DMDC: Delayed Memory Dependence Checking through Age-Based Filtering*

Fernando Castro, Luis Pinuel, Daniel Chaver, Manuel Prieto, Michael Huang†, Francisco Tirado
University Complutense of Madrid
fcastror@fis.ucm.es, {lpinuel, dani02, mpmatias, ptirado}@dacya.ucm.es
†University of Rochester
michael.huang@rochester.edu

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 of memory instructions. Traditional CAM-based associative queues can be very slow and energy hungry. In this paper we introduce two new management schemes. The first one is a filtering scheme based on simple age-tracking. This scheme can easily avoid 95-98% of associative load queue (LQ) searches using only a few registers. This translates into significant power savings. More importantly, however, this filtering makes our second scheme, Delayed Memory Dependence Checking (DMDC), practical. With a small hash table, DMDC completely avoids the need for an associative LQ and relies on indexing-based checking at the commit phase and hence cuts the energy spent on LQ by an average of 95%. At an average of about 0.3%, the performance impact is negligible. When the energy cost of the increased execution time is factored in, the processor still makes net energy savings of about 3-8%, depending on the configuration and the applications.

1. Introduction

With high operation 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. This has 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 are allocated in the instructions 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.

This associative search operation is a major concern for the scalability of these queues as not only energy consumption increases, the latency of accesses also worsens with the increase of queue size and may present complications in the logic design. As such, a range of implementations that avoid associative searches have been explored recently. The main observation behind these designs is that memory 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 first introduce a filtering scheme that reduces the associative check frequency of the LQ through explicit tracking of instruction age. (We call the technique YLA-based filtering). Unlike previous filtering proposals, the design's hardware cost is (almost) negligible: only one dedicated global register is enough in the basic implementation. Inspired by this filtering technique, we propose a new LQ management technique (denoted as *delayed memory dependence checking*, or DMDC), which delays the checking for premature loads until the commit phase and is performed in a sequential fashion with only table indexing. As a result, an associative LQ is no longer needed. The design significantly reduces energy consumption to carry out the functionality of the LQ and incurs a very small performance degradation.

The rest of the paper is organized as follows. Section 2 recaps the conventional design of the LQ. Sections 3 and 4

*This work has been supported in part by the Spanish government through the research contract CYCIT-TIN 2005/5619, the Hipeac European Network of Excellence, and by the National Science Foundation through the grant 0509270.

present our YLA-based filtering scheme and delayed memory dependence checking mechanism respectively. Section 5 describes the experimental framework employed, and Section 6 analyzes the experimental results. Section 7 discusses related work. Finally, Section 8 concludes.

2. Conventional Design

Out-of-order microprocessors typically allow early execution of loads for high performance, even if some older 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 parallel with 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 re-execute (or, replay) premature loads and their dependents. To simplify implementation, however, processors typically replay many more instructions (such as all instruction groups following the store [22]), as these premature loads are rare in general and sometimes extra logic is employed to further reduce their occurrence [8].

Coherence The LQ also serves the purpose of maintaining load-load ordering for coherence. A cache-coherent design requires *write serialization*: all writes to the same locations have to appear in the same order to all processors. In a system implementing a relaxed consistency model [8, 22], even though 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 write serialization: (a) load i issues, obtaining some data X; (b) an invalidation message then arrives due to a store from another processor; (c) finally, load j, older in program order than i, issues and obtains the new data Y. This effectively makes the store of Y appears earlier than the store of X. In this case, the common practice is to replay from load i to ensure that it gets newer data.

Detecting this sequence is not trivial, however. In [22], 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, write serialization becomes a standard feature even for uniprocessors.

In summary, in a typical implementation, 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. YLA-Based Filtering

Rationale for age-based filtering Although processors are designed to allow out-of-order issue of memory instructions, in typical programs, the actual sequence of memory instructions is often not wildly different from the program order. When memory instructions happen to issue in program order, many of the queue searches become unnecessary. For example, when a load or 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 altogether. To know that, we can track explicitly the *age* of loads. Specifically, the age of the youngest load issued.

The YLA register We keep a dedicated register to hold the age (*e.g.*, the ROB ID with some simple extension) of the youngest issued load. We call the register *YLA* (Youngest issued Load Age). When a store is resolved, it compares its age with that recorded in YLA. If the store is younger, the LQ search is omitted. We call that a YLA-hit. Otherwise (YLA-miss), a potential violation of memory ordering can exist and the store must search the LQ as usual.

As a load executes, the YLA register is updated if the load is younger than the recorded age. Since this update happens at execution time, loads from wrong paths can corrupt YLA. Though this does not affect the correctness, it affects the filtering effectiveness. Precisely repairing the YLA during misprediction recovery is both difficult and unnecessary. A simple and effective remedy is to reset the YLA register to the branch's age during recovery (if the branch's age is older than the content of YLA).

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.

The idea is to use multiple YLA registers to cover dif-

[1] A store has two operands – the address and the data – which can be separately handled [22]. The store is *resolved* when the address is ready. If only the address of a store is ready, the SQ can not perform forwarding and instead will *reject* the consumer load to issue later [22]. We assume such an implementation in this paper.

ferent 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.

Filtering for stores Finally, we note that the same principle of age-based filtering can be extended to the SQ. For instance, we can use a dedicated register to keep track of the age of the oldest in-flight store. Any load older than the recorded age can bypass SQ searching. Our experiments show that such loads are not rare (about 20%). This suggests that such filtering can be worthwhile. For the rest of the paper, however, we will focus on LQ filtering only.

4. Delayed Memory Dependence Checking

Although the simple age-based filtering is already very effective in reducing the associative searches for the LQ, its main appeal 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. To this end, we have explored one such option that we call *Delayed Memory Dependence Checking* (DMDC).

4.1. Main Idea

The main idea is that for every store, (a) instead of using associative searches of the LQ, we employ a sequential process to inspect each possible load, and (b) we delay the process to commit time.

The high-level procedure consists of 3 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* store and *unsafe* store 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 time.

From a high-level point of view, the reason such a process is viable is two-fold. 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.

4.2. Proposed Implementation

From an implementation point of view, delaying memory dependence violation check 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. Unfortunately, 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. Furthermore, the address and timing information of an unsafe store would have to be moved from the SQ to some special storage after the store has been committed.

We choose a much simpler implementation with two approximations. First, we use approximate timing information of the loads. When a load overlaps with the store in question and may have executed before the store, we conservatively replay the load. This way, we avoid the need to maintain execution timestamps. Second, we map full memory addresses to a hash table called the *checking table*. Two memory accesses are considered overlapping when they hash to the same entry.

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 the store's address bank. We will remember that age, and check every load in between. These loads effectively form a *checking window*.

The architectural support for the design is straightforward. In addition to the YLA registers and the checking table discussed above, we only need a global *end check* register. Furthermore, we no longer need the LQ to be associative. 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. The operation procedure is 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.
- **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.

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 and mark the load as a safe load. 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 [14]. 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 (Figure 1-(a)).

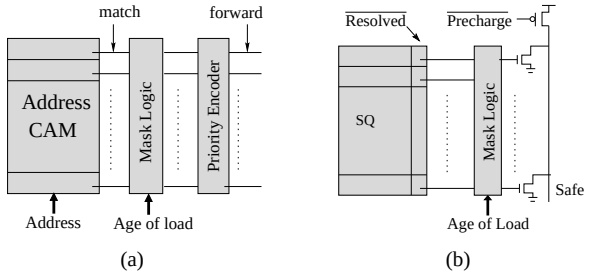

Figure 1. (a) SQ forwarding logic and (b) safe load detection logic.

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 (Figure 1-(b)).

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, thanks to the large number of load and stores identified as safe, the number of false replays is very limited and thus the performance degradation is low.

4.3. 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. A thorough exploration of the design space and necessary remedies is outside the scope of this paper. Here we will only focus on ensuring write serialization as it is required for all cache-coherent systems.

Recall that write serialization 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, even the combination of a subset of the conditions may be infrequent enough that preventing them from happening has insignificant performance implications. For instance, one can perform a replay if an in-flight load overlaps with an invalidation. This essentially enforces sequential consistency. Intuitively, when external invalidation rate is sufficiently low, this is a viable option.

In our system, the timing information of loads is not preserved. Thus, we will need to conservatively perform a replay on the younger load, when two same-location loads are both in-flight while an invalidation happens. 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, which we call the *WRT* bit. When an external invalidation arrives, the checking mode is also activated. But instead of setting the WRT bit, we set the INV bit of the appropriate entries.

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.

When committing a load during the 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, but set the WRT bit instead. This ensures that if a second load to that location is discovered in the checking period, a replay will happen.

4.4. Other Design Options

Local DMDC In our implementation, 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 to the end of another store's checking window. In the worst 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, false replays will become more numerous.

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.

Handling multiple data sizes Unfortunately for memory order tracking, memory accesses are performed at different sizes. Tracking accesses at a coarse granularity (*e.g.*, double-word) will incur unnecessary replays. In this paper, we index the checking table using quad-word address, but use a 4-bit bitmap to discern accesses with a smaller width. An adaptive approach that tracks different widths depending on the address is also possible [11].

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. Experimental Framework

We have evaluated our proposed design using a heavily-modified version of SimpleScalar [4] with the Wattch extension [3]. The modeling of the LSQ is modified to faithfully reflect the state of the art in modern microprocessors. We allow the issues of loads with unresolved older stores. The store queue supports load rejection [22] and rejected load retries later. The processor also handles partial memory matches between memory addresses.

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. Some of the simulation parameters are listed in Table 1. We have experimented with 3 machine configurations to understand the issue of scalability. For brevity, we only report those on *config2*.

The evaluation is performed using all 26 benchmarks from the SPEC CPU2000 suite. For the experiments, we simulate single sim-point regions [20] of one hundred million instructions.

Processor core
*Issue/decode/commit width:*8/8/8 *Functional units:* INT 8+2 mul/div, FP 8+2 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, 1 way, 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, Checking table= 1024
config 2: Issue queue= 48INT/ 48FP, ROB=256, LQ/SQ= 96/48, Registers= 200INT / 200FP, Checking table= 2048
config 3: Issue queue= 64INT/ 64FP, ROB=512, LQ/SQ= 192/64, Registers= 400INT / 400FP, Checking table= 4096

Table 1. Simulation parameters.

6. Evaluation

In this section, we perform some quantitative analysis to further understand the rationale behind age-based mechanisms and the effectiveness of the proposed design. 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).

6.1. YLA-Based Filtering

We first inspect the basic YLA design. As shown in Figure 2, with even a single YLA register, an average of 71% (INT) and 80% (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 Figure 2, with 8 registers, they are about 95-98%.

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 Figure 2, quad-word-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 4 quad-word-interleaved YLA registers. Therefore, we choose to employ two sets of 8 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 8 YLA registers, the reduction in LQ energy is about 32.4%. That translates into about

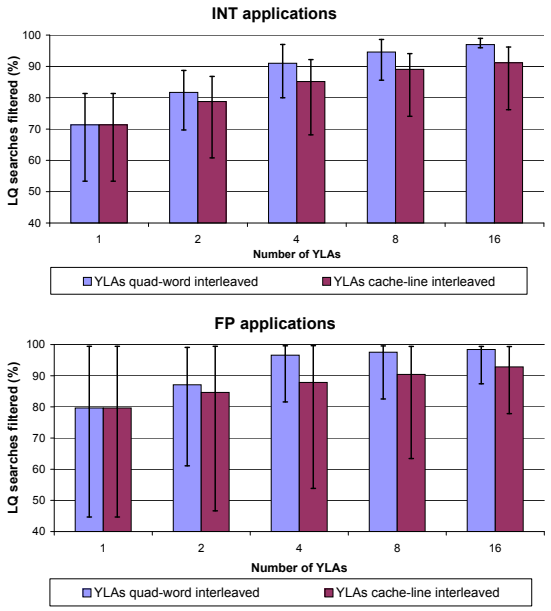

Figure 2. Percentage of safe stores marked using YLA registers with different interleaving. Each bar shows the average percentage of the group of applications, whereas the "I-beams" shows the range of the value within the group of the applications.

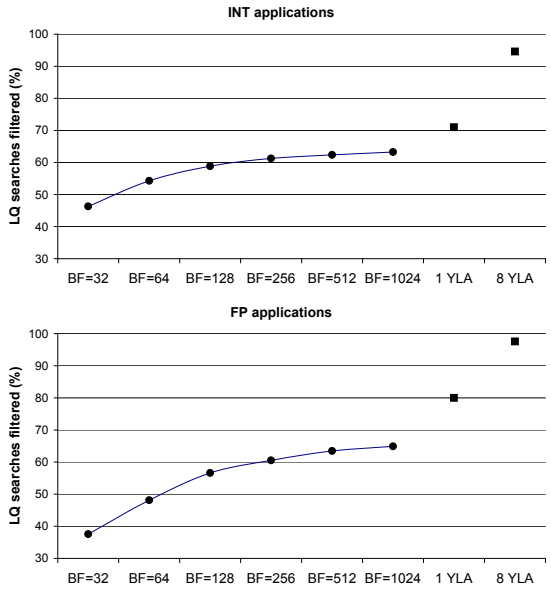

Figure 3. Comparison of the filtering capability of using 1 or 8 YLA registers and bloom filters (BF) with different sizes (H0 hashing function [18] is used).

1.7% 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 (70-80%) of stores. When address interleaving is employed, the effect is quite dramatic – only a few percent of stores are left un-filtered. In comparison, this is much more effective than if only address information is used, such as with a bloom filter [18]. These can be seen from Figure 3.

6.2. DMDC

6.2.1 Main results

We first look at the main results of using a DMDC design to replace the conventional LQ. Figure 4 shows the performance impact and the energy savings (both in the LQ only and processor-wide) in 3 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, about a few percent of the original value. As a result, overall energy reduction in implementing the LQ functionality is about 95-97%, depending on the configuration. Note that we are not considering coherence requirements, which will be dealt with in Section 6.2.4. If that 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 multi-ported, increasing the cost of every search.

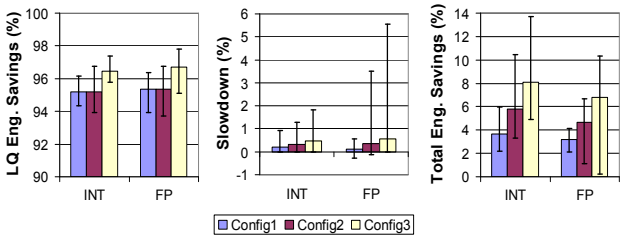

Figure 4. LQ energy savings (a), performance degradation (b), and total processor-wide energy savings of DMDC in 3 processor configurations.

Second, the performance degradation is very limited. The main reason is that false replays as a result of DMDC's approximations are relatively rare. In *config2* for example, the average number of false replays is about 168 and 35 per 1 million committed instructions for INT and FP applications respectively. The worst-case performance degradation is about 1.3% and 3.5% for INT and FP applications, respectively. The performance can increase as well, as seen in the best-case results in the FP applications. This is because in DMDC, without the associative LQ, the limit on the number of in-flight load instructions can be easily made

much higher.

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 result, the energy savings from DMDC become more pronounced. Overall, taking into account the energy overhead from the performance degradation and extra circuit, DMDC can make a net energy savings of about 3-8% depending on the machine configuration.

6.2.2 In-depth analysis

Next, we will discuss various statistics collected during the simulations to better understand the inner working of the mechanism.

Checking window Table 2 shows some statistics of the checking window. On average, a checking window covers about 33 instructions and has 10 loads in between. Out of these, about 4 loads are determined as safe at issue time. The rest needs to go through the checking process. Clearly, with only about 2-5% of stores characterized as unsafe, and each one only needs to be cross-checked with a handful of loads, the sequential process used in DMDC is sufficient to provide the throughput for dependence enforcement.

	instructions	loads	safe loads
INT	33.6	10.3	3.57
FP	33.0	10.1	4.10

Table 2. Number of instructions, loads, and safe loads within a checking window.

On average, the processor spends about 10% and 2.5% of the cycles in checking mode for INT and FP applications respectively. 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, 63% of the windows contain just one unsafe store. In INT applications, this becomes 57%. Having multiple unsafe stores increases the number of entries marked and thus the chance of a load hashing into a marked entry causing a replay. This is part of the reason why INT applications have more false replays due to hashing (more details later).

Safe loads Safe loads are quite numerous. On average, 81% (INT) and 94% (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 non-trivial, about 35-40%. (This is not surprising as the checking mode is triggered only when there are out-of-order executions – and hence non-safe loads.)

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 52% and as high as 97% in integer applications. In other words, without safe loads, the number of replays will double. The simple circuit to detect safe loads is clearly worthwhile. In floating-point applications, the reduction is less significant, about 20%.

False replays Recall that DMDC makes two approximations in the dependence check: 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 timing approximation, a load may be suspected of violating dependence with an older store even though the load actually issued after the store. 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 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.

		Load issued before store	Load issued after store X	Load issued after store Y
INT	Address match	–	109 (65%)	37 (22%)
	Hashing conflict	19 (11%)	0.8 (1%)	2.1 (1%)
FP	Address match	–	11 (32%)	13 (37%)
	Hashing conflict	9.0 (26%)	0.3 (0%)	1.7 (5%)

Table 3. Breakdown of the number of false replays per 1 million committed instructions. The two subcategories are as follows. X: load falls into the "real" checking window of the store. Y: load is checked because multiple checking windows are merged together.

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.

Secondly, with the particular configuration studied (2K-entry checking table), imperfect hashing is not the dominant cause of false replays, accounting for around 11% and 26% 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.

6.2.3 Design options

Local vs. global In local DMDC, we extend the microarchitecture to locally record checking window for every unsafe store. This way, the checking windows are less likely to overlap to form bigger ones. The effect is shown in Table 4: windows are 13-25% shorter and contain proportionally fewer loads. The percentage of safe loads, however, reduces faster. This is expected as the shrinking windows

are more likely to exclude safe loads.

	instructions	loads	safe loads
INT	25.3	7.92	2.27
FP	28.9	8.61	3.01

Table 4. Number of instructions, loads, and safe loads within a checking window in local DMDC implementation.

The main benefit of having small windows is having more chances to clear the table to avoid unnecessary false replays. To that end, the local DMDC approach is quite effective. The average number of replays per 1 million committed instructions reduces from 168 to 134 (by 20%) for integer codes and from 35.4 to 23.7 (by 33%) for floating-point codes. Table 5 shows the breakdown of false replays, which can be compared to Table 3. Although the statistics are imperfect for pinning down exactly which replays are avoided[2], they do suggest that false replays due to overlapping windows (the Y column) are indeed mitigated.

		Load issued before store	Load issued after store X	Y
INT	Address match	–	91 (68%)	21 (15%)
	Hashing conflict	20 (15%)	1.1 (1%)	1.4 (1%)
FP	Address match	–	11 (45%)	0.6 (2%)
	Hashing conflict	10 (43%)	0.7 (3%)	1.6 (7%)

Table 5. Breakdown of the number of false replays (per million committed instructions) in local DMDC.

Overall, because false replays are already rare even in global DMDC, the difference in energy and performance between global and local versions of DMDC is not dramatic, as can be seen in Figure 5. The local version moderately improves the effectiveness at the expense of a small increase in design complexity. One thing worth noting is that under local DMDC, the worst-case performance degradation of any single application is noticeably lower, especially in FP applications.

Associative queue vs. hashing table Instead of using a hash table, we can 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 (2K entries) is equivalent

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

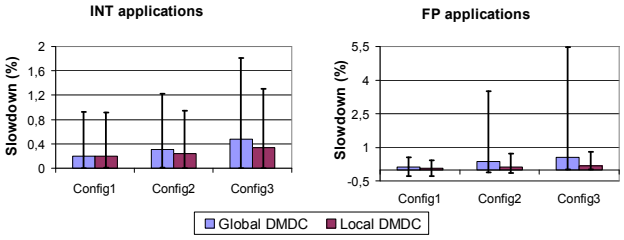

Figure 5. The comparison of slowdown between local and global DMDC.

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.

6.2.4 The impact of invalidations

In this paper, our focus is uni-processors or small-scale multi-core systems running single-threaded applications. Thus, as mentioned earlier, the discussion so far does not include the effect of coherence activities. The conventional baseline configuration also does not consider coherence.

While a detailed analysis in a true multiprocessing environment is left for future work, we have performed some experiments using injected random invalidations at certain rates. The changes to a range of statistics are recorded and summarized in Table 6. Despite the imperfection in this methodology, we can still obtain some insights.

	Invalidations per 1000 cycles	0	1	10	100
INT	% cycles in checking mode	10	10.3	12.2	23.2
	Relative checking window size	1	1.01	1.11	1.37
	Relative false replay rate	1	1.1	1.47	4.59
	Slowdown	0.31	0.34	0.46	1.36
FP	% cycles in checking mode	2.5	3.0	7.8	27.7
	Relative checking window size	1	1.25	1.92	3.49
	Relative false replay rate	1	1.36	1.72	5.35
	Slowdown	0.36	0.38	0.48	1.16

Table 6. Changes to key statistics under external invalidations of different rates.

From the table, we can see that with up to 10 invalidations per 1000 cycles, the increase in all statistics remain moderate. Clearly, the design is capable of handling such traffic. When invalidations are as frequent as 1 every 10 cycles, the system *starts* to show sign of stress: false replays are about 5 times higher than without any invalidations and the slowdown is also becoming more noticeable. However, with the slowdown still around only 1%, the design is perhaps still acceptable. In environments with invalidation rate higher than 1 per 10 cycles, additional mechanisms to filter

these invalidations become desirable.

7. Related Work

In recent years, many schemes are designed to improve the conventional LSQ. Most designs still consist of two separate logic blocks (corresponding to the SQ and LQ), one handles forwarding at the load issue time, the other verifies the correctness of the earlier forwarding. Some schemes continue to rely on associative queues in these two blocks, but cut down their access frequency using, for example, filtering. Others use alternative circuit structure, often indexed queue, to replace or augment the associative queues.

Sethumadhavan et al. [18] propose to use *bloom filters* 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. 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). In terms of the effect, it is very similar to Cain and Lipasti's work [5], though the implementation is very different. DMDC still uses the conventional approach of ruling out dependence or coherence violation through address and timing information, albeit with approximations. The value-based approach ignores that information and only uses the value information. The downside of the approach is the elevated memory bandwidth requirement. To reduce this requirement, timing and address information can no longer be ignored as it enables effective filtering [5, 17].

A central enabling factor for 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 powerful techniques. Roth also explores age information in [17]. 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 re-execution 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 re-execution. 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. As such, our age information tracks execution timing, whereas theirs essentially tracks the commit time of a store.

Garg et al. replace the associative LQ with a hash table explicitly tracking age information [11]. Each entry of the tables records the youngest load executed whose address hashes into the entry. Upon store's execution, if the age recorded in the entry is younger, a replay is triggered. DMDC improves upon this design in important ways. Rather than using one table to maintain age and (partial) address information as in [11], we use a two-step approach with decoupled data structures. Only a few age registers are used to store timing information. A separate hash table only encodes address information without the need to store age information, which costs more bits. This is not only hardware-efficient, but also energy-efficient. Furthermore, only a much smaller set of loads and stores access the checking table to detect possible violations, reducing energy even more. Finally, delaying the process to commit time not only simplifies circuitry, but also naturally avoids pollution to the checking table.

In addition to the work discussed above, two other papers also optimize the LQ. This is done with dependence prediction [6] or with the help from software analysis to mark loads guaranteed not to cause dependence violation [13].

While our work optimizes the verification logic, another set of related work optimizes the forwarding logic. Many proposals rely on memory dependence prediction [7, 15] to narrow the range of stores to forward from. Park et al. [16] 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 [1, 2, 10, 23] 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 [19, 21] or with software support using a feedback-directed approach [9].

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 re-execution to provide a correctness guarantee [12].

8. Conclusions

In this paper, we have introduced an alternative to CAM-based LQs. The design is based on a new age-based filtering mechanism. The central observation behind the proposed design is that even with out-of-order execution, a significant majority of loads and stores demonstrate partial ordering. With the presence of the associative SQ, this partial ordering can rule out store-load replays in a large majority of cases. This makes a fully-associative LQ an overkill.

Age-based filtering is done using a few address-interleaved registers to keep track of the age of youngest loads issued. These registers can filter out 95-98% of

stores from further dependence-violation checking. For the remaining stores, these registers delineate the window of loads that need further inspection. Thanks to the effective filtering, the remaining loads and stores can be easily inspected using a sequential checking process, thereby eliminating the need for a large, fully-associative LQ. Furthermore, for design simplicity, this process is delayed to the commit time. We call this Delayed Memory Dependence Checking (DMDC).

We have explored a specific implementation of DMDC using a hash table to communicate address information between loads and stores. In addition to detecting memory dependence violation, DMDC can also enforce coherence and consistency requirements. The architectural support is straightforward and achieves the functionality of the conventional LQ at merely 5% of its energy cost. The design incurs only a very small number of false replays. The performance degradation, at 0.3%, is negligible. The overall processor-wide energy savings ranges from 3-8% depending on the configuration and the application suite.

In addition to the basic design, we have also performed in-depth analyses, optional design optimizations, and a study on the impact of external invalidation messages. Specifically, we found that for a small-scale bus-based multiprocessors, the proposed design is capable of handling a moderate level of invalidation traffic (up to 1 per 10 cycles). A more systematic study of the dependence checking logic in multiprocessor domain is our future work.

References

[1] H. Akkary, R. Rajwar, and S. Srinivasan. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. In *International Symposium on Microarchitecture*, pages 423–434, Dec. 2003.

[2] L. Baugh and C. Zilles. Decomposing the Load-Store Queue by Function for Power Reduction and Scalability. *IBM Journal of Research and Development*, 50(2-3):287–298, 2006.

[3] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *International Symposium on Computer Architecture*, pages 83–94, June 2000.

[4] D. Burger and T. Austin. The SimpleScalar Tool Set, Version 2.0. Technical report 1342, Computer Sciences Department, University of Wisconsin-Madison, June 1997.

[5] H. Cain and M. Lipasti. Memory Ordering: A Value-based Approach. In *International Symposium on Computer Architecture*, pages 90–101, June 2004.

[6] F. Castro, D. Chaver, L. Pinuel, M. Prieto, M. Huang, and F. Tirado. Load-Store Queue Management: an Energy Efficient Design based on a State Filtering Mechanism. In *International Conference on Computer Design*, Oct. 2005.

[7] G. Chrysos and J. Emer. Memory Dependence Prediction Using Store Sets. In *International Symposium on Computer Architecture*, pages 142 –153, June–July 1998.

[8] Compaq Computer Corporation. *Alpha 21264/EV6 Microprocessor Hardware Reference Manual*, Sept. 2000. Order number: DS-0027B-TE.

[9] C. Fang, S. Carr, S. Onder, and Z. Wang. Feedback-Directed Memory Dismabiguation Throuh Store Distance Analysys. In *International Conference on Supercomputing*, June 2006.

[10] A. Gandhi, H. Akkary, R. Rajwar, S. Srinivasan, and K. Lai. Scalable Load and Store Processing in Latency Tolerant Processors. In *International Symposium on Computer Architecture*, pages 446–457, June 2005.

[11] A. Garg, F. Castro, M. Huang, L. Pinuel, D. Chaver, and M. Prieto. Substituting Associative Load Queue with Simple Hash Table in Out-of-Order Microprocessors. In *International Symposium on Low-Power Electronics and Design*, Oct. 2006.

[12] A. Garg, M. Rashid, and M. Huang. Slackened Memory Dependence Enforcement: Combining Opportunistic Forwarding with Decoupled Verification. In *International Symposium on Computer Architecture*, pages 142–153, June 2006.

[13] R. Huang, A. Garg, and M. Huang. Software-Hardware Cooperative Memory Disambiguation. In *International Symposium on High-Performance Computer Architecture*, pages 248–257, Feb. 2006.

[14] S. Meier. Store Queue Multimatch Detection, Feb. 2003. US Patent No. 6,523,109.

[15] A. Moshovos, S. Breach, T. Vijaykumar, and G. Sohi. Dynamic Speculation and Synchronization of Data Dependences. In *International Symposium on Computer Architecture*, pages 181–193, June 1997.

[16] I. Park, C. Ooi, and T. Vijaykumar. Reducing Design Complexity of the Load/Store Queue. In *International Symposium on Microarchitecture*, pages 411–422, Dec. 2003.

[17] A. Roth. Store Vulnerability Window (SVW): Re-Execution Filtering for Enhanced Load Optimization. In *International Symposium on Computer Architecture*, pages 458–468, June 2005.

[18] S. Sethumadhavan, R. Desikan, D. Burger, C. Moore, and S. Keckler. Scalable Hardware Memory Disambiguation for High ILP Processors. In *International Symposium on Microarchitecture*, pages 399–410, Dec. 2003.

[19] T. Sha, M. Martin, and A. Roth. Scalable Store-Load Forwarding via Store Queue Index Prediction. In *International Symposium on Microarchitecture*, Dec. 2005.

[20] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, Oct. 2002.

[21] S. Stone, K. Woley, and M. Frank. Address-Indexed Memory Disambiguation and Store-to-Load Forwarding. In *International Symposium on Microarchitecture*, Dec. 2005.

[22] J. Tendler, J. Dodson, J. Fields, H. Le, and B. Sinharoy. POWER4 System Microarchitecture. *IBM Journal of Research and Development*, 46(1):5–25, Jan. 2002.

[23] E. Torres, P. Ibanez, V. Vinals, and J. Llaberia. Store Buffer Design in First-Level Multibanked Data Caches. In *International Symposium on Computer Architecture*, June 2005.