# Implementation Issues of Slackened Memory Dependence Enforcement
## Technical Report

Alok Garg, M. Wasiur Rashid, and Michael Huang
Department of Electrical & Computer Engineering
University of Rochester
{garg, rashid, michael.huang}@ece.rochester.edu

## Abstract

*An efficient mechanism to track and enforce memory dependences is crucial to an out-of-order microprocessor. The conventional approach of using cross-checked load queue and store queue, while very effective in earlier processor incarnations, suffers from scalability problems in modern high-frequency designs that rely on buffering many in-flight instructions to exploit instruction-level parallelism. In this paper, we make a case for a very different approach to dynamic memory disambiguation. We move away from the conventional exact disambiguation strategy and adopt an opportunistic method: we allow loads and stores to access an L0 cache as they are issued out of program order, hoping that with such a laissez-faire approach, most loads actually obtain the right value. To guarantee correctness, they execute a second time in program order to access the non-speculative L1 cache. A discrepancy between the two executions triggers a replay. Such a design completely eliminates the necessity of real-time violation detection and thus avoids the conventional approach's complexity and the associated scalability issue. We show that even a simplistic design can provide similar performance level achieved with a conventional queue-based approach with optimistically-sized queues. When simple, optional optimizations are applied, the performance level is close to that achieved with ideally-sized queues.*

## 1 Introduction

Due to their limited ability of buffering in-flight instructions, current out-of-order processors fail to hide very long latencies such as those of off-chip memory accesses. This limitation stems from the fact that scaling up buffering structures usually incur latency and energy penalties that undermine the overall design goal. With the popularization of multi-core products, the precious off-chip bandwidth will be subject to more contention and in turn further exacerbate memory latencies. Thus, latency tolerance will continue to be an important design goal of microarchitecture.

In this paper, we focus on the memory disambiguation and forwarding logic. To orchestrate the out-of-order execution of a large number of in-flight instructions, an efficient mechanism to track and enforce memory dependency is imperative. The conventional approach uses age-based queues – often collectively referred to as the load-store queue (LSQ) – to buffer and cross-compare memory operations to ensure that the out-of-order execution of memory instructions does not violate program semantics under a certain coherence and consistency model. The LSQ buffers memory updates, commits their effects in-order, forwards values between communicating pairs of load and store, and detects incorrect speculation or potential violation of coherence and consistency. Due to the complex functionality it implements and the fact that the address for a load or a store needs to be matched against those of the opposite kind in an associative manner and matching entries need to go through time-consuming priority logic, the LSQ is perhaps the most challenging microarchitectural structure to scale up.

We propose a novel design that moves away from the conventional load-store handling mechanism that strives to perform most accurate forwarding in the first place and proactively detect and recover from any dependence violation. Instead, we adopt a very passive, "slackened" approach to forwarding and violation detection. First, we use an L0 cache to perform approximate, opportunistic memory forwarding at the execution stage, hoping that most loads would actually get correct data. In this stage, there is minimum interference to instruction execution. Second, we also avoid relying on proactive monitoring to detect violation. Our correctness guarantee comes from an in-order re-execution of memory accesses. Discrepancy between the two different executions triggers a replay. Such a decoupled approach not only makes it easier to understand, design, and optimize each component individually, it is also effective. Without any associative search logic, in a processor that can buffer 512 in-flight instructions, even a naive implementation of our design performs similarly with a traditional LSQ-based design with optimistically-sized queues. The performance is further improved by using optional optimizations and approaches that of a system with ideally-scaled LSQ: on average, about 2% slower for integer applications and 4.3% slower for floating-point applications. Furthermore, both naive and optimized design scale very well.

The rest of the paper is organized as follows: Section 2 recaps the basics of the memory dependence logic and highlights recent optimization proposals; Section 3 describes the basic design and optional optimization techniques; Section 4 describes the experimental methodology; Section 5 provides some quantitative analysis; and Section 6 concludes.

## 2 Background and Related Work

Out-of-order processors strive to maximize instruction-level parallelism (ILP) by aggressively finding future ready instructions to execute. To maintain sequential program semantics, data dependences have to be preserved. While register-based dependences are tracked explicitly via physical register IDs, memory-based dependence tracking is much less straightforward because without calculating the effective address, it is almost impossible for the hardware to figure out the dependence relationship. To make things even more complicated, an execution order of loads and stores different from the program order may result in a different outcome of a parallel program even though from one processor's perspective, the reordering does not appear to violate sequential semantics. On the other hand, strictly enforcing program order for memory instructions would significantly hamper an out-of-order processor's capability of extracting ILP. Thus, most implementations allow load store reordering, allow load's execution in the presences of unresolved earlier stores, and employ a dynamic mechanism to ensure that the execution order preserves program semantics.

### 2.1 Conventional Queue-Based Mechanism

In a typical out-of-order core, the result of a store instruction is not released to the memory hierarchy until the store commits so as to easily support precise exception. A load instruction, on the other hand, is allowed to access the memory hierarchy as soon the address is available. In a uniprocessor system, this speculative (potentially premature) access will return correct data as long as no in-flight stores earlier than the load will modify any part of the data being loaded. If there is a store with the same address as the load, then there are the following three possibilities by the time the load executes and accesses the cache: 1. Both the address and the data of the store are available. The load can then use the data directly (called load forwarding), ignoring the data fetched from the cache. 2. Only the address of the store is available. Then the load has to wait until the store data is available for forwarding. 3. The address of the store is unknown yet. Then the fact that the data in the cache is stale can only be detected at a later time when the store's address is known. (Alternatively, to avoid dealing with this situation, the load can be held off until the addresses of all prior stores are resolved, which can unnecessarily delay independent and otherwise ready loads.) The hardware has to handle all these possibilities correctly.

In current implementations, this is done using a circuitry generally referred to as the Load-Store Queue (LSQ), which is usually divided into a separate load queue (LQ) and a store queue (SQ). When a load executes, parallel to the cache access, its address is checked against those of older stores in the SQ. Among all the matches, the youngest store is the producer and its data, if available, should be forwarded to the load. If the data is not available, the load is rejected and retried later [22]. Conversely, when a store executes, the address is checked against those of younger loads to find out if any has executed prematurely. All premature loads and all their dependents need to be squashed and re-executed. In a practical design, the processor squashes all instructions from the oldest premature load (or even from the triggering store) onward. This is often called a *replay* [7].

Though conceptually straightforward, current designs of (age-based) LSQ are complex and hard to scale. A number of factors hinder the scalability. First, the queues are priority CAMs. Not only is an address used to search through the queue in a fully associative manner, the match results also have to go through a priority logic in order to pin-point the oldest or the youngest instance. Second, the searches in the store queue are on the timing critical path of load execution, limiting its latency. Furthermore, unlike in a set-associative cache where a virtually-indexed, physically-tagged design can be easily implemented to hide the address translation latency, the search in the fully-associative SQ is serialized with address translation. Third, the design is further complicated due to the handling of coherence and consistency with other processors or I/O devices, or corner cases such as partial overlaps of operands.

### 2.2 Highlight of Optimized and Alternative Designs

Recognizing the scalability issue of the LSQ, many different proposals have emerged recently. Due to the complex nature of the technical detail and the size of the body of work, we only attempt to highlight the solutions here, rather than to discuss the nuances. We will contrast this body of work and our proposal later in Section 3.3.

A large body of work adopts a two-level approach to disambiguation and forwarding. The guiding principle is largely the same. That is to make the first-level (L1) structure small (thus fast and energy efficient) and still able to perform a large majority of the work. This L1 structure is backed up by a much larger second-level (L2) structure to correct/complement the work of the L1 structure. The L1 structure can be allocated according to program order or execution order (within a bank, if banked) for every store [1, 8, 23] or only allocated to those stores predicted to be involved in forwarding [3, 16]. The L2 structure is also used in varying ways due to different focuses. It can be banked to save energy per access [3, 16]; it can be filtered to reduce access frequency (and thus energy) [1, 18];

or it can be simplified in functionality such as removing the forwarding capability [23].

In general, if the L1 structure handles a large portion of the task without accessing the L2, then the energy consumption and latency of the L2 structure is certainly less critical. However, unlike a cache hierarchy, a disambiguation/forwarding logic has an important difference that makes a hierarchical structure less effective. The L1 structure does not have a fundamental filtering effect for the L2 structure: to perform exact disambiguation, one needs to check *all* in-flight stores, even if there is a match in the L1. A notable exception is when entries in L1 are allocated according to program order. In that case, a matching store in the L1 "shields" all other stores in the L2. Unfortunately, this requirement can prevent optimizations that maximize the hit rate of L1.

Another body of work only uses a one-level structure (for stores) but reduces check frequency through clever filtering or prediction mechanisms [15, 18]. In [18], a conservative membership test using bloom filter can quickly filter out accesses that will not find a match. In [15], only loads predicted to need forwarding check the SQ. The safety net is for stores to check the LQ to find mis-handled loads at the commit stage.

Value-based re-execution presents a new paradigm for memory disambiguation. In [6], the LQ is eliminated altogether and loads re-execute to validate the prior execution. Notice that the SQ and associated disambiguation/forwarding logic still remain. Filters are developed to reduce the re-execution frequency [6, 17]. Otherwise, the performance impact due to increased memory pressure can be significant [17].

Research has shown that dependence relationship between static load and store instructions shows predictability [13, 14]. Prediction allows one to focus on a smaller set of stores in the disambiguation and forwarding logic. In [19], if a producer can be pin-pointed, the load forwards directly from the store's SQ entry. If there are several potential producers, however, another predictor provides a *delay index* and instructs the load to wait until the instruction indicated by the delay index commits. In [20], loads predicted to be dependents of stores are delayed. When allowed to execute, it accesses a *store forwarding cache* (SFC) and a *memory dependence table*. This design avoids the need to exactly predict the identity of the producer as memory addresses help to further clarify the producer.

Finally, software analysis is shown to be effective in helping to reduce disambiguation resource pressure [10].

# 3 Slackened Memory Dependence Enforcement (SMDE)

In an attempt to simplify the design of memory dependence enforcement logic, we adopt a different approach from conventional designs and recent proposals. Instead of achieving the two goals of high performance and correctness by encouraging a highly out-of-order execution of memory instructions and *simultaneously* imposing a proactive monitoring and interference of the execution order, we use two decoupled executions for memory instructions, each targeting one goal: a highly out-of-order *front-end* execution with little enforcement of memory dependency to keep the common-case performance high, and a completely in-order *back-end* execution (with no speculation) to detect violations and ensure program correctness. Separating the two goals allows a much more "relaxed" implementation (hence the name) which not only mitigates the escalating microprocessor design complexity, but can lead to opportunities a more sophisticated and proactive approach can not provide. For example, our design can effortlessly allow an arbitrarily large number of in-flight memory instructions. Such benefits can offset the performance loss due to the simplicity of the design.

In the following, we first describe the basic architectural support to ensure functional correctness of the design. This results in a naive, simplistic design that is hardly efficient. Yet, as we will show later, due to the ability to allow any number of memory instructions to be in-flight at the same time, even this naive version of SMDE can perform similarly as a system with optimistically-sized LSQ (but is otherwise scaled up to buffer more in-flight instructions). We then discuss simple, optional optimization techniques that address some of the performance bottlenecks. With these optimizations, the design can perform close to idealized LSQ.

## 3.1 A Naive Implementation of SMDE

Figure 1 is the block diagram of the naive design, which shows the memory hierarchy and high-level schematic of the processor pipeline. For memory instructions, a front-end execution is performed out of program order like in a normal processor at the execution stage. However, in a naive SMDE, memory instructions are issued *entirely* based on their register dependences. No other interference is imposed. As a result, it is possible to violate memory dependences and this makes the front-end execution fundamentally speculative (and it is treated as such). Thus the L0 cache accessed in the front-end execution does not propagate any result to L1. To detect violations, memory accesses are performed a second time, totally in-order at the commit stage of the pipeline. Any load that obtains different data from the two executions will take the result of the back-end execution and trigger a squash and replay of subsequent instructions.

From one perspective, the only execution in our design that is absolutely required is the back-end execution. In theory, therefore, *any* front-end execution scheme would work (even if it only returns garbage values). This relieves the

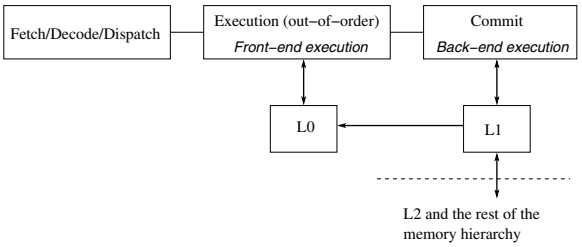



**Figure 1.** Diagram of the pipeline and memory hierarchy of the proposed design.

design burden to maintain correctness for the front-end execution, effectively relegating it to a value predictor. When we simply issue memory instructions based on the register dependences and completely disregard their memory dependences, the front-end execution effectively becomes a slow but very accurate value predictor: on average, about 98% of loads obtain a correct value.

From another perspective, the front-end execution can afford to simplify the memory dependence enforcement logic because the back-end execution provides a safety net for incorrect speculation. We push that simplification to the extreme by completely eliminating any enforcement logic. Such a laissez-faire approach works for normal programs as our empirical observation shows that about 99% of loads happen in the "right" time, that is, after the producer has executed, but before the next writer executes.

The primary appeal of SMDE is its simplicity. The entire framework is conceptually very straightforward and it is rid of the conventional LSQ using priority CAM logic or any form of memory dependence prediction, thus avoiding the scalability problem or circuit complexity issues. Furthermore, the two execution passes are entirely decoupled, making the design modular. The dedicated back-end execution offers a large degree of freedom to the design of the front-end execution. Any rare incidents such as race conditions, corner cases, or even concerns for soft errors can be completely ignored for design simplicity. We will discuss examples later.

#### 3.1.1 Front-end execution

Central to our architectural support is an unconventional, speculative L0 cache. As explained earlier, at the issue time of a load, we simply access this L0 cache. Meanwhile, we also allow stores to write to this L0 cache as soon as they execute. Since the L0 cache is used to handle the common cases, we want to keep its control extremely simple. No attempt is made to clean up the incorrect data left by wrong-path instructions. When a line is replaced, it is simply discarded, even if it is dirty. And in fact, no dirty bit is kept, nor is the logic to generate it. Keeping track of and undoing wrong-path stores and preventing cast-outs undoubtedly complicate the circuitry and undermine the principle of keeping the common case simple and effective.

Initially, it would appear that such an unregulated L0 cache may be quite useless as load may access the cache before the producer store has written to the cache or after a younger store has, and the data could be corrupted in numerous ways: by wrong-path instructions, due to out-of-order writes to the same location, and due to lost updates by eviction. However, our value-driven simulation shows that an overwhelming majority (98% on average) of loads obtain correct value. Upon closer reflection, this is not as surprising as it appears. First, compilers do a good job in register allocation and memory disambiguation. This means that close-by store-to-load communications are infrequent in normal programs. Far apart store-load pairs are very likely to execute in the correct order. Second, write-after-read (WAR) violations are also infrequent: a load followed closely by a same-location store are very likely to be part of a load-operate-store chain, whose execution order will be enforced by the issue logic. Third, data corruptions in the L0 cache are not permanent – they are naturally cleansed by new updates and evictions.

Recall that in a conventional LSQ, forwarding data from SQ needs associative search followed by priority selection. Additionally, the virtual address needs to be translated before used to search the SQ, otherwise, the forwarding can be incorrect due to aliasing. However rare it is, a partial overlap (where a producer store only writes part of the loaded data) has to be detected. Therefore the logic has to be there and it is invoked every time the queue is searched. In stark contrast, the L0 is simply accessed like a cache. As mentioned before, thanks to the back-end execution, the concern about these corner case situations is that of performance, not of correctness, and we can safely ignore them in the front-end.

#### 3.1.2 In-order back-end execution

In-order re-execution to validate a speculative execution is not a new technique [6, 9]. However, the extra bandwidth requirement for re-execution makes it undesirable or even impractical. Prior proposals address this issue by monitoring the earlier, speculative execution order, reasoning about whether a re-execution is necessary, and suppressing unnecessary re-executions [6, 9, 17].

A small but crucial difference between our design and this prior approach is that we do not rely on avoiding re-execution. Instead, our dual-cache structure naturally and easily provides the bandwidth needed for the re-execution. Although misses from L0 still access L1 and thus increase the bandwidth demand, this increase is (at least partly) offset by the fact that most wrong-path loads do not access L1 for back-end execution.

The benefit of faithful reload without any filtering is concrete, especially in enforcing coherence and consistency. For modern processors, cache coherence is a must (even in uniprocessor systems, as requiring the OS to maintain cache

coherent with respect to DMA operation is undesirable). Regardless of how relaxed a memory consistency model the processor supports, cache coherence alone requires careful handling to ensure store serialization: if two loads accessing the same location are re-ordered and separated by an invalidation to that location, the younger load must be replayed. Hence the memory dependence enforcement logic needs to heed every invalidation message, not just those that reach the L1 cache (as cache inclusion does not extend to in-flight loads). Faithfully re-executing every memory access in-order greatly simplifies coherence and consistency considerations.

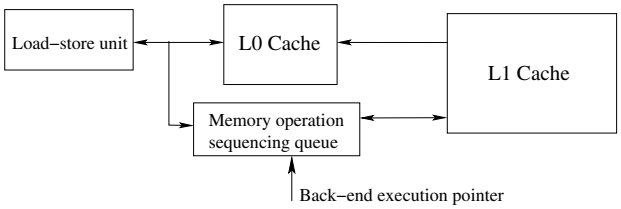



**Figure 2.** In-order back-end execution via the memory operation sequencing queue.

Figure 2 shows the relationship of the logical structures involved in back-end execution. An age-ordered *memory operation sequencing queue* (MOSQ) is used to keep the address and data of all memory operations generated during the front-end execution. During back-end execution, the recorded address is used to access the L1. Note that the back-end execution is not a full-blown execution, but only a repeated memory access. There is no re-computation of the address, hence no need for functional units or register access ports. For loads, the data returned from the L1 is compared to that kept in the MOSQ. A difference will trigger a replay. A successful comparison makes the load instruction ready for commit. For stores, the data kept in the MOSQ is written to the L1 cache when the store is committed. The back-end execution is totally in-order, governed by the execution pointer shown in Figure 2. Note that for store instructions, re-execution can only happen when they are committed. Essentially, when pointing to a store, the back-end execution pointer (BEEP) is "clamped" to the commit pointer. For load instructions, however, BEEP can travel ahead of the commit pointer allowing reload to happen before commit.

### 3.1.3 Scope of replay

The scope of instructions to be replayed is similar to that in a conventional design [7] with an important difference. We do not re-execute the load instruction that triggers the replay. Instead, we only re-execute instructions younger than the load. The reason is two-fold (we *can* and we *must*). First, unlike in a conventional system where the detection of a memory order violation merely suggests that the load obtained potentially incorrect data without providing the right result, in our system, the value from the reload is always correct and can be used to fix the destination register of the load instruction. Second, it is necessary *not* to re-execute the load to avoid the rare but possible infinite loop. When a load is replayed in a conventional system, it will eventually become safe from further replay (*e.g.*, when it becomes the oldest in-flight memory instruction). Therefore, forward progress is always guaranteed. In our system, we do not place *any* constraint on the front-end execution and thus can not expect any load to be correctly processed regardless of how many times it is re-executed. Furthermore, a replay is triggered by the difference between two loads to two different caches at different time. Fundamentally, there is no guarantee that the two will be the same. In the pathological scenario, disagreement between the two can continue indefinitely. For example, when another thread in the parallel program is constantly updating the variable being loaded.

The replay trap is handled largely the same way as in a conventional design [7]. The only additional thing to perform is to fix the destination register of the triggering load.

### 3.1.4 L0 cache cleanup

In addition to the basic operation of replay, we can perform some *optional* cleanup in the L0 cache. We emphasize again that we do not require any correctness guarantee from the front-end execution, so the difference between cleanup policies is that of timing, not correctness.

Intuitively, by the time a replay is triggered, the L0 cache already contains a lot of future data written by in-flight stores that will be squashed. Thus we can invalidate the entire cache and start afresh. This eliminates any incorrect data from the L0 cache and reduces the chance of a future replay. The disadvantage, of course, is that it also invalidates useful cache lines and increases the cache miss rate. This has its own negative effects, especially in modern processors, which routinely use speculative wakeup of dependents of load instructions. When a load misses, this speculation fails and even unrelated instructions may be affected. For example, in Alpha 21264, instructions issued in a certain window following a mis-speculated load will be squashed and restart the request for issue [7]. Thus, whole-cache invalidation can be an overkill.

A second alternative, single-flush, is to only flush the line accessed by the replay-triggering load. The intuition is that this line contained incorrect data at the time the load executed and probably still contains incorrect data. This approach stays on the conservative side of L0 cleanup and thus will incur more replays than whole-cache invalidation. However, it does not affect the cache miss rate as much and it is simpler to implement.

A third alternative, which we call selective flush, comes in between the two extremes. In addition to flushing the line that triggered the replay, we can invalidate all the cache lines written to by the squashed stores. This can be done by

traversing the MOSQ and use the address of the stores that have finished execution to perform the invalidation.

For implementation simplicity we use single-flush. Note that not flushing any line is also an option, but the replay rate is much higher than single-flush, which makes its unattractive. We will show relevant statistics of different replay strategies in Section /refsec:eval. As we will see, single-flush actually outperforms other alternatives.

### 3.2 Performance Optimizations

In the naive implementation of SMDE, we select the most straightforward implementation in each component. Such a simplistic design leaves several opportunities for further optimization. We discuss a few techniques here.

#### 3.2.1 Reducing replay frequency with the fuzzy disambiguation queue

Although an incorrect load using the L0 cache will not affect program correctness, it does trigger a replay which can be very costly performance-wise. Hence, simple mechanisms that help reduce replay frequency may be desirable. Our analysis shows that a large majority of replays are triggered directly or indirectly (a replay rolls back the processor and thus leaves some future data in the L0 cache which can cause further replay) by violation of read-after-write ordering. In these cases, oftentimes, while the data of the producer store is unavailable, the address is ready. In other words, if we keep track of address information, we can reject some premature loads as in a conventional design [22] and reduce the replay rate. For this purpose, we introduce a *Fuzzy Disambiguation Queue* (FDQ).

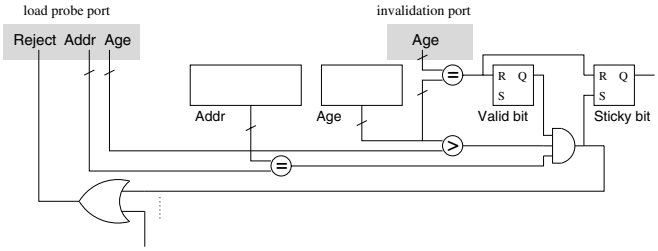



**Figure 3.** The fuzzy disambiguation queue. For clarity, the figure shows one entry of the queue with one load probe port and one invalidation port.

Each entry of the FDQ contains the address of a store and the age of the instruction (Figure 3). A straightforward representation of the age is the ROB entry ID plus one or two extra bits to handle the wrap-around[1]. Since we only keep track of stores between address generation and execution, we allocate an entry at address generation and deallocate it when the data is written to the L0 cache.

When a load executes, it sends the address and the age through one *load probe port* to probe the FDQ. Out of all the valid entries, if any entry matches the address of the load and has an older age, we reject the load (Figure 3), which will be retried after a certain delay. At the same time a *sticky bit* is set to prevent the displacement of the matching entry. We do this because given limited space, we want to maximize the number of "useful" entries in the queue. When we allocate an entry in the FDQ, if there is no invalid entry, we randomly evict a valid entry to make room for the new store. If all entries have the sticky bit set, no entry is allocated. When a store executes, it attempts to deallocate its FDQ entry, if any. This is done by sending its age through the *invalidation* port to clear the valid bit of any matching entry (Figure 3). When a branch misprediction is detected, we can flash-clear all valid bits to eliminate orphan entries (entries whose owners have been squashed). These orphan entries must be eliminated because once they match a load, they will perpetually reject the load causing a deadlock. We note that for a slightly more complex circuit, we can use the age of the mispredicted branch to selectively invalidate only entries younger than the branch. This invalidates all the orphan entries but keeps other entries intact. Finally, if during issue, stores also probe the FDQ, we can similarly reject "premature" stores to prevent write-after-write violations.

Note that an FDQ differs significantly from an SQ in numerous ways. An FDQ does not have a priority logic: We are only interested in *whether* there is an older store to the same location pending execution, not *which* store. There is no data forwarding logic in an FDQ either. Data is provided by the L0 cache alone (no multiplexer is involved). Since FDQ is an optional filtering mechanism, it is not subject to the same requirements of an SQ for correctness guarantee and hence there is quite a bit of latitude in its design. There is no need to try to accommodate all stores and therefore there is little scalability pressure. Thanks in part to the shorter life span of an entry (from address generation of a store to actual writing to the L0 cache), a small queue is sufficient for reducing the replay rate. The address used does not need to be translated nor is it necessary to use all address bits.

In contrast to prediction-based memory dependence enforcement [19, 20], our FDQ-based mechanism, though perhaps less powerful, is much more straightforward. First, the operation remains entirely address-based. With the addresses, dependence relationship is clear and unambiguous, thus we only need one structure. Memory dependence prediction, on the other hand, requires a multitude of tables, some of which very large and fully associative and many tables have a large number of ports. Second, while it is relatively easy to predict the presence of dependence, it is more challenging to pin-point the producer instruction [14].

[1] Although one bit is sufficient, a two-bit scheme is easier to explain. The most-significant two bits of the age get increased by 1 every time the write pointer of the ROB wraps around. When comparing ages, these two bits follow: $0 < 1$, $1 < 2$, $2 < 3$, and $3 < 0$.

Thus, to ensure accurate communication, additional prediction mechanisms [19] or address-based mechanisms [20] are used. Third, once a dependence is predicted, it still needs to be enforced. Memory dependence prediction designs [14, 19, 20] depend on the issue logic to enforce the predicted memory-based dependence in addition to the register dependence. We use the simpler rejection and retry method to keep the design less intrusive and more modular.

In summary, the intention of using FDQ is not to rival sophisticated prediction or disambiguation mechanisms in detecting and preventing dependence violations. It is meant as a cost-effective mechanism to mitigate performance loss in an SMDE system.

#### 3.2.2 Streamlining back-end execution

When a load reaches the commit stage, even if it has finished the front-end execution, it may still block the stage if it has not finished the back-end execution (reload). This will reduce the commit bandwidth and slow down the system. Ideally, the back-end execution pointer (BEEP) should be quite a bit ahead of the commit pointer, so that reload and verification can start early and finish before or when the instruction reaches the commit stage. Unfortunately, a store can only execute when the instruction is committed. This means, every time BEEP points to a store instruction, the back-end execution will be blocked until the commit pointer "catches up".

To streamline the back-end execution, we can employ the oft-used write buffer to allow stores to start early too, which indirectly allows loads to start early. If the processor already has a write buffer (to temporarily hold committed stores to give loads priority), then the buffer can be slightly modified to include a "not yet committed" bit that is set during back-end execution and cleared when the store is committed. Naturally, the write buffer has to provide forwarding to later reloads. Also, when handling memory synchronizers such as a write barrier or a release, the content of the write buffer has to be drained before subsequent reload can proceed. In our design, L0 cache fills do not check the write buffer (for simplicity).

Write buffering always has memory consistency implications. The sequential semantics require a reload to reflect the memory state after previous stores have been performed. A load must also reflect the effects of stores from other processors that according to the consistency model, are ordered before itself. In a sequentially consistent system, stores from all processors are globally ordered. Therefore, when an invalidation is received, it implies that the triggering store (from another processor) precedes all the stores in the write buffer and transitively precede any load after the oldest store in the write buffer. Thus, if BEEP has traveled beyond the oldest store, we need to reset it to the oldest store entry in the MOSQ and restart from there. In a processor with a weaker consistency model, stores tend to be only partially ordered, relaxing the need to roll back BEEP. For example, in a system relying on memory barriers, when we receive an external invalidation, we only need to roll back BEEP to restart from after the oldest memory barrier.

#### 3.2.3 Other possibilities of replay mitigation

While the FDQ successfully addresses the largest source of replays (Section 5.2), there are other techniques to further reduce replay frequency or to mitigate their performance impact. Some of these can be quite effective in reducing replays. It is also possible to use a write buffer without forwarding capabilities and stall a reload where the write buffer *potentially* contains an overlapping store. We discuss these possibilities here and show some quantitative analysis analysis later in Section 5.5.

**Replay suppression** When a replay is triggered, the common practice is to discard all instructions after the triggering instruction. This may throw out a lot of useful work unnecessarily. Within the instructions after the triggering load, only the dependents (direct or indirect) need to be re-executed. Therefore a selective replay is conceivable. We modeled selective replay and found that it is useful on top of the naive design. When replay frequency is reduced by optimization techniques, however, the benefit naturally reduces as well. In current microarchitectures, it is not easy to support selective replay. However, there is one degenerate case of it that could be supported: when the triggering load has no in-flight dependents, we only need to fix the destination register and the replay can be suppressed. Our simulations show that such load instructions are not rare: about 14% on average, and can be as high as 100%. Intuitively, it makes sense to detect these cases and avoid replays.

It is quite easy to track which load instruction has no in-flight dependents: At register allocation, we can maintain a bit vector, each bit corresponding to one *physical* register to indicate whether there is an in-flight instruction sourcing from it. Every instruction upon renaming will set the bits corresponding to the (renamed) source registers. When a load instruction is decoded, we reset the bit corresponding to the load's destination physical register. When a load triggers a replay and its destination register's bit is not set, we know there is no dependents in-flight and we can suppress the replay.

**Age-based filtering of L0 cache** When the processor recovers from a branch misprediction or a replay, the L0 cache is not proactively cleansed to remove pollution left by squashed instructions. This results in a period when the front-end execution is very likely to load a piece of incorrect data, triggering a replay. One option to mitigate the impact is to filter out some of the pollution by keeping track of the age of the data in the L0 cache: A store updates the age of the cache line and a load checks the age. If the age of the cache line is younger than that of the load, it is probable that the data is left by stores squashed due to misprediction or replay. In

that case, going directly to the L1 cache may be a safer bet than consuming the presumably polluted data. With a FDQ, we are already generating age IDs for memory operations. So tracking the age in the L0 requires only minimal extra complexity.

The age-tracking of L0 cache works as follows. The L0 cache's tag field is augmented with an age entry. A store instruction updates the cache line's age, whereas a load instruction compares its age with the age stored in the cache line at the same time tag comparison is being done. When the age of the load is older, we simply "fake" an L0 cache miss and access the L1. Note that here the age tracking is only a heuristic, therefore the tracking is not exact. The granularity is at the cache line level and we do not enforce the age to monotonically increase. Indeed, when out of order stores happen, age can decrease. Recall that we do not require any correctness guarantee from the front end, so we conveniently choose any tracking to reduce complexity. However, since we are using a finite number of bits to represent age, we need make sure that an old age ID is not misinterpreted as a new one in the future – otherwise, we may incur too many unnecessary misses in the L0. This can be easily achieved: When we extend the ROB ID by 2 bits to represent age, we reserve a coding of the 2-bit prefix as "old". For example, we can cyclically assign 1, 2, and 3 as the prefix for age and reserve prefix 0. An age ID with prefix 0 is older than another age with a non-zero prefix. Thus, when the commit pointer of ROB wraps around, we know that all instructions with a certain prefix (say 2) have committed and that prefix 2 will be recycled for a later time period, we then reset those entries whose age have a prefix of 2 to 0. Thus, these entries will not be incorrectly treated as pollution. When a cache line is brought in by a load, its age prefix is also set to 0.

### 3.3 Differences Between SMDE and Other Approaches

Recently, many techniques were proposed to address the LSQ scalability issue in one way or another. While most, if not all, techniques focus on providing a scalable alternative design that rivals an ideally scaled LSQ, our main focus is to reduce the conceptual and implementation complexity. Besides the difference in focus, the mechanisms we use in the speculation and the recovery from mis-speculation also differ from this prior work.

In a two-level disambiguation approach [1, 3, 8, 16, 23], the fundamental action is still that of an exact disambiguation: comparing addresses and figuring out age relationship to determine the right producer store to forward from. The speculation is on the *scope* of the disambiguation: only a subset of the stores are inspected. In contrast to these, in our front-end execution, we allow the loads to blindly access the cache structures. Our speculation is on the *order* of accesses: if left unregulated, the relative order of loads and stores to the same address is largely correct (same as program order).

Two recent designs eliminate fully-associative LQ and SQ [19,20]. They rely on dependence prediction to limit the communication of load to only a few stores. This is still a form of scope speculation – the scope of the stores communicating to a particular load is reduced to the extreme of one or a handful. Although both this approach and ours achieve an LSQ-free implementation, the two styles use very different design tradeoffs. While memory dependence is found to be predictable, *pin-pointing* the exact producer of a load instruction, on the other hand, is very ambitious and requires numerous predictor tables. Such a design also requires support from the issue logic to enforce the predicted dependence. Our design performs a different speculation and thus does not require the array of predictors or the extra dependence enforcement support from the issue logic.

The use of a small cache-like structure in architecture design is very common. However, the way our L0 cache is used is very different from prior proposals. In [20], great care is taken to maintain the access order of cache, so that no corruption will take place, even under branch mispredictions. In [14], the *transient value cache* is intended to reduce bandwidth consumption of stores that are likely to be killed. It contains only architecturally committed stores and is fully associative. In contrast to both these small caches, our L0 cache is fundamentally speculative. No attempt is made to guarantee the absence of data corruption. Indeed, corruptions of all sorts are tolerated in our L0 cache: out of order updates, wrong-path updates, lost updates due to eviction, virtual address aliasing, or even soft errors due to particle strikes.

For mis-speculation detection and recovery, we use an existing technique: in-order re-execution [6, 9]. However, the extra bandwidth consumption is a problem associated with this technique [17]. While prior approaches rely on reducing re-access frequency [6, 17, 19], our dual cache structure lends itself well to provide that extra bandwidth demand effortlessly, thereby avoiding the problem.

Finally, we note that two-pass execution is also used in entirely different contexts such as for whole-program speculation [21] and for fault-tolerance [2]. Our design shares the same philosophy of relying on another independent execution to detect problems. However, because of the different context, we do not re-execute the entire program nor attempt to repair all the state.

In all, our SMDE design employs novel speculation strategies and structures and uses existing techniques with new twists to solve problems in a simpler way. As we show in Section 5, although our target is design simplicity, the performance of an optimized version comes very close to that of an idealized LSQ system.

## 4 Experimental Setup

To conduct experimental analyses of our design, we use a heavily modified SimpleScalar [5] 3.0d. On the processor model side, we use separate ROB, physical register files, and LQ/SQ. We model load speculation (a load issues despite the presence of prior unresolved stores), store-load replay [7], and load rejection [22]. A rejected load suppresses issue request for 3 cycles. When a store-load replay happens, we model the re-fetch, re-dispatch, and re-execution. When simulating a conventional architecture, unlike what is done in the original simulator, we do not allocate an entry in the LQ for prefetch loads, *i.e.*, loads to the zero register (R31 or F31).

We also extended the simulator to model data values in the caches. Without modeling the actual values in the caches, we will not be able to detect incorrectly executed loads. We model the value flow very faithfully. For example, when a load consumes an incorrect value due to misspeculation and load from wrong address, its pollution in the L0 cache is faithfully modeled. When a branch consumes a wrong value and arrive at a wrong conclusion, we force the simulator to model the (spurious) recovery.

We also increased the fidelity in modeling the scheduler replay [7, 11]. A scheduler replay is related to speculative wakeup of dependent instructions of a load [12]. In the Alpha 21264 scheduler logic, when the load turns out to be a miss, all instructions issued during a so-called shadow window are pulled back to the issue queue. We model after Alpha 21264: there is a two-cycle shadow window in the integer scheduler, but in the floating-point issue logic, this is handled slightly differently. Only dependents of loads need to be pulled back [7].

Our consistency model is after that of the Alpha21264 processor as well. Since we are simulating sequential applications with no write barriers and there is no bus invalidation traffic, the only thing to note is that at the time a store is committed, if it is a cache miss, we keep the data in the SQ (in the conventional configurations) or the write buffer (if any, in the SMDE configurations) and allow the store to be committed from the ROB [7]. Note that in a naive SMDE implementation, there is no write buffer, so the store is not committed until the cache miss is serviced.

Our quantitative analyses use highly-optimized Alpha binaries of all 26 applications from the SPEC CPU2000 benchmark suite. We simulate half a billion instructions after fast-forwarding one billion instructions. The simulated baseline conventional processor configuration is summarized in Table 1. To focus on dynamic memory disambiguation, we size the ROB and register files aggressively, assuming optimization techniques such as [24] are used.

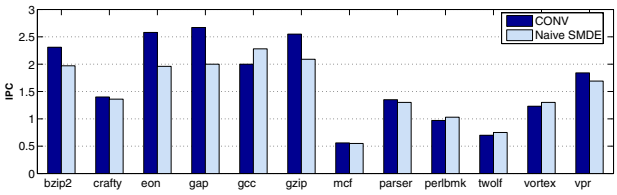
| Processor core | |
|---|---|
| Issue/Decode/Commit width | 8 / 8 / 8 |
| Issue queue size | 64 INT, 64 FP |
| Functional units | INT 8+2 mul/div, FP 8+2 mul/div |
| Branch predictor | Bimodal and Gshare combined |
| - Gshare | 1M entries, 20 bit history |
| - Bimodal/Meta table/BTB entries | 1M/1M/64K (4 way) |
| Branch misprediction penalty | 7+ cycles |
| ROB/Register(INT,FP) | 512/(400,400) |
| LSQ(LQ,SQ) | 112(64,48) - 1024(512,512), 2 search ports<br>1 cycle port occupancy, 2-cycle latency |
| **Memory hierarchy** | |
| L0 speculative cache | 16KB, 2-way, 32B line, 1 cycle, 2r/2w |
| L1 instruction cache | 32KB, 2-way, 64B line, 2 cycles |
| L1 data cache | 64KB, 2-way, 64B line, 2 cycles, 2r/2w |
| L2 unified cache | 1MB, 8-way, 128B line 10 cycles |
| Memory access latency | 250 cycles |

**Table 1.** System configuration.

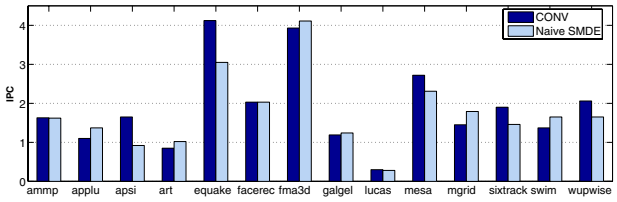
## 5 Experimental Analysis

### 5.1 Naive Implementation

We start our experimental analysis with the comparison of IPCs (instruction per cycle) achieved in a baseline conventional system (with LSQ) and in a naive implementation of SMDE. We use a baseline conventional design with optimistically-sized queues: 48 entries in SQ and 64 entries in the LQ. Note that, in a high-frequency design, supporting a large number of entries in LQ and SQ is indeed challenging. Even the increase of the SQ size from 24 to 32 in Intel's Pentium 4 processor requires speculation to ease the timing pressure [4].




(a) Integer applications.




(b) Floating-point applications.

**Figure 4.** Comparison of IPCs in the baseline conventional system (CONV) and a naive SMDE system.

From Figure 4, we see that *on average*, the naive SMDE system performs slightly worse than the baseline. While naive SMDE suffers from performance degradation due to replays, it does not have any limit on the number of in-flight memory instructions and this can offset the performance degradation due to replays. In general, floating-point applications tend to benefit from a large number of in-flight

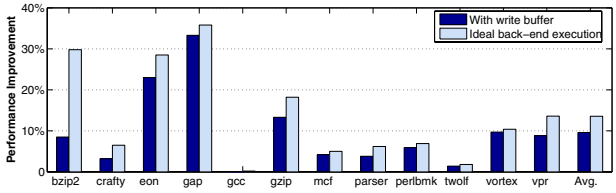
instructions and thus tend to perform better in an SMDE system than in a conventional one. Bear in mind that Figure 4 shows the naive design, which, though functional, is hardly an efficient implementation of an SMDE paradigm. Yet, we can see that even this simplistic design achieves an acceptable performance level. Given its simplicity, this result is very encouraging.

In the above and the following discussion, we use a single-flush policy because of its simplicity (Section 3.1.4). In Section 5.5, we show some details about the different flush policies.

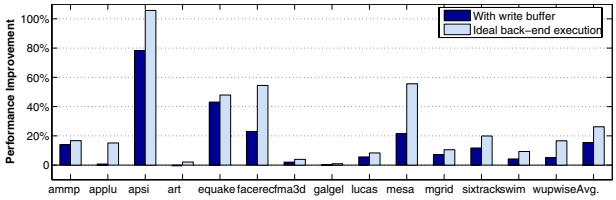
### 5.2 Effectiveness of Optimization Techniques

Although the naive design eliminates any stall time due to LSQ fill-up, it introduces other performance degradation factors. We now look at the effect of the mitigation techniques.

**Back-end execution bottleneck** Recall that in the back-end execution of the naive implementation, a store blocks the advance of the execution pointer until commit. This makes the latency of subsequent reloads more likely to be exposed, reducing commit bandwidth. In Figure 5, we show the performance improvement of using the write buffer described in Section 3.2.2. We also included a configuration with an idealized back-end execution where the latency and L1 cache port consumption of re-executing loads and stores are ignored. (Of course, replays still happen.) In this configuration, only the baseline processor's commit bandwidth and whether an instruction finishes execution limit the commit. All performance results are normalized to the naive SMDE configuration (without a write buffer).




(a) Integer applications.

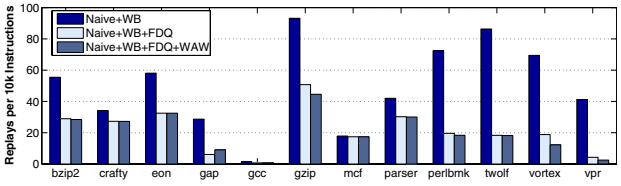



(b) Floating-point applications.

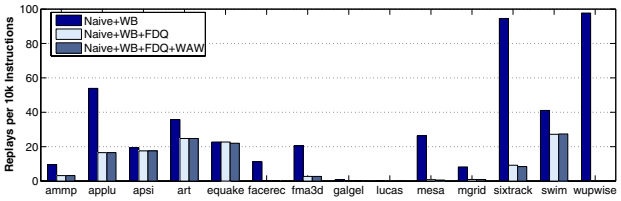
**Figure 5.** Performance impact of using an 8-entry write buffer in the back-end execution and of having an ideal back-end execution.

The first thing to notice is that there is a very significant difference between the naive and the ideal implementation of the back-end execution. For example, in the case of *apsi*, when removing the restriction placed by the back-end execution, the performance more than doubles. Second, an 8-entry write buffer is able to smooth out the re-execution of loads and stores and provide a performance improvement of 10-15%, very close to that of an ideal implementation. Although for a few applications, such as *bzip2* and *mesa*, there is still room for significant improvement.

**Replay frequency reduction** Figure 6 shows the frequency of replays before and after enabling a 16-entry FDQ (Section 3.2.1). We also use FDQ to detect and reject out-of-order stores to prevent write-after-write (WAW) violations. In this study, all configurations have an 8-entry write buffer in the back-end.




(a) Integer applications.




(b) Floating-point applications.

**Figure 6.** Replay frequency under different configurations.

The replay frequency varies significantly from application to application and can be quite high in some applications. After applying the FDQ, the replay frequency drastically reduces for many applications. In *wupwise* for example, the frequency changes from about 97.7 replays per 10,000 instructions to no replays at all in the 500 million instructions simulated. We see that using the FDQ to detect WAW violations have a much smaller impact and can, in some cases, lead to a slight increase in the replay rate. However, the overall effect is positive. In the following, when we use FDQ, by default we detect WAW violations too.

In Figure 7, we present another view of the effect of using various optimization techniques: the breakdown of replays into different categories. A replay is caused by loading a piece of incorrect data from the L0 cache. This can be caused by memory access order violations (RAW, WAW, and WAR) in the front-end execution. A replay can also be caused by L0 cache pollution when the execution is recovered from a branch misprediction or a replay, or due to

a cache line eviction. A replay can have multiple causes. For the breakdown, we count a replay towards the last cause in time. Therefore, when a technique eliminates a cause for one load, it may still trigger a replay due to a different cause and increase the number of replays in another category. However, this effect is small. In this experiment, we start from the naive design and gradually incorporate all the optimization techniques. Since there is overlap between the effects of different techniques, the result of applying multiple techniques is not additive, and the techniques introduced later on tend to demonstrate diminishing returns. For clarity, we only show the average of the number of replays per 10,000 committed instructions for integer applications and for floating-point applications.

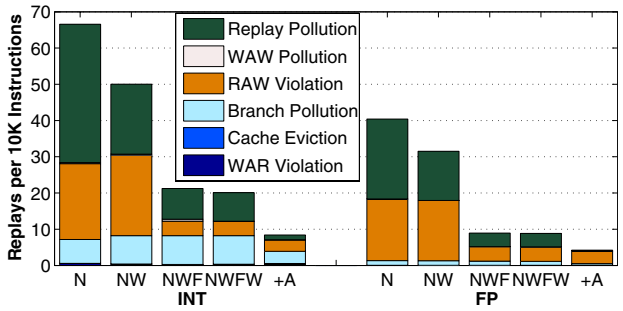



**Figure 7.** The breakdown of the number of replays into different categories. From left to right, the three bars in each group represent different systems: Naive (N), with write-buffer but without the FDQ (Naive + Write-buffer: N+W); with the FDQ to detect RAW violations (N+W+F) and with FDQ to also detect WAW violations (N+W+F+W), and finally, adding age-based filtering (+A) (on top of N+W+F+W). The stacking order of segments is the same as that shown in the legend. Note that the two bottom segments are too small to discern.

The first interesting thing to note is that when write buffer is introduced to the system, replay frequency is significantly reduced. For example, in integer code, the average replays per 10,000 committed instructions reduce from about 67 to 50 in integer programs and from 40 to 31 in floating-point programs. This may appear puzzling initially since the write buffer is not designed to reduce replay frequency. However, as we can see, the reduction is primarily due to that of replay pollution. Indeed, as we speed up the back-end of the processor, we can detect a replay more quickly and prevent too many stores from leaving future data in the L0 cache. In application *bzip2* for example, without write buffer, every replay squashes 46 stores. With the write buffer, this number reduces drastically to 13.

Next, we see that in a system without FDQ (N+W), the two main *direct* sources of replays are branch (misprediction) pollution and RAW violation. Of the two, RAW is the larger source by a small margin in integer applications and by a large margin in floating-point applications. As replays leave the L0 cache in a state with many "future" data, they are likely to trigger replays again. Such secondary replays account for an average of 40% of all replays. In both groups of applications, FDQ is capable of preventing a large majority (75% to 80%) of RAW violation-triggered replays. As a result, the number of secondary replays also reduces significantly (by an average of 55% and 73% in integer or floating-point applications respectively). We can also see that when FDQ is also used to prevent WAW violations, the number of WAW pollution-triggered replays indeed goes down, though the overall impact is quite small.

Finally, we can see that although in floating-point applications, FDQ can cut down the replays by about 75% to a small 7 per 10,000 instructions, in integer applications, there are still about 20 replays left. These replays are mainly due to branch pollution, directly or indirectly. With the age-based filtering, we are able to filter out a large portion of pollution due to replay and branch misprediction recovery in both groups of applications: 95% (FP) and 84% (INT). The reduction in branch pollution is smaller, but still significant: 60% in both groups of applications.

### 5.3 Putting it Together

We now compare the performance of several complete systems which differ only in the memory dependence enforcement logic. We take the naive design, add an 8-entry write buffer and a 16-entry FDQ. We call this design *Improved*. We compare *Naive* and *Improved* to the baseline conventional system, which uses the conventional disambiguation logic with a 64-entry LQ and a 48-entry SQ, and to an idealized system where we set the LQ and SQ size equal to that of the ROB. In Figure 8, these results are shown as minimum, average, and maximum for the integer and floating-point application groups as before. (The per-application detailed results are listed in Table 2.) We see that there are applications that perform dramatically worse in the naive design than in the baseline but there are others that achieve significant improvements as well. On average, the naive design performs only slightly worse than the baseline (5.3% and 2.7% slower for INT and FP applications, respectively). We point out that with a combined LSQ capacity of 112 entries, even the baseline is optimistically configured. (A smaller but more realistic 32-entry SQ with 48-entry LQ would slow down the system by an average of 11% when executing floating-point applications). Undoubtedly, the naive design is much more complexity-effective.

The *Improved* design is significantly better than *Naive* and comes very close to the ideal configuration. Individual applications' slowdown relative to the ideal configuration can be as high as 15.6%. However, except for 3 applications, all others are within 10% of *Ideal*. In fact, *Improved* actually outperforms the ideal configuration in seven appli-

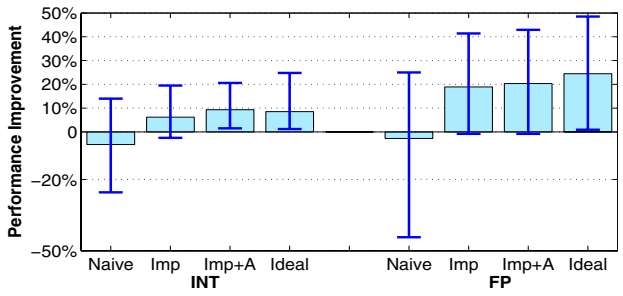



**Figure 8.** Performance improvement of *Naive* and *Improved* (Imp) design over the baseline conventional system. Adding age-based filtering on top of *Improved* (Imp+A) and an ideal conventional system are also shown.

cations, most notably *gcc* and *equake*. This is not unreasonable: Our design turns the transistors used in building LSQ or predictor tables into a small L0 cache than can cache more data and provide the data to the execution core faster. If we apply additional techniques, such as age-based filtering to reduce the replay frequency, a few more applications would run faster in an SMDE design than in *Ideal*. Indeed, adding age-based filtering on *Improved*, we have a system that outperform an ideal LSQ-based system on average (Figure 8).

Overall, comparing *Improved* with *Ideal*, the average slowdown is only 2.0% for integer applications and 4.3% for the floating-point applications. Although practicality prevents a pair-wise comparison study with the numerous LSQ-optimization designs, we note that these results (both average and worst-case) come very close to those reported previously, for example, in [19].

Finally, to understand the effect of the age-based filtering (Section 3.2.3), we add it to *Improved* and show the effect in Figure 8. We can see the notable effect of age-based L0 cache filtering, especially on integer application. Indeed, with this filtering, the average performance improvement over baseline is actually higher than that with ideal LSQ.

### 5.4 Scalability

Perhaps more interesting than the actual performance is the scalability trend. In this paper, we perform a limited experiment scaling the system to a 1024-entry ROB. We scale the size of the register files and the issue queues proportionally but keep the disambiguation logic exactly the same as in Figure 8 for the baseline conventional system and the SMDE designs (Naive and Improved). In other words, the size of the L0 cache, the write buffer, and the FDQ remains the same. The LQ and SQ in the ideal conventional system, however, are scaled together with the ROB. We again normalize to the baseline conventional configuration. We also "scale up" the branch predictor by randomly correcting 75% of mispredictions in the simulator. We show the result in Figure 9.

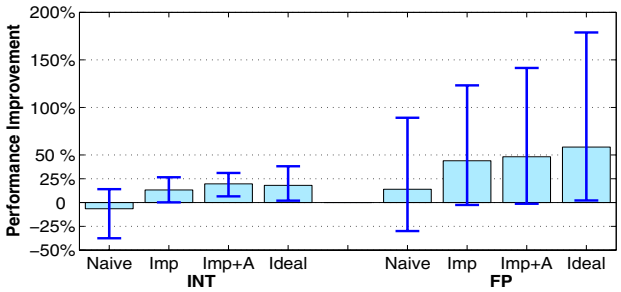



**Figure 9.** Scalability test for SMDE configurations (*Naive* and *Improved*) and the ideal conventional configuration.

First, we see that the naive design gains ground against the baseline for floating-point applications. This is expected. Without any limit on the number of in-flight instructions, the naive approach can easily use the enlarged ROB to exploit long-range ILP. This benefit adds to that of better branch prediction in the scaled-up system. For the baseline conventional system however, the large ROB has no effect due to the limit of the LSQ. The benefit almost entirely comes from better branch prediction in the scaled-up system. Thus, naive SMDE's performance improves relatively to the baseline.

Second, the improved SMDE design continues to track the ideal conventional configuration well. This is a key benefit in addition to the simplicity of the SMDE design: Once the design is in place, hardly any resource scaling is needed.

Finally, age-based filtering shows more benefit in the scaled up configuration. However, the difference is not dramatic. If the underlying microarchitecture changes and the cost of replay is higher, then it may become a more useful mitigation technique.

### 5.5 Other Findings

Finally, during the design and evaluation of SMDE, we performed many quantitative experiments. We summarize some of the findings here.

**Replay suppression** When a replay is triggered, there are times when no in-flight instruction is a dependent of the triggering load. Depending on the exact configuration, the percentage of such cases vary but remains non-negligible. For example, in the *Improved* configuration, on average, there are about 20% of replay-triggering loads in floating-point applications that do have any in-flight dependents. In certain applications this rate can be as high as 100%. Therefore, in an SMDE design, we do not need to trigger an actual replay of subsequent instructions. However, the end performance implication is rather limited in this configuration: about 1%.

| | bzi | cra | eon | gap | gcc | gzip | mcf | par | pbm | two | vor | vpr | amm | app | aps | art | eqk | fac | fma | gal | luc | mes | mgr | six | swi | wup |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | -14.6 | -2.8 | -24.0 | -25.4 | 14.0 | -18.1 | -1.7 | -3.4 | 6.1 | 8.0 | 5.7 | -7.8 | -0.7 | 25.0 | -44.3 | 19.9 | -25.8 | -0.1 | 4.4 | 4.7 | -7.1 | -15.0 | 23.4 | -23.3 | 20.2 | -19.8 |
| 2 | -2.5 | 2.0 | 0.9 | 9.4 | 10.7 | 0.5 | 2.9 | 1.8 | 15.4 | 4.3 | 19.5 | 9.6 | 15.1 | 39.2 | 1.0 | 17.6 | 6.6 | 41.4 | 6.8 | 5.6 | -0.8 | 20.2 | 41.1 | 4.5 | 30.3 | 35.9 |
| 3 | 1.5 | 7.0 | 10.4 | 11.1 | 10.4 | 10.4 | 3.8 | 5.1 | 16.6 | 4.8 | 20.6 | 9.9 | 15.6 | 42.9 | 1.9 | 18.2 | 19.1 | 41.4 | 8.0 | 5.6 | -0.8 | 20.2 | 41.3 | 5.6 | 38.2 | 27.3 |
| 4 | 11.3 | 10.4 | 6.2 | 11.8 | 1.9 | 1.2 | 3.6 | 1.7 | 16.2 | 4.1 | 24.8 | 9.0 | 19.3 | 48.5 | 19.6 | 31.7 | 4.8 | 41.1 | 0.9 | 7.0 | 1.5 | 32.2 | 46.2 | 9.7 | 42.3 | 37.8 |

**Table 2.** Performance improvement (in %) of Naive (1), Improved (2), Improved with age-based filtering of L0 cache (3), and Ideal (4) design over the baseline conventional system.

**Flush policy** When handling a replay, we can perform some L0 cache cleaning. We emphasize again that this is purely an optimization to reduce the chance of more replay and does not affect correctness in any way. Among the possible policies, we compared no flushing (F0), flush the line that triggered the replay (F1), selective flushing (FS) by walking through the MOSQ to flush all the lines belonging to the to-be-squashed stores, and finally, flush the entire L0 cache (FA). It is not hard to see that F0 and F1 are fairly straightforward to implement, while FA and especially FS can be complex in circuit and expensive in energy at runtime, though they definitely reduce the replay rate.

In Table 3, we show the percentage of loads triggering replay in the *Improved* design if we use different flushing policies. We can see that as we flush more data from the L0, naturally, the replay frequency reduces. However, L0 cache miss rate increases as we flush more data. An L0 cache miss not only delays the execution of dependents of the load, it also disrupts the execution of instructions scheduled in the vicinity of the load due to scheduling replay. As we can see, all other policies give lower average performance when compared to F1. (The effect of FS is very close to that of F1.) Fortunately, unlike FS, F1 is very easy to implement.

| | INT | | | | FP | | | |
|---|---|---|---|---|---|---|---|---|
| | Max | Avg | Min | Perf. | Max | Avg | Min | Perf. |
| F0 | 218 | 54 | 1 | -7.43 | 80 | 18 | 0 | -5.51 |
| F1 | 45 | 20 | 1 | 0 | 27 | 9 | 0 | 0 |
| FS | 27 | 12 | 1 | 0.99 | 24 | 5 | 0 | -1.49 |
| FA | 28 | 9 | 1 | -2.35 | 23 | 4 | 0 | -3.25 |

**Table 3.** Number of loads triggering replay per 10,000 instructions in *Improved* under different L0 cache flushing policies (shown in maximum, average, and minimum of the entire group of applications), and their average performance impact (Perf.) in percentage compared to the single-line flush policy (F1).

**Understanding the write buffer** The write buffer we use serve three purposes. First, it buffers write misses. When a store instruction is being committed but misses in the cache, depending on the design of the memory subsystem, the processor may not be able to remove the instruction from the ROB and continue to commit other instructions. In Alpha [7], for example, the instruction is removed from the ROB, whereas the data is kept in the SQ in order to properly forward to load instructions. In SMDE, without a SQ, we can either stall commit when there is a write miss, or if there is the write buffer, we can keep the data in the write buffer and retire the instruction. Note that at the execution time for the store instruction, processors typically prefetch the cache line. Thus, write miss at commit time is quite rare. A second functionality of the write buffer is to improve the utilization of cache port. When a store being committed compete with a load for the cache ports, the write buffer allows both to proceed by time-shifting the store's usage of cache port to a later cycle. Finally, the third functionality is to allow the BEEP pointer to travel ahead with respect to the retirement pointer of the ROB so as to hide the latency of back-end execution of load instructions.

To understand the effect of all three functions, we incrementally add them to the *Naive* system and measure their effect in terms on performance improvement on top of *Naive* (without a write buffer). Table 4 shows this experiment. As we can see, hiding load latency in the back-end execution is the most important contribution of the write buffer, but the other functionalities also contribute non-trivial amount of improvement.

| | INT | FP |
|---|---|---|
| WB holding write misses only | 2.15% | 5.75% |
| Also time-shift cache ports | 4.47% | 7.69% |
| All three functions | 15.95% | 16.08% |

**Table 4.** Effect of different functionalities of the write buffer.

**Membership test** As we saw earlier, the effect of even a small write buffer is quite substantial. However, the forwarding capability from a small write buffer is not necessarily critical. In fact, we found that only 3-4% of loads forward from this write buffer on average. Therefore it is conceivable to have a non-forwarding write buffer and provide a simple and quick membership test to detect any address overlapping and stall the load until the conflicting store is drained out of the buffer. We performed a limited study using an ideal membership test to study the performance impact of having to delay conflicting loads. We found that the average performance degradation is about 0.5% for both groups of applications and the maximum slowdown is only about 2% in any single application.

## 6 Conclusions

In this paper, we have presented a slackened memory dependence enforcement (SMDE) approach aimed at simplifying one of the most complex and non-scalable functional blocks in modern high-performance out-of-order processors. In an SMDE design, memory instructions execute twice, first in a front-end execution where they execute only according to register dependences and access an L0 cache to perform an opportunistic communication. They then access memory a second time, in program order, accessing the non-speculative L1 cache to detect mis-speculations and recover from it.

A primary advantage of SMDE is its simplicity. It is also very effective. We have shown that even a rudimentary implementation rivals a conventional disambiguation logic with optimistically sized LQ and SQ. When two optional optimization techniques are employed, the improved design offers performance close to that of a conventional system with ideal LSQ. Another advantage of the design is that when scaling up the in-flight instruction capacity, almost no change is needed and yet the performance improves significantly.

The SMDE approach is distinct in several ways from conventional design and recent proposals. First, it is significantly more decoupled between the forwarding and monitoring/verification component. This allows for modular design, verification, and optimization. Second, the forwarding component, working at core execution speed has minimal external monitoring or interference. There is no need to communicate with front-end predictors either. There is even no need for address translation. Third, the verification component is straightforward and handles all cases: misspeculation or coherence/consistency constraints. By providing a separate cache, we *avoid* the bandwidth consumption problem of re-execution.

## Acknowledgements

This work is supported in part by National Science Foundation under grant CNS-0509270. We wish to thank Ruke Huang for contributions at the early stage of the work. We thank the anonymous reviewers and Jose Renau for their helpful comments.

## 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, San Diego, California, December 2003.

[2] T. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *International Symposium on Microarchitecture*, pages 196–207, Haifa, Israel, November 1999.

[3] L. Baugh and C. Zilles. Decomposing the Load-Store Queue by Function for Power Reduction and Scalability. In *Watson Conference on Interaction between Architecture, Circuits, and Compilers*, Yorktown Heights, New York, October 2004.

[4] D. Boggs, A. Baktha, J. Hawkins, D. Marr, J. Miller, P. Roussel, R. Singhal, B. Toll, and K. Venkatraman. The Microarchitecture of the Intel™Pentium™4 Processor on 90nm Technology. *Intel Technology Journal*, 8(1):1–17, February 2004.

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

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

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

[8] 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*, Madison, Wisconsin, June 2005.

[9] K. Gharachorloo, A. Gupta, and J. Hennessy. Two Techniques to Enhance the Performance of Memory Consistency Models. In *International Conference on Parallel Processing*, pages I355–I364, St. Charles, Illinois, August 1991.

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

[11] R. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 9(2):24–36, March 1999.

[12] I. Kim and M. Lipasti. Understanding Scheduling Replay Schemes. In *International Symposium on High-Performance Computer Architecture*, pages 198–209, Madrid, Spain, February 2004.

[13] 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, Denver Colorado, June 1997.

[14] A. Moshovos and G. Sohi. Streamlining Inter-operation Memory Communication via Data Dependence Prediction. In *International Symposium on Microarchitecture*, pages 235–245, Research Triangle Park, North Carolina, December 1997.

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

[16] A. Roth. A High-Bandwidth Load-Store Unit for Single- and Multi- Threaded Processors. Technical Report (CIS), Development of Computer and Information Science, University of Pennsylvania, September 2004.

[17] A. Roth. Store Vulnerability Window (SVW): Re-Execution Filtering for Enhanced Load Optimization. In *International*

*Symposium on Computer Architecture*, Madison, Wisconsin, 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, San Diego, California, December 2003.

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

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

[21] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream Processors: Improving both Performance and Fault Tolerance. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 257–268, Cambridge, Massachusetts, November 2000.

[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, January 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*, Madison, Wisconsin, June 2005.

[24] J. Tseng and K. Asanovic. Banked Multiported Register Files for High-Frequency Superscalar Microprocessors. In *International Symposium on Computer Architecture*, pages 62–71, San Diego, California, June 2003.