is generated as follows [9]: The physical address of the load is sent to the SQ's address CAM. The match signals of all the entries then go through a mask logic that inhibits match signals from entries younger than the load. These masked match signals then go through a priority encoder to find the youngest entry for forwarding (Fig. 3a).

To determine the safety of a load, we only need to feed the bit indicating unresolved address into the same kind of mask logic and use the masked result to pull down a global line precharged to high: if any masked bit is high, then the load is not safe (Fig. 3b).

In summary, the complexity of our proposed implementation of DMDC is very low as we use various approximations. The loss of information obviously leads to false replays. However, as we will see later in Section 6, due to the large

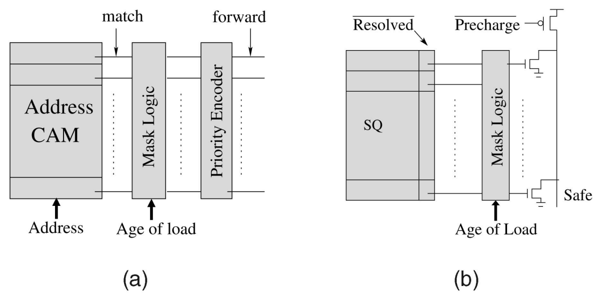


Fig. 3. (a) SQ forwarding logic. (b) Safe load detection logic.

number of loads and stores identified as safe, the number of false replays is very limited and thus the performance degradation is low.

5.5 Supporting Coherence and Consistency

DMDC essentially uses simplified forms to record the timing and address information of loads. Thus, coherence and consistency functionalities of the conventional LQ can also be supported, albeit with more conservative policies. The performance impact of these approximations depends on the exact consistency model supported and the characteristics of the coherence traffic.

5.5.1 *Ensuring Write Serialization*

Recall that WS is violated only when *all* conditions are satisfied at the same time: loads executed out of order, an external invalidation happened in between, and all accesses are to the same location. In reality, performing a correction when only a subset of the conditions happen may be sufficient: it is more conservative and thus guarantees correctness; and the combination of even a subset of the conditions may be infrequent enough that the cost of extra replays is minimal.

In DMDC, the timing information of loads is not kept in the LQ. Thus, we will not be able to tell if two loads executed out of program order with respect to each other. We could add another structure to explicitly track timing as in Section 4.1.2, but we choose to leverage the existing DMDC mechanism and achieve WS enforcement economically and effectively. We do this by ignoring the timing of the loads and conservatively performing a replay when two same-location loads are both in-flight when an invalidation happens. We replay from the younger load. This can be detected as follows:

We extend the checking table to have one extra bit per entry to capture the invalidation address—now, each entry has an *INV* bit in addition to the original bit(s), which we call the *WRT* bit(s). When an external invalidation arrives, the checking mode is also activated. However, instead of setting the WRT bit, we set the INV bit of the appropriate entries.

When committing a load during this checking period, the hash table is indexed. If the WRT bit is set, we replay as before. If only the INV bit is set, we do not replay as there is only one load (so far) to the location being invalidated. Instead, we "upgrade" the entry by setting the WRT bit(s). This ensures that if a second load to the same location is discovered in the checking period, a replay will happen.

To properly determine the end of the checking period, we need additional YLA registers. This is because for store-load dependence checking, YLA registers are best banked by word address. Since invalidations are at cache line granularity, we add another set of YLA registers banked by cache line address. With two sets of YLA registers, every address maps to one in each set. Of course, a load will need to update the two and a store is safe as long as one of the two records an older age.

Finally, we note that when an invalidation message starts or extends a checking window, a safe load (which is "safe" with respect to store instructions from the same processor) within that checking window can no longer be treated as safe.

5.5.2 Ensuring Sequential Consistency

Supporting SC is very straightforward. We mark *INV* bit as before and if a committing load maps to an entry with the *INV* bit set, we replay from this load onward and terminate the checking mode.

5.6 Other Design Options

5.6.1 Handling Multiple Data Sizes

As discussed earlier, memory accesses are performed at different sizes. Tracking accesses at a coarse granularity (e.g., longword) will incur unnecessary replays. However, unlike the issue-time dependence checking mechanism discussed in Section 4, the checking table in the DMDC scheme does not record age IDs but only marks single bits. Therefore, it is very easy to use a bitmap to preserve more fine-grain information about the memory accesses. In this paper, we index the checking table using quadword address but use a 4-bit bitmap to discern accesses with a smaller width.

5.6.2 Using a Checking Queue

In the design discussed above, we use a hash table to record the store address information. The primary advantage is the conceptual simplicity. As multiple unsafe stores can have overlapping checking periods, a load may need to be checked against the address of multiple stores. A hash table can accommodate any number of stores, and loads only need indexing. An alternative is to use an associative checking queue to keep track of the address of unsafe stores. Of course, when the queue overflows, a replay is necessary.

5.7 Contrasting the Two Designs

Though both adopt a timing-centric approach to dependence checking, the two designs have quite different implementations. The issue-time memory dependence checking (IMDC) design takes a unified dependence checking step and acts much like a drop-in replacement of the conventional LQ. In contrast, DMDC is more decoupled, taking two steps to finish the checking: the first step acting mostly on timing information, and the second step, the address information.

The chief advantage of such a decoupled approach is the economy and simplicity of encoding information. The timing information is maintained by a handful of registers. This allows a much simpler and cheaper bit table to be used for address information, providing two tangible benefits. First, for the same hardware cost, more entries can be maintained to encode more address information and thus reduce false replays due to hashing conflicts. Second, as each access can be represented by only a bit, we can track different access widths very effectively via a bitmap and can avoid certain types of false replays that the side table approach used in IMDC could not avoid.

This better discernment of spatial information comes at the expense of some loss of information about timing: The checking window encompasses loads executed after the store and these loads, if having an overlapping address, will be misconstrued as having violated execution order and trigger a false replay. Note that the optimization exploiting safe loads significantly mitigates this problem. Section 6 shows quantitative results to help understand better the tradeoff between the two designs.

6 Experimental Analysis

6.1 Experimental Setup

We have evaluated our proposed design using a heavily modified version of SimpleScalar [10] with the Wattch extension [11]. The modeling of the LSQ is modified to faithfully reflect the state of the art in modern microprocessors. We allow the issue of loads with unresolved older stores. The SQ supports load rejection [1] and rejected load retries later. The processor also handles partial memory matches between memory addresses. For energy modeling, the Wattch is modified to reflect the common approach of using a split LQ/SQ organization. A memory instruction incurs energy cost for insertion in one queue and associative search of the other. Our table structures are modeled similarly as a pattern history table used in the branch predictor, using different number of entries and entry width, of course. The technology parameters correspond to 0.1 μm, with a 1.9-V V_{dd} and 1.25-GHz clock.

In the applications and simulation windows we studied, true store-load replays are very rare, on the orders of a few per million instructions on average. Even with our approximations, replays remain rare. Thus, PC-based prediction and replay prevention mechanisms seem unnecessary and are not modeled for either the baseline or our designs. The microarchitecture loosely models after POWER4 [1]. Some of the simulation parameters are listed in Table 1. To understand the issue of scalability, we use three configurations, varying only the buffering structures' size.

The uniprocessor evaluation is performed using all 26 benchmarks from the SPEC CPU2000 suite compiled for the DEC Alpha instruction set. For the experiments, we simulate single sim-point regions [12] of 100 million instructions. To better understand the timing-centric design in multiprocessing environment, we also simulate a set of 10 parallel applications including those in the SPLASH-2 benchmark suit [13] and the parallel version of the traveling salesman problem [14] (*tsp*). When simulating parallel benchmarks, we fast-forward through the initialization phase and simulate 1.6 billion instructions for all threads or to completion, whichever comes first. This multiprocessor simulator is an in-house version based on SimpleScalar that supports both SC and DEC Alpha's consistency model

TABLE 1
Simulation Parameters

Processor core
Issue/decode/commit width: 8/6/12 *Functional units:* INT 3+1 mul/div, FP 3+1 mul/div *Branch predictor:* Bimodal and Gshare combined -Gshare:8K entries, 13 bit history -Bimodal/Meta table/BTB entries:4K/8K/4K (4 way) Branch misprediction penalty: 7 cycles
Memory hierarchy
L1 instruction cache: 64KB, direct-mapped, latency= 2 cycles *L1 data cache:* 32KB, 2 way, latency= 2 cycles, 2 ports *L2 unified cache:* 1MB, 8 way, 128B line, latency= 15 cycles *Memory access:* 120 cycles
Simulated configurations
config 1: Issue queue= 32INT/ 32FP, ROB=128, LQ/SQ= 48/32, Registers= 100INT / 100FP, DMDC: Checking table= 1024, IMDC: Main Table= 512 - Side Table= 128 *config 2:* Issue queue= 48INT/ 48FP, ROB=256, LQ/SQ= 96/48, Registers= 200INT / 200FP, DMDC: Checking table= 2048, IMDC: Main Table= 1024 - Side Table= 256 *config 3:* Issue queue= 64INT/ 64FP, ROB=512, LQ/SQ= 192/64, Registers= 400INT / 400FP, DMDC: Checking table= 4096, IMDC: Main Table= 2048 - Side Table= 512
Prefix for age representation: 5 bits

[2]. The Alpha consistency is used when testing the enforcement of WS.

As we lack a validated energy model for multiprocessors, our parallel benchmark simulation will be focused on studying the performance impact and the number and source of false replays. The modeled chip multiprocessor (CMP) contains eight cores, each as described in Table 1, taking only *config2*. However, the L2 cache is a globally shared 2-Mbyte eight-way (128-byte line) cache. For the SPLASH-2 benchmarks, the L1 cache size is set to 8 Kbytes, according to the recommendation in [13] to mimic realistic cache miss rates. For *tsp*, the L1 size is 32 Kbytes. Cache coherence is achieved with a snoop-based MESI protocol implemented on top of an on-chip bus interconnect with a one-cycle link latency and a 16-byte link capacity. All write-serialization and SC incurred replays are modeled following [2]—the pipeline is flushed and instruction fetch restarts from the faulting instruction.

In the following, we perform some quantitative analysis to further understand the rationale behind age-based mechanisms and the effectiveness of the proposed designs. For brevity, applications are treated as two groups—integer (INT) and floating point (FP)—and most results are only shown as the average of metrics, often normalized to the conventional configurations (baselines). As a reference, we also show the absolute per-application performance results in the baseline architecture. Results of SPEC benchmarks are summarized in Fig. 4. Results of parallel benchmarks are shown later in Fig. 11.

6.2 Overall Comparison

6.2.1 Performance and Energy Results

We start with the bottom-line results where we compare the performance and energy of complete systems using either

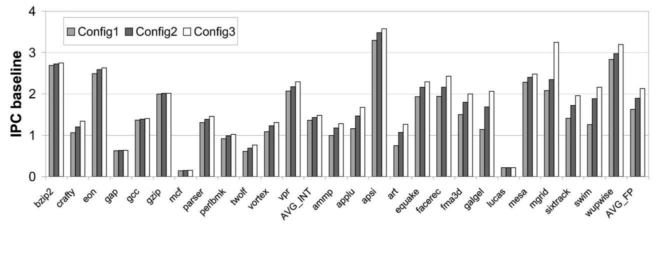


Fig. 4. Absolute performance in our baseline configurations.

conventional LQ or the timing-centric alternatives (IMDC and DMDC). For brevity, we use only the local version of DMDC. Fig. 5 shows the performance impact and the energy savings (both in the LQ only and processor-wide) in three configurations.

The figure illustrates the following points. First, by eliminating the need for associative LQ, most of the energy consumption of the LQ is eliminated. The added energy compared to that is very insignificant, especially in DMDC. As a result, overall energy reduction in implementing the LQ functionality is about 88 percent to 97 percent, depending on the design and configuration. Note that this comparison is very conservative in that we are not considering coherence requirements. If coherence is considered, the baseline LQ energy consumption can be far higher as not only far more searches to the LQ are performed, the LQ itself needs to be multiported, increasing the cost of every search.

Second, the performance degradation is very limited. The main reason is that false replays as a result of approximations in timing-centric designs are relatively rare. In *config2* for example, the average number of false replays (DMDC) is about 25 and 11 per 1 million committed instructions for INT and FP applications, respectively. The average slowdown is practically negligible. In DMDC, the worst-case slowdown among all configurations and for all applications is only 1.2 percent. Furthermore, without LQ capacity issue in the timing-centric designs, the chance of stalling is reduced, which can offset some of the slowdown due to replays. Indeed, performance can even increase, as seen in the best-case results in the FP applications.

Third, as the machine scales up its capacity of in-flight instructions and, in particular, the size of the associative LQ, the portion of energy spent there also increases. As a

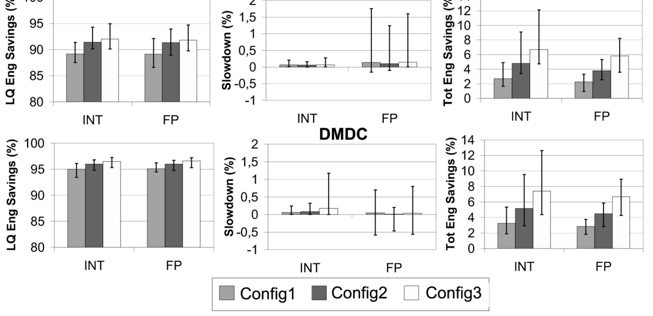


Fig. 5. LQ energy savings, slowdown, and total processor-wide energy savings of IMDC and DMDC in three processor configurations. Each bar shows the average of the group of applications, whereas an "I-beam" shows the range of the value within that group of the applications.

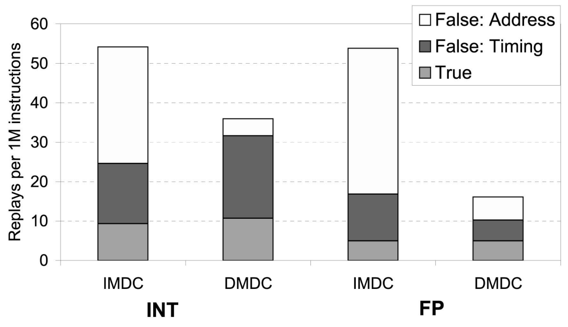


Fig. 6. Breakdown of store-load replays.

result, the energy savings from timing-centric designs become more pronounced. Overall, taking into account the energy overhead from the performance degradation and extra circuit, chip-wide energy savings can be about 2.2 percent to 7.5 percent depending on the design and configuration (again, conservatively ignoring multiprocessor issues).

Finally, DMDC is generally more effective than IMDC, showing more energy savings and less performance degradation. This is due to the decoupled approach DMDC uses, which allows a simpler and more effective disambiguation of address information.

6.2.2 Analysis of Replays

Clearly, the timing-centric approaches involve simpler and cheaper operations in memory dependence checking: For the most part, the system performs only comparisons between one or two pairs of age IDs, which are 12 to 14 bits. In contrast, conventional LQ performs associative matches of wide operands (e.g., 64-bit memory addresses) with tens of others. However, the main cost of the timing-centric design is false replays due to approximation of address or timing information. Thus, understanding the replays is very important.

Fig. 6 shows the breakdown of replays in the timing-centric implementations. In general, false replays are relatively rare, less than 1 for every 10,000 instructions. False replays can be roughly broken down into address related and timing or pollution related. As explained earlier in Section 5.7, DMDC allows a simpler, more space-efficient bitmap table to resolve addresses. As a result, we get fewer hashing conflicts and better handling of accesses with different widths. Although delaying address comparison to commit time results in some loss of timing information (and thus false replays), it naturally eliminates replays falsely generated by squashed instructions. In contrast, for IMDC, squashed loads pollute the load table—even though pollution is mitigated by our age representation.

6.3 In-Depth Analysis

To better understand why the timing-centric approach is effective, we need some in-depth analysis to reveal the inner working of the mechanism. For brevity, in the following, we focus on *config2*. The results for the other configurations show similar trends.

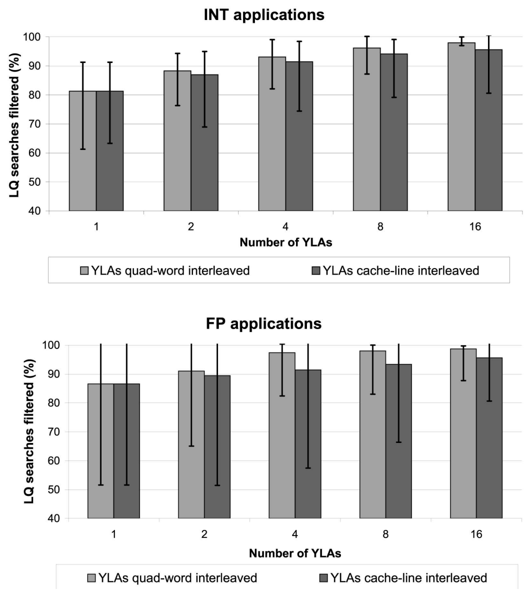


Fig. 7. Percentage of safe stores marked using YLA registers with different interleaving.

6.3.1 YLA-Based Filtering

We first inspect the YLA registers as a filtering mechanism. As shown in Fig. 7, with even a single YLA register, an average of 81 percent (INT) and 86 percent (FP) of stores can be marked as safe, and their LQ searches filtered out. With multiple address-interleaved YLAs, these percentages are even higher. As shown in Fig. 7, with eight registers, the filtering efficiency is a remarkable 96 percent to 98 percent—only 2 percent to 4 percent of stores are not marked as safe.

Recall that to support invalidations, we use another set of YLA registers to determine the end of an invalidation-triggered checking window. In that case, the YLA registers have to be cache line interleaved. Naturally, one possibility is to use only one set of cache-line-interleaved YLA registers. However, as we can see in Fig. 7, quadword-interleaved YLA registers are far more effective to handle in-flight stores. Indeed, using 16 line-interleaved YLA registers, we are only able to mark about as many safe stores as using four quadword-interleaved YLA registers. Therefore, we choose to employ two sets of eight registers each using different interleaving.

Energy savings. Using YLAs alone can save a significant number of LQ searches. As a result, energy consumption in the LQ is also reduced. With eight YLA registers, the reduction in LQ energy is about 31.7 percent. That translates into about 1.5 percent processor-wide energy savings. Note that the savings are obtained without a performance impact.

Comparison with address-only filtering. YLA exploits an important characteristic of load and store execution to rule out dependence violation: their relative timing. Using only one age register, we can already filter out a very significant portion (81 percent to 86 percent) of stores. When address interleaving is employed, the effect is quite dramatic—only a few percent of stores are left unfiltered. In comparison, this is much more effective than if only address information is used, such as with a bloom filter (BF) [15], as can be seen from Fig. 8.

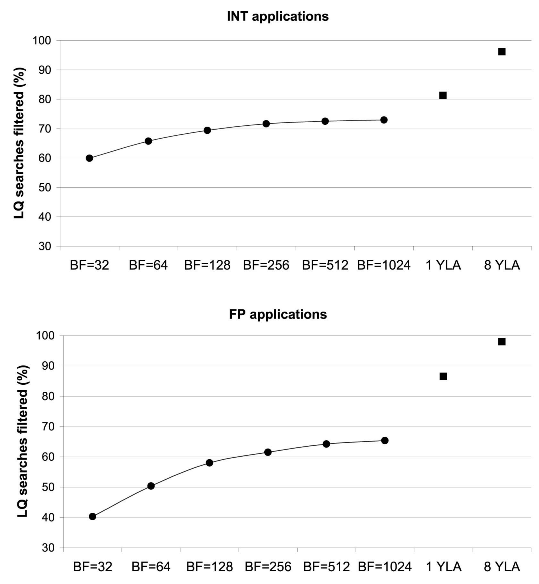


Fig. 8. Comparison of the filtering capability of using one or eight YLA registers and BFs with different sizes (H0 hashing function [15] is used).

6.3.2 Checking Window

Table 2 shows some statistics of the checking window for the DMDC approach. For global DMDC, on average, a checking window covers about 26 (INT) and 98 (FP) instructions and has 8 (INT) and 34 (FP) loads in between. Out of these, only about four to seven loads are unsafe and need to go through the checking process. Clearly, with only about 2 percent to 4 percent of stores characterized as unsafe, and each one only requiring to be cross-checked with a handful of loads, the sequential process used in DMDC is sufficient to provide the throughput for dependence enforcement. Furthermore, although DMDC detects a violation later than conventional system or IMDC, this delay (19-45 cycles) is relatively small, especially considering that replays are infrequent.

In local DMDC, due to locally recording checking window for every unsafe store, the windows are less likely to overlap to form bigger ones. The effect is that windows are about 20 percent shorter and contain proportionally fewer loads. The percentage of safe loads, however, reduces slightly faster. This is expected as the shrinking windows are more likely to exclude safe loads.

On average, the processor spends about 7.4 percent (INT) and 2.2 percent (FP) of the cycles in checking mode. As there are more safe stores in FP applications in general, it is more likely to finish the current checking window before encountering another unsafe store. On average, 51 percent of the windows contain just one unsafe store. In INT applications, this becomes 45 percent.

6.3.3 Safe Loads

Safe loads are quite numerous in typical executions. On average, 91 percent (INT) and 97 percent (FP) of loads are safe. However, as seen in Table 2, the percentage of safe loads is much smaller inside the checking window but still nontrivial, about 35 percent to 56 percent. (This is not surprising as the checking mode is triggered only when there are out-of-order executions—and hence nonsafe loads.)

TABLE 2
Number of Instructions, Loads, Safe Loads within a Checking Window, and Cycles of Delay in Handling True Replays in Global and Local DMDC Implementations

Global	instructions	loads	safe loads	delay
INT	26.0	8.1	3.3	19.6
FP	97.9	34.6	19.4	45.0
Local	instructions	loads	safe loads	delay
INT	20.2	6.4	2.2	19.0
FP	77.4	28.2	15.8	37.5

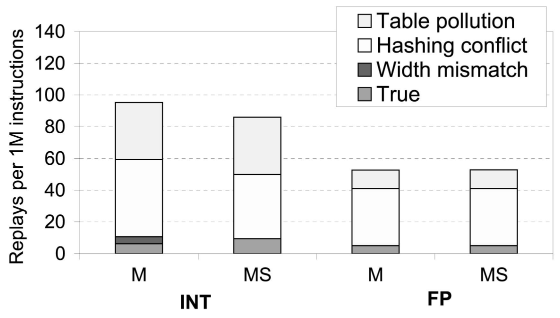


Fig. 9. Breakdown of replays in IMDC.

The primary benefit of detecting safe loads is to cut down the number of false replays. Indeed, with the safe-load mechanism, the number of false replays is reduced by an average of 92 percent and as high as 97 percent in integer applications. In other words, without safe loads, the number of false replays can increase by more than an order of magnitude. The simple circuit to detect safe loads is clearly worthwhile. In FP applications, the reduction is less significant, about 75 percent.

6.3.4 False Replays

IMDC. Fig. 9 shows the breakdown of the replays in IMDC into the following categories (from bottom up): true replays, false replays due to the mismatch between the access width and the tracking width, those due to hashing conflict, and finally, those due to table pollution caused by squashed instructions. We show the results of IMDC using only the main table (M) and using the main plus the side table (MS).

We see that the results are rather intuitive: 1) Integer applications tend to have many narrow-width data accesses, which cause some false replays when we track access order only using the main table at quadword granularity. FP applications, on the other hand, have far fewer replays due to width mismatch when we only use the main table. With the use of the side table, these false replays are almost completely eliminated; 2) due to the pollution-resistant age representation, false replays due to pollution are moderate. Integer applications tend to suffer more from table pollution than FP applications, and 3) because the working set of FP applications is generally larger than their integer counterpart, hashing conflict-induced false replays are relatively higher. Because they tend to have fewer branch mispredictions and replays, pollution-induced replays are fewer for FP applications.

TABLE 3
Breakdown of the Number of False Replays per 1 Million Committed Instructions in Global and Local DMDC

Global		Load issued before store	Load issued after store	
			X	Y
INT	Address match	–	27.8 (67%)	7.4 (18%)
	Hashing conflict	4.1 (10%)	1.2 (3%)	0.9 (2%)
FP	Address match	–	4.7 (35%)	5.2 (38%)
	Hashing conflict	2.9 (22%)	0.1 (1%)	0.5 (4%)

Local		Load issued before store	Load issued after store	
			X	Y
INT	Address match	–	17.4 (69%)	3.0 (12%)
	Hashing conflict	4.0 (16%)	0.5 (2%)	0.3 (1%)
FP	Address match	–	4.6 (41%)	0.4 (4%)
	Hashing conflict	4.5 (40%)	0.5 (5%)	1.1 (10%)

The two subcategories are given as follows. X: load falls into the "real" checking window of the store. Y: load is checked because multiple checking windows are merged together.

DMDC. Recall that DMDC makes two approximations in the dependence checking: address and timing. We can thus break down the false replays according to the approximations that triggered them. This breakdown is shown in Table 3. In the timing approximation, a load may be suspected of violating dependence with an older store even though the load actually issued after the store. Within this type of loads, there are two subcategories. In the first case (X), the load indeed falls into the checking window of the store. In the second case (Y), the load does not even fall into the determined checking window of the store. However, when multiple checking windows are merged together, a load will effectively be checked against other stores, even though it does not belong to their original checking window.

The first interesting thing to observe from the table is that the large majority of false replays are triggered because of either the timing approximation or the address (hashing) approximation but not both. This suggests that we can improve upon the two approximations largely independently.

Second, with the particular configuration studied (2,048-entry checking table), imperfect hashing is not the dominant cause of false replays in global DMDC, accounting for around 10 percent and 22 percent of all replays for INT and FP applications, respectively. Thus, increasing the size of the checking table will have limited effectiveness due to diminishing returns.

In local DMDC, the main benefit of having smaller windows is having more chances to clear the table to avoid unnecessary false replays. The average number of replays per 1 million committed instructions reduces from 41 to 25 (by 39 percent) for integer codes and from 14 to 11 (by 21 percent) for FP codes. Although the statistics are imperfect for pinning down exactly which replays are avoided,[4] they do suggest that false replays due to overlapping windows (the Y column) are indeed mitigated in local DMDC.

4. Note that the breakdown can fluctuate because 1) having different replays changes the timing of execution and thus can affect whether other loads will cause a replay and 2) different timing can affect the way we categorize false replays in our simulator in certain situations.

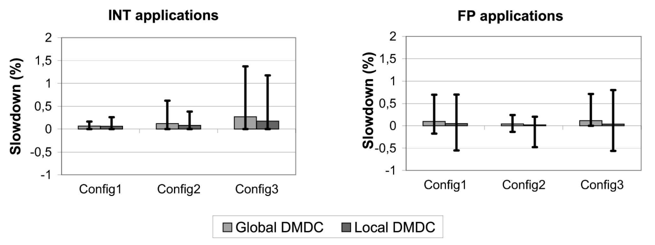


Fig. 10. The comparison of slowdown between local and global DMDC.

Pollution-resistant age representation. The pollution-resistant age representation attempts to increment the age prefix when a misprediction recovery or a replay takes place. Without this simple measure, pollution will be rampant and false replays will be unacceptably high. However, this measure requires sufficient number of bits in the prefix. Otherwise, we may frequently encounter the situation where we cannot increment the prefix and keep using the old one. We found that 3 bits are insufficient, whereas 5 bits are enough to make pollution level acceptable.

6.3.5 Performance Comparison of Local and Global DMDC

Because false replays are already infrequent even in global DMDC, the difference in energy and performance between global and local versions of DMDC is insignificant, as can be seen in Fig. 10. The local version moderately improves the effectiveness at the expense of a slight increase in design complexity.

6.3.6 Using Associative Queue in DMDC

Instead of using a hash table in DMDC, another option is to keep unsafe stores' address in an associative queue and check loads against all valid addresses in the queue.

This way, we will not have replays due to hashing conflicts but instead will have to replay when the queue cannot accommodate a new store. Based on statistics of the degree of overlapping checking windows, we estimate that the checking table we used (2,048 entries) is equivalent to a 16-entry associative queue in terms of *average* number of replays. Note that this estimate can only serve as a rough equivalency measure because individual applications behave wildly differently. If we calculate a per-application equivalent queue size, the results will be so divergent that their average is perhaps no longer meaningful.

6.4 Multiprocessor Considerations

Due to the various approximations, invalidations generated in a cache-coherent multiprocessor environment can also trigger false replays in our timing-centric designs. We performed studies to quantify the cost associated with our designs. We model two environments enforcing only WS (alpha consistency model) and enforcing SC. Fig. 11 shows the absolute performance in the conventional design and the relative performance degradation of using our timing-centric implementation to replace the associative LQ.

On average, the performance degradation is very small, which suggests that the increase in replays induced by the timing-centric designs is tolerable. In some cases, increases in replays degrade performance more noticeably. In other cases, the performance actually improves (indicated by bars below the axis) due to the absence of LQ fill-up induced stalls.

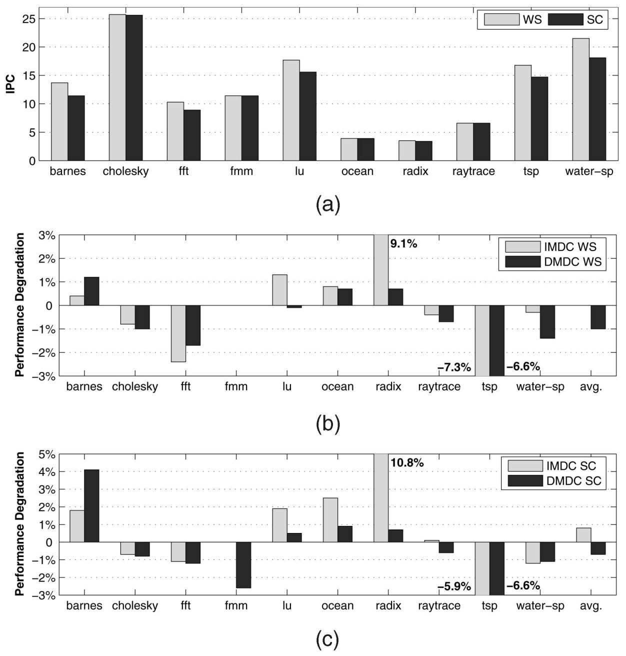


Fig. 11. Absolute performance of the parallel benchmarks on the baseline CMP machine enforcing either WS or SC, and performance degradation caused by IMDC and DMDC. (a) Baseline performance. (b) Enforcing WS. (c) Enforcing SC.

In fact, when ensuring only WS, we have an average performance gain. Our results show that extra false replays due to external invalidations are insignificant when enforcing WS, and tolerable when enforcing SC. Table 4 summarizes the absolute replay rate and a percentage breakdown by causes.[5]

Finally, we stress test our system by introducing false invalidations. The injected invalidations do not actually cause any cache line invalidation but only force our timing-centric memory logic to generate more false replays. We inject extra fake invalidations at an extreme rate of 1 every 10 cycles. The real invalidation rate observed is only about 1 every 100 cycles. Under this elevated invalidation frequency, with a relaxed consistency model (enforcing only WS), both IMDC and DMDC perform essentially the same as without injected invalidations. The average performance degradation is 0.01 percent and 0.16 percent, respectively. When enforcing SC, the impact of false replays is more noticeable. The average performance degradation for IMDC and DMDC is 1.5 percent and 3.1 percent, respectively. These results suggest that the approximations are reasonable and the performance results are quite robust against different degrees of background invalidation activities.

7 Related Work

In recent years, many schemes are designed to improve the conventional LSQ. Some schemes continue to rely on associative queues in these two blocks but cut down their access frequency using, for example, filtering. Others use

5. The per-cause absolute replay rate often varies by several orders of magnitude from application to application. This makes averaging per-cause absolute rates misleading as they tend to be distorted by extreme outliers. So, the breakdown is calculated as an average of per-application breakdown.

TABLE 4
The Breakdown of Replays by Cause and the Geometric Mean of Total Number of Replays per 1 Million Committed Instructions

IMDC		Breakdown			Total
		True	False		
			Address	Pollution	
WS	internal	7.0%	71.5%	20.8%	277
	External	0	0.6%	0.1%	
SC	Internal	3.6%	50.9%	23.0%	441
	External	1.9%	16.9%	3.8%	

DMDC		Breakdown			Total
		True	False		
			Address	Timing	
WS	internal	5.5%	29.5%	50.4%	119
	External	0	12.1%	2.5%	
SC	Internal	4.9%	16.9%	46.8%	99
	External	4.1%	17.4%	9.9%	

Internal replays are caused by dependence violations between instructions within one processor. External replays are caused by (potential) violations of WS or SC.

alternative circuit structures, such as indexed queues, to replace or augment the associative queues.

Sethumadhavan et al. [15] propose to use *BFs* to cut down the access of the queues. With a much smaller hardware budget, our age-based filtering is much more effective in cutting down unnecessary searches for LQ. However, we have yet to explore the implementation for SQ filtering.

Age-based filtering allows us to use a completely different process for verification: a sequential checking delayed to commit time (DMDC). Such sequential verification at commit time is not a new concept. Gharachorloo et al. suggested that a conceptual way of detecting speculative memory accesses that are incorrect is to repeat the access when the consistency model allows and compare the value [4]. In an implementation, naively reaccessing the memory subsystem for every load would lead to elevated memory bandwidth requirement. To reduce this requirement, timing and address information of the speculative accesses could be tracked to enable efficient filtering [16], [17]. The key difference between DMDC and this past work is that our focus is efficient circuit implementation. DMDC still follows the conventional approach of *ruling out* dependence or coherence violation through address and timing information of memory accesses. We showed that by primarily relying on timing information, we can use cheaper circuitry with acceptable performance loss. Unlike these value-based approaches, actual *results* of the memory accesses are not used at all in our design.

A central enabling factor for both IMDC and DMDC is that the relative timing information is very useful in ruling out dependence violations. While timing information is implicitly encoded in the conventional age-ordered LQ/SQ, explicitly recording and comparing age enables new ways of dependence checking. To that end, Roth [17] also explores age information. Through a hash table, a load can know if the last store to the same memory location has already retired before it is decoded. If so, load reexecution is not necessary. The intended applications of the two mechanisms are different and so are the design choices. For example, false negatives in [17] result in load reexecution. In contrast, a false negative causes a replay in our design, which is much more costly. As such, we need to keep false replays very rare. Also, while we have a conventional SQ without speculation, [17] is designed to support techniques such as SQ speculation [18], [19]. As such, our age information tracks execution timing, whereas theirs essentially tracks the commit time of a store.

While our work optimizes the violation detection logic (LQ), another set of related work optimizes the forwarding logic (SQ). Many proposals rely on memory dependence prediction [20], [21] to narrow the range of stores to forward from. Park et al. [22] employ the store-load pair predictor to predict the necessity of searching the SQ, thereby saving SQ search bandwidth. A number of two-level designs [23], [24], [25], [26] keep only a subset of in-flight stores in the smaller, faster first-level structure. This structure is allocated to stores predicted by dependence predictor or simply according to execution or program order. The larger second-level structure is either slower, address-banked, or without forwarding capability. A number of approaches have been proposed to predict the exact store for a load to forward from. This is either done entirely in hardware [27], [28] or with software support using a feedback-directed approach [29].

Finally, Garg et al. propose Slackened Memory Dependence Enforcement, where loads and stores are allowed to communicate via an L0 cache with minimum effort to correctly enforce dependence and simply rely on reexecution to provide a correctness guarantee [30].

8 CONCLUSIONS

In this paper, we have introduced a timing-centric alternative to CAM-based LQs. The central observation behind the proposed designs is that even with out-of-order execution, a significant majority of loads and stores demonstrate partial ordering. With the presence of the forwarding SQ, this partial ordering can rule out store-load replays in a large majority of cases. This makes a fully associative LQ an overkill and timing information should be a primary consideration in implementing a more efficient memory dependence checking mechanism.

We have presented two different timing-centric designs. The first one is an issue-time implementation (IMDC), which can serve as a drop-in replacement for the conventional LQ. While LQ implicitly encodes age information (as relative position in the queue) and explicitly records address, IMDC explicitly records age in a hash table, which implicitly encodes address. This way, power-hungry associative address comparison is replaced by much more energy-efficient indexing and simpler age comparison operations.

The second design, DMDC, further decouples the use of timing and address information. Age-based filtering is done using a few address-interleaved registers at instruction issue time. These registers keep track of age information and filter out about 95 percent of stores from further dependence-violation checking. For the remaining stores, these registers delineate the window of loads that need further inspection. Due to the remarkably effective filtering, the remaining loads and stores can be easily inspected using a sequential checking process at the commit time using only address information. Compared to IMDC, DMDC brings two additional benefits. First, it allows an even simpler, more space-efficient hashing bitmap to be used to track address information. This in turn cuts down on the number of false replays due to hashing conflict or access width mismatch. Second, the commit-time checking process naturally avoids certain false replays caused by squashed instructions. These benefits outweigh the cost of timing approximation due to delaying the checking process.

Besides greatly simplifying the LQ circuit structure, both designs allow significant savings in LQ energy consumption. At about 0.3 percent, the average performance overhead is negligible. Depending on the design and configuration, the average processor-wide energy savings for SPEC application suites range between 2 percent and 7 percent. Studies of parallel benchmarks showed that the timing-centric designs also work well in multiprocessor environment, with insignificant performance impacts.

ACKNOWLEDGMENTS

This work was supported in part by the Spanish government through Research Contract CICYT-TIN 2005/5619, CICYT-TIN 2008/508, Consolider Ingenio2010 2007/2011, by the Hipeac2 European Network of Excellence, and by the US National Science Foundation through Grants 0509270, 0719790, and 0747324.

REFERENCES

[1] J. Tendler, J. Dodson, J. Fields, H. Le, and B. Sinharoy, "POWER4 System Microarchitecture," *IBM J. Research and Development,* vol. 46, no. 1, pp. 5-25, Jan. 2002.

[2] *Alpha 21264/EV6 Microprocessor Hardware Reference Manual,* Compaq Computer, Sept. 2000.

[3] S. Adve and K. Gharachorloo, "Share Memory Consistency Models: A Tutorial," *Computer,* vol. 29, no. 12, pp. 66-76, Dec. 1993.

[4] 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),* pp. I355-I364, Aug. 1991.

[5] K. Yeager, "The MIPS R10000 Superscalar Microprocessor," *IEEE Micro,* vol. 16, no. 2, pp. 28-40, Apr. 1996.

[6] A. Garg, F. Castro, M. Huang, L. Piñuel, D. Chaver, and M. Prieto, "Substituting Associative Load Queue with Simple Hash Table in Out-of-Order Microprocessors," *Proc. Int'l Symp. Low-Power Electronics and Design (ISLPED '06),* pp. 268-273, Oct. 2006.

[7] P. Jordan, B. Konigsburg, H. Le, and S. White, *Data Processing System and Method for Using a Unique Identifier to Maintain an Age Relationship between Executing Instructions,* US Patent 5,805,849, Sept. 1998.

[8] F. Castro, L. Piñuel, D. Chaver, M. Prieto, M. Huang, and F. Tirado, "DMDC: Delayed Memory Dependence Checking through Age-Based Filtering," *Proc. 39th Ann. IEEE/ACM Int'l Symp. Microarchitecture (MICRO '06),* pp. 297-308, Dec. 2006.

[9] S. Meier, *Store Queue Multimatch Detection,* US Patent 6,523,109, Feb. 2003.

[10] D. Burger and T. Austin, "The SimpleScalar Tool Set, Version 2.0," Technical Report 1342, Computer Sciences Dept., Univ. of Wisconsin-Madison, June 1997.

[11] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations," *Proc. 27th Ann. Int'l Symp. Computer Architecture (ISCA '00),* pp. 83-94, June 2000.

[12] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically Characterizing Large Scale Program Behavior," *Proc. 10th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS '02),* pp. 45-57, Oct. 2002.

[13] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proc. 22nd Ann. Int'l Symp. Computer Architecture (ISCA '95),* pp. 24-36, June 1995.

[14] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, and W. Zwaenepoel, "TreadMarks: Shared Memory Computing on Networks of Workstations," *Computer,* vol. 29, no. 2, pp. 18-28, Feb. 1996.

[15] S. Sethumadhavan, R. Desikan, D. Burger, C. Moore, and S. Keckler, "Scalable Hardware Memory Disambiguation for High ILP Processors," *Proc. 36th Ann. IEEE/ACM Int'l Symp. Microarchitecture (MICRO '03),* pp. 399-410, Dec. 2003.

[16] H. Cain and M. Lipasti, "Memory Ordering: A Value-Based Approach," *Proc. 31st Ann. Int'l Symp. Computer Architecture (ISCA '04),* pp. 90-101, June 2004.

[17] A. Roth, "Store Vulnerability Window (SVW): Re-Execution Filtering for Enhanced Load Optimization," *Proc. 32nd Ann. Int'l Symp. Computer Architecture (ISCA '05),* pp. 458-468, June 2005.

[18] S. Subramaniam and G. Loh, "Fire-and-Forget: Load/Store Scheduling with No Store Queue," *Proc. 39th Ann. IEEE/ACM Int'l Symp. Microarchitecture (MICRO '06),* pp. 273-284, Dec. 2006.

[19] T. Sha, M. Martin, and A. Roth, "NoSQ: Store-Load Communication without a Store Queue," *Proc. 39th Ann. IEEE/ACM Int'l Symp. Microarchitecture (MICRO '06),* pp. 285-296, Dec. 2006.

[20] A. Moshovos, S. Breach, T. Vijaykumar, and G. Sohi, "Dynamic Speculation and Synchronization of Data Dependences," *Proc. 24th Ann. Int'l Symp. Computer Architecture (ISCA '97),* pp. 181-193, June 1997.

[21] G. Chrysos and J. Emer, "Memory Dependence Prediction Using Store Sets," *Proc. 25th Ann. Int'l Symp. Computer Architecture (ISCA '98),* pp. 142-153, June/July 1998.

[22] I. Park, C. Ooi, and T. Vijaykumar, "Reducing Design Complexity of the Load/Store Queue," *Proc. 36th Ann. IEEE/ACM Int'l Symp. Microarchitecture (MICRO '03),* pp. 411-422, Dec. 2003.

[23] H. Akkary, R. Rajwar, and S. Srinivasan, "Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors," *Proc. 36th Ann. IEEE/ACM Int'l Symp. Microarchitecture (MICRO '03),* pp. 423-434, Dec. 2003.

[24] A. Gandhi, H. Akkary, R. Rajwar, S. Srinivasan, and K. Lai, "Scalable Load and Store Processing in Latency Tolerant Processors," *Proc. 32nd Ann. Int'l Symp. Computer Architecture (ISCA '05),* pp. 446-457, June 2005.

[25] E. Torres, P. Ibanez, V. Vinals, and J. Llaberia, "Store Buffer Design in First-Level Multibanked Data Caches," *Proc. 32nd Ann. Int'l Symp. Computer Architecture (ISCA '05),* pp. 469-480, June 2005.

[26] L. Baugh and C. Zilles, "Decomposing the Load-Store Queue by Function for Power Reduction and Scalability," *IBM J. Research and Development,* vol. 50, nos. 2-3, pp. 287-298, 2006.

[27] T. Sha, M. Martin, and A. Roth, "Scalable Store-Load Forwarding via Store Queue Index Prediction," *Proc. 38th Ann. IEEE/ACM Int'l Symp. Microarchitecture (MICRO '05),* pp. 159-170, Dec. 2005.

[28] S. Stone, K. Woley, and M. Frank, "Address-Indexed Memory Disambiguation and Store-to-Load Forwarding," *Proc. 38th Ann. IEEE/ACM Int'l Symp. Microarchitecture (MICRO '05),* pp. 171-182, Dec. 2005.

[29] C. Fang, S. Carr, S. Onder, and Z. Wang, "Feedback-Directed Memory Disambiguation through Store Distance Analysis," *Proc. 20th Ann. Int'l Conf. Supercomputing (ICS '06),* pp. 278-287, June 2006.

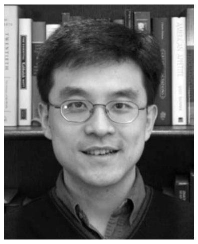
[30] A. Garg, M. Rashid, and M. Huang, "Slackened Memory Dependence Enforcement: Combining Opportunistic Forwarding with Decoupled Verification," *Proc. 33rd Ann. Int'l Symp. Computer Architecture (ISCA '06),* pp. 142-153, June 2006.

Fernando Castro received the MS degree in physics from the University of Santiago de Compostela in 2000 and the MS degree in electrical and computer engineering and the PhD degree in computer science from the Universidad Complutense de Madrid in 2004 and 2008, respectively. He is currently a teacher assistant of physics, electrical and computer engineering, and computer science at the Universidad Complutense de Madrid. His research interests include energy-aware processor design and efficient memory management. His recent activities focused on the LSQ structure, exploring new techniques to reduce its energy consumption without affecting performance. He is a member of the IEEE and the IEEE Computer Society.

Regana Noor received the BS degree in computer science and engineering from Bangladesh University of Engineering and Technology in 2003 and the MS degree in electrical and computer engineering from the University of Rochester in 2007. She is currently with the University of Rochester. Her research interests include designing scalable processor microarchitecture and investigation of efficient techniques for handling memory disambiguation.

Alok Garg received the BTech degree in electrical engineering from Indian Institute of Technology, Kharagpur, India, in 2002 and the MS degree in electrical and computer engineering from the University of Rochester, New York, in 2006. He is currently a PhD candidate at the University of Rochester. From 2002 to 2004, he was a design engineer at Paxonet communications, leading the design of north-side bridge for MIPS RM7000 processor. During his PhD, he also interned at Intel, Folsom, California, for eight months, where he worked on the verification of cache coherency protocol. His research interests include microarchitecture design energy-efficient and complexity-effective architectures, software-hardware cooperative architectures, decoupled architectures, and multicore architectures. He is a student member of the IEEE.

Daniel Chaver received the BS degree in physics from the University of Santiago de Compostela in 1998 and the BS degree in electrical engineering and the MS and PhD degrees in computer science from the Universidad Complutense de Madrid in 2000, 2002, and 2006, respectively. He is currently an assistant professor of electrical and computer engineering and of computer science at the Universidad Complutense de Madrid. His research interests include various aspects of high-performance computer architecture such as processor microarchitecture and energy-efficient and complexity-effective design. He is part of the Spanish government's research project CYCIT-TIN 2005/5619 and the Hipeac European Network of Excellence.



Michael C. Huang received the BS degree in computer science and engineering from Tsinghua University, Beijing, in 1994 and the MS and the PhD degrees in computer science from the University of Illinois, Urbana-Champaign in 1999 and 2002, respectively. He is currently an associate professor of electrical and computer engineering and of computer science at the University of Rochester. His research interests include various aspects of high-performance computer architecture such as processor microarchitecture, communication and memory substrate, reliability, and energy-efficient and complexity-effective design. His research focuses on addressing emerging issues and exploring new technologies in the underlying device, circuit, and manufacturing technology. He is a recipient of the US National Science Foundation (NSF) CAREER Award. He is a member of the IEEE, the IEEE Computer Society, and the ACM.

Luis Piñuel received the PhD degree in computer science from the Universidad Complutense de Madrid (UCM). He is an associate professor in the Departamento de Arquitectura de Computadores y Automática, UCM. His research interests include computer architecture, high-performance computing, compiling for novel architectures, low-power microarchitectures, embedded systems, and resource management for emerging computing systems. He is a member of the IEEE and the IEEE Computer Society.

Manuel Prieto received the PhD degree in computer science from the Universidad Complutense de Madrid (UCM). He is an associate professor in the Departamento de Arquitectura de Computadores y Automática, UCM. His research is focused on computer architecture and parallel processing, with a special emphasis on emerging architectures, code generation, and optimization. He is a member of the ACM, the IEEE, and the IEEE Computer Society.

Francisco Tirado received the BS degree in applied physics and the PhD degree in physics from the Universidad Complutense de Madrid (UCM) in 1973 and 1977, respectively. He is a professor of computer architecture and technology at UCM. He has worked on computer architecture, parallel processing, and design automation. His current research areas are parallel algorithms and architectures, and processor design. He is a coauthor of more than 200 publications. He has served in the organization of more than 60 international conferences and has also held various positions such as the dean of the Physics Science and Electronic Engineering Faculty, the general manager of the Spanish National Program for Robotics and Advanced Automation, and a member and the chair of the research evaluation committee of Spain. He is currently the director of the Center for Super Computation (CSC) and Madrid Science Park, a member of the Informatics Advisory Board of the UCM, and the adviser of the National Agency for Research and Development (CICYT). He is a senior member of the IEEE, the IEEE Computer Society, and several European institutions and committees.