Reducing Design Complexity of the Load/Store Queue

Il Park, Chong Liang Ooi, and T. N. Vijaykumar
School of Electrical and Computer Engineering, Purdue University
{parki, cliang, vijay}@ecn.purdue.edu

Abstract

With faster CPU clocks and wider pipelines, all relevant microarchitecture components should scale accordingly. There have been many proposals for scaling the issue queue, register file, and cache hierarchy. However, nothing has been done for scaling the load/store queue, despite the increasing pressure on the load/store queue in terms of capacity and search bandwidth. The load/store queue is a CAM structure which holds in-flight memory instructions and supports simultaneous searches to honor memory dependencies and memory consistency models. Therefore, it is difficult to scale the load/store queue.

In this study, we introduce novel techniques to scale the load/store queue. We propose two techniques, store-load pair predictor and load buffer, to reduce the search bandwidth requirement; and one technique, segmentation, to scale the size. We show that a load/store queue using our predictor and load buffer needs only one port to outperform a conventional two-ported load/store queue. Compared to the same base case, segmentation alone achieves speedups of 5% for integer benchmarks and 19% for floating point benchmarks. A one-ported load/store queue using all of our techniques improves performance on average by 6% and 23%, and up to 15% and 59%, for integer and floating-point benchmarks, respectively, over a two-ported conventional load/store queue.

1 Introduction

In a modern out-of-order microprocessor, the load/store queue is designed to absorb bursts in cache accesses and to maintain the order of memory operations by keeping all in-flight memory instructions in program order. As CPU clocks become faster, wire delays to the cache hierarchy worsen and the processor-memory performance gap widens. As a result, there are more in-flight memory instructions in the pipeline, increasing the pressure on the load/store queue. Therefore, we need *higher capacity* in the load/store queue. In addition, modern microprocessors employ wider issue for higher performance, requiring *higher search bandwidth* in the load/store queue to allow more memory instructions to overlap.

With faster processor clocks and wider pipelines, all relevant microarchitecture components should scale accordingly. Otherwise, the processor performance will show little improvement despite the faster clock and wider pipeline. To address this problem, a lot of research has been done for scaling the issue queue, register file, and cache hierarchy; yet nothing has been done for scaling the load/store queue. In fact, the techniques proposed for scaling the issue queue, register file, and cache hierarchy increase not only the number of in-flight instructions but also the overlap among instructions [8,7,5]. These techniques greatly increase the demand for both *higher capacity* and *higher search bandwidth* in the load/store queue.

The load/store queue is a CAM structure that supports simultaneous associative searches to honor memory dependencies and memory consistency models. Consequently, it is extremely difficult to scale the capacity and bandwidth of the load/store queue. Brute-force approaches to scaling the load/store queue are not likely to work.

In a modern processor, the load/store queue is implemented as two separate queues and has three functions: (1) The load/store queue buffers and maintains all in-flight memory instructions in program order. (2) The load/store queue supports associative searches to honor memory dependence. A load searches the *store queue* to obtain the most recent store value, and a store searches the *load queue* to find any premature loads (*store-load order violation*). (3) In some processors, the load/store queue supports associative searches to enforce memory consistency (in shared-memory multiprocessors). Specifically, the ordering among loads that access the same address is an important special case. If this ordering is relaxed, subtle correctness problems arise: if two loads to the same address are issued out of order and the value is changed by another processor in between the two loads [1], the later load will obtain an earlier value whereas the earlier load will obtain a later value. To avoid this problem, some processors (e.g., Alpha [3] and POWER4 [10]) guarantee load-load ordering for loads to the same address. For this guarantee, a load searches the load queue to find any out-of-order-issued loads (*load-load order violation*).

In this paper, we propose three techniques to scale the load/store queue; two of these techniques reduce the search bandwidth demand on the load/store queue, and the other technique increases the capacity of the load/store queue.

We use two key observations to reduce the search bandwidth demand. First, previous studies show that store-load order violations are highly predictable and infrequent [6, 2]. They use this observation to reduce the number of pipeline

IEEE COMPUTER SOCIETY

squashes due to store-load order violations, while allowing out-of-order memory operations as much as possible. While they show that store-load order violations are rare, we observe that not only are the violations rare but a majority of stores and loads do not even access the same address. We use this observation to reduce the search bandwidth demand on the *store queue*. We use the store-set predictor from [2] to predict store-load dependencies. If a load is predicted to be independent of any preceding stores, the load will not search the store queue. If there is a dependent earlier store, then the prediction is wrong. Consequently, when the store completes, it will squash the load and subsequent instructions.

Second, we observe that in order to detect a load-load order violation, any given load needs to search only those loads that were issued out of order with respect to the load, and that out-of-order-issued loads are far fewer in number than all in-flight loads. Across the SPEC2K benchmarks, the average number of out-of-order-issued loads is less than 3, while the number of in-flight loads is 41. We use this observation to reduce the search bandwidth demand on the *load queue*. We introduce the *load buffer*, which is a small buffer to hold loads that are issued out of order with respect to earlier yet-to-be-issued loads. When a load issues, it searches the much-smaller load buffer instead of searching the entire load queue. Thus the load buffer moves the detection of load-load order violations away from the load queue.

The contributions of the paper are as follows:

- **Reducing the search bandwidth demand on the store queue:** While [2] uses the store-set predictor to avoid store-load order violations, our novelty is in applying the predictor to the new problem of reducing the store queue bandwidth and appropriately changing the load/store queue implementation. With this technique, we reduce the search bandwidth demand on the store queue by 72% on average for SPEC2K benchmarks.
- **Reducing the search bandwidth demand on the load queue:** Our novelty is in observing that out-of-order-issued loads are only a few in number and in proposing the load buffer. With this technique, we reduce the search bandwidth demand on the load queue by 76% on average for SPEC2K benchmarks.
- **Increasing the load/store queue capacity:** We segment the load/store queue into multiple smaller queues and connect them in a chain. The idea of segmentation is not new—e.g., [4] [8] segment the issue queue. While [8] treats the segments as a hierarchy by moving instructions from slow to fast segments, and [4] treats the segments as discrete units of power dissipation, our novelty is that we treat the segments as a pipeline. Such pipelining essentially makes our load/store queue a variable-latency structure, a design point not explored before.

Using the SPEC2K benchmarks, we show that a load/store queue using our techniques needs only one port to outperform a two-ported, conventional load/store queue. Compared to the same base case, segmentation in isolation achieves speedups of 5% for integer benchmarks and 19% for floating point benchmarks. A one-ported load/store queue using all our techniques improves performance on average by 6% and 23%, and up to 15% and 59% for integer and floating-point benchmarks, respectively, over a two-ported conventional load/store queue.

This paper is organized as follows. In Section 2 we introduce two techniques to reduce the search bandwidth demand. In Section 3 we discuss the segmentation of the load/store queue and its impact on bandwidth and latency. We present and analyze the results of these three techniques in Section 4. Finally, we conclude in Section 5.

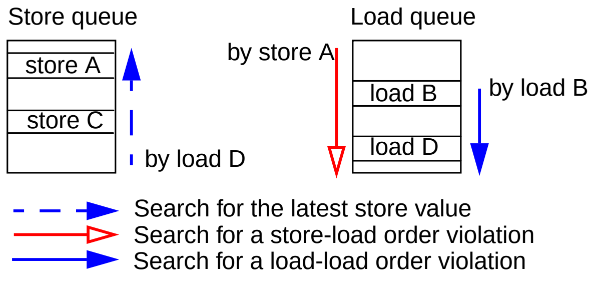

Store queue
store A
store C
by load D
by store A
Load queue
load B
load D
by load B
Search for the latest store value
Search for a store-load order violation
Search for a load-load order violation

Figure 1: Searches in the ld/st queue.

2 Reducing Search Bandwidth Demand

A modern processor performs three important searches on the load/store queue which is implemented as two separate queues. Figure 1 shows these searches in the load/store queue. First, when a load executes, it searches the *store queue* to compare its load address to the addresses of all stores. If there is a match with an earlier store, then the load obtains its value from the store queue and ignores the value from the cache. Second, when a store has a valid address, it searches the *load queue* to compare its store address to the addresses of all loads. If the address matches with a younger, speculatively-serviced load, this *premature load* and all subsequent instructions are squashed and fetched again (store-load order violation). Third, when a load executes, it searches the load queue to compare its address to the addresses of all loads, in some processors ([3, 10]). If the address matches with a younger out-of-order-issued load, this load and all subsequent instructions are squashed and fetched again (load-load order violation).

2.1 Reducing Store Queue Search: Store-Load Pair Predictor

As mentioned above, a load needs to search the store queue to obtain the latest store value. While [6] shows that store-load order violations are rare, we observe that not only are the violations rare, but a majority of stores and loads do not even access the same address. Consequently, most of the

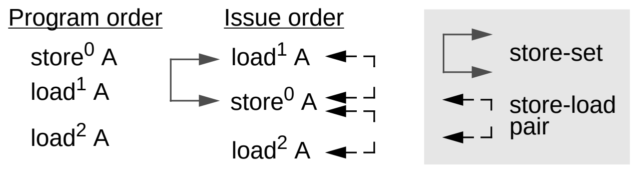


Figure 2: Store-set vs. store-load pair prediction.

time the search fails, and the load ends up using the value from the cache hierarchy. Our study confirms that only about 14% of searches find a matching store. If we can tell ahead of time that a load will not find any matching store, we will avoid performing useless store queue searches. Thus, we can reduce the search bandwidth in the store queue by 86% in the ideal case.

The store-set study shows that the store-load order violation is highly predictable by using a reasonably simple predictor [2]. We extend the store-set predictor to predict the matches between loads and stores. We call our scheme the *store-load pair predictor.* A load will search the store queue only when the store-load pair predictor predicts that there is a potentially-dependent store in the queue and tells the load to obtain its value from the store queue. Otherwise, the load simply obtains its value from the cache hierarchy without searching the store queue.

While the store-set predictor detects only those store-load pairs that cause dependence violations, our store-load pair predictor detects *all* matching pairs of loads and stores regardless of whether they cause violations. Figure 2 illustrates this point. The store-set predictor will need to detect only the store^0-load^1 pair, while our store-load pair predictor will detect both the store^0-load^1 pair and the store^0-load^2 pair.

In the case of a misprediction of a store-load pair under our predictor, the load will not search the store queue and instead will use a stale value from the cache, without knowing that there is a matching store in the store queue. Therefore it is up to the store to detect this problem. The store must handle two cases: the load is issued either before (e.g., load^1) or after (e.g., load^2) the store. The store detects the first case when it executes and searches the load queue for premature loads. However, the store cannot detect the second case because the load has not been issued when the store searches the load queue.

To handle this condition *without increasing the search bandwidth,* we change the timing for the detection of the store-load order violation; now, a store searches the load queue for a matching load when the store commits (writes to the cache) and removes the entry from the store queue, not when the store executes. Thus, all loads with stale or premature values will be detected when the store commits. This change further reduces the number of stores to search the load queue because the number of stores committed is significantly less than the number of stores executed. While this change does not incur more searches, and hence higher bandwidth, this change does increase the penalty of store-load order violations and store-load pair mispredictions. Previously, a store-load order violation was detected by the store when the store executed and computed the effective address. Now, the store cannot detect the hazards until it commits. Therefore, our store-load pair prediction must be conservative enough to avoid the performance degradation due to this extra misprediction penalty; however, it should not be too conservative to kill all the opportunity for reducing the search bandwidth demand on the store queue.

2.1.1 Implementing the Store-Load Pair Predictor

The store-load pair predictor uses structures similar to the store-set predictor, including the Store Set ID Table (SSIT) and the Last Fetched Store Table (LFST). Each store set is a collection of a load and one or more potentially-matching stores. The SSIT is indexed by the program counter and maintains the store-sets using a tag for each load and the stores in its store-set. The LFST is indexed by the store-set identifier obtained from the SSIT entry and maintains the information about the most-recently fetched store for each store-set [2]. While the store-set's LFST has a valid bit in each entry, the store-load pair predictor uses an extra counter in each LFST entry instead of the valid bit.

In the store-set predictor, the valid bit of an LFST entry is set when a store relevant to the entry is fetched. The valid bit is reset when the store issues, because there is no possibility for the store to violate store-load order after that point in time. When a load is fetched, it accesses the SSID and LFST to determine whether the load is potentially dependent on any store in flight. A load that is predicted to be dependent monitors the valid bit of the LFST entry relevant to the load when it is ready to issue. If the valid bit is set, the load has to wait until the valid bit is cleared.

A single valid bit is not sufficient for the store-load pair predictor. As mentioned earlier, a store searches the load queue to detect store-load order violations when the store commits. From the store-load pair predictor's point of view, the LFST entry is valid from the time the store is fetched to the time the store commits. As a result, the time interval for the LFST entry to be valid from the store-load pair predictor's point of view—from fetch to commit—is much longer than that from the store-set predictor's point of view—from fetch to issue.

This longer time interval causes the store-load pair predictor to have a higher probability of encountering multiple in-flight stores with the same program counter. If there are multiple stores with the same program counter and we were to use a single valid bit, then the first store would invalidate the LFST entry at commit without waiting for the remaining in-flight stores. Subsequent loads would assume incorrectly that there are no more dependent stores in the pipeline.

To solve this problem, we introduce a multi-bit counter for each LFST entry. By using the multi-bit counter, we can

		Fetch	Issue	Commit	Commit stores
store-set only	store	Valid = true; update LFST	Valid = false;	update SSID	
	load	access SSID&LFST	read Valid	update SSID	
store-set + **store-load pair**	store	Valid = true; **Counter++;** **update LFST**	Valid = false;	**update SSID**	**Counter--;**
	load	**access SSID&LFST**	read Valid **read Counter**	**update SSID**	

Figure 3: Combining the store-load pair predictor with the store-set predictor.

correctly infer when all of the in-flight stores commit without being affected by just one of the stores' retirement.

The store-load pair predictor's counter increments when a store is fetched and decrements when the store commits. When a load that is predicted to be dependent is ready to issue, it accesses both the valid bit and the counter. If the counter is larger than zero, the load is predicted as being dependent on the stores in flight, and the load must search the store queue.

2.1.2 Low Cost Implementation

In the simplest design, the store-set predictor and store-load pair predictor may be implemented separately. However, because the two predictors use identical structures, we combine the two so that the predictors share the same physical tables (i.e., SSIT and LFST). This strategy lowers the implementation cost. The only change is that each LFST entry includes both a valid bit and a counter. From the store-load pair predictor's point of view, the LFST entry is valid when the counter is non-zero; whereas from the store-set predictor's point of view, the same LFST entry is valid when the valid bit is set. Therefore, the same LFST entry has two meanings of validity (or invalidity) depending on which predictor is accessing the entry.

In the event of recovery from misspeculations such as branch mispredictions or memory-dependence violations, the SSIT and LFST do not need to be modified during roll-back. However, squashed stores should roll back each counter properly for accuracy of future prediction. To model this extra work in recovery, we charge an extra cycle to our misprediction penalty. Our study shows that a three-bit counter is large enough to achieve high prediction accuracy.

By definition, store-load pair prediction subsumes store-set prediction. Therefore, only the store-load pair predictor updates the prediction tables, causing two problems for store-set prediction.

First, as shown in Figure 2, the store-load pair predictor has to keep the pairing information for not only the store^0-load^1 pair but also the store^0-load^2 pair in the prediction table. However, from the store-set predictor's point of view, keeping the store^0-load^2 pair in the prediction table is not necessary and would reduce the effective size of the table. Such unnecessary store-load pairs may increase aliasing in the store-set prediction, leading to false alarms. We use a 4K-entry SSIT as was used in the previous work [2], and our experiments show that a 4K-entry table is large enough to absorb this extra pressure on the SSIT.

Second, having the unnecessary store^0-load^2 pair in the prediction table causes the store-set predictor to find a valid entry for load^2, even though load^2 has not incurred any store-load order violation. However, store^0 issues and invalidates the entry of the store^0-load^2 pair before load^2 issues. Therefore, load^2 issues without waiting for store^0. Hence, such unnecessary store-load pairs do not affect performance. Figure 3 summarizes the duties of the store-set predictor and store-load pair predictor in the pipeline. Statements in bold are performed by the store-load pair predictor.

2.2 Reducing Load Queue Search: Load Buffer

Modern processors send stores to the cache in program order. However, loads are handled differently. Because load values are needed for the computation to proceed, most processors allow loads to access the cache out of program order. Servicing loads out of order causes correctness problems in the context of memory consistency models for shared-memory multiprocessors. Some processors use the load/store queue to avoid the correctness problems.

Specifically, there is a special case of load-load ordering when the loads go to the same address. If this ordering is relaxed, subtle correctness problems arise: if two loads to the same address are issued out of order and the value is changed by another processor in between the two loads [1], the later load will obtain an earlier value whereas the earlier load will obtain a later value. Note that this problem cannot happen if the value is changed by a store from the same processor, because the store would detect the later load to be premature and squash the load along with all subsequent instructions.

This load-load ordering problem may be handled by either software or hardware. We explain the software option first. If the consistency model supported is a relaxed one, processors provide a "memory barrier" instruction to allow the programmer to enforce ordering among memory operations wherever needed. The details depend on the specifics of the particular relaxed model implemented. The programmer is expected to use the instruction to prevent the problem.

However, extensive use of memory barriers is an overkill and hurts performance. To spare the programmer from using too many memory barriers, some processors provide hardware support to prevent this problem by guaranteeing load-load ordering for loads to the same address (e.g., Alpha [3]). This guarantee is typically provided by using one of two schemes. (1) *Any* load violating load-load ordering is squashed regardless of whether an intervening store (from another processor) occurs or not. (2) Squash is done *only when* an intervening store is detected.

In the first scheme, the load queue is searched by *every* load to ensure that no later load to the same address executes out of order and obtains an old value. When a load executes, it searches the load queue to compare its address to the addresses of all loads. If there is a match with a younger load issued out of program order, then the out-of-order-issued load and subsequent instructions are squashed and fetched again. While this search reduces the burden on programmers and improves performance, the search bandwidth pressure on the load queue greatly increases. Therefore, we target these searches and optimize them.

The second scheme does not significantly increase the load queue search bandwidth. The invalidation signal from the other processor's store is used to detect ordering violations. For example, the MIPS R10000 uses the invalidation signal to find *all* outstanding loads (including premature loads) when a shared value is changed by another processor [9]. The invalidation searches the load queue and if *any* matching outstanding load (premature or otherwise) is found, the load and the subsequent instructions are squashed. The fact that the Alpha supports a relaxed consistency model and the MIPS supports sequential consistency does not matter here; load-load ordering causes problems in all models. Because invalidations are significantly less frequent than out-of-order loads, and because invalidations may be filtered further by L2 or L3 caches, the load queue searches caused by invalidations may not need any special bandwidth-reduction technique. Therefore, we do not address this invalidation-caused search bandwidth.

To maximize performance, some recently-announced processors (e.g., POWER4 [10]) implement a combination of the two schemes. However, the POWER4 also searches the load queue to identify out-of-order-issued loads. Because the industry trend is in the direction of performing load queue searches, we optimize these searches.

To that end, we observe that an issued load needs to search only those loads that were issued out of order before previous non-issued loads. Across the SPEC2K benchmarks, the average number of out-of-order-issued loads is small (< 3). We use this observation to reduce the search bandwidth demand on the load queue. We employ an extra buffer with fewer than four entries, called the *load buffer*, to keep only out-of-order-issued loads separate from the load queue. When a load executes, instead of searching the entire load queue, it searches the load buffer, which is much smaller than the load queue.

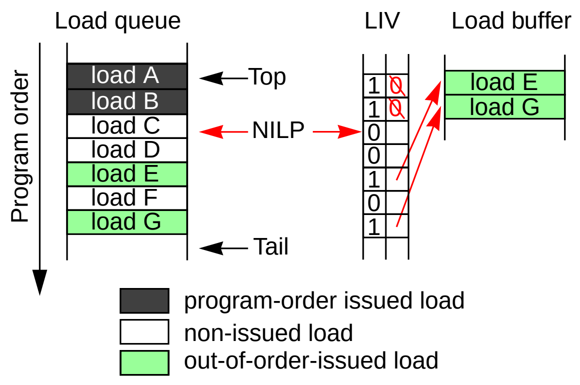


Figure 4: Load buffer implementation.

Figure 4 shows the basic idea of how the load buffer works. When *load E* and *load G* issue, there are older non-issued loads, namely *load C* and *load D*. Therefore, *load E* and *load G* are out-of-order-issued loads. When *load E* and *load G* execute, they put their addresses in the load buffer. When *load E* does not have any older non-issued load, it invalidates its relevant load buffer entry. To find out-of-order-issued loads, *load E* searches the load buffer and compares its address against the address of *load G*. When *load C* issues, there is no older non-issued load. Therefore, *load C* does not put itself in the load buffer. When *load C* executes later, it searches the load buffer for out-of-order-issued loads and compares its address against the load buffer (i.e., only *load E* and *load G*) instead of comparing against the entire load queue.

2.2.1 Implementing the Load Buffer

The key implementation concern for the load buffer is how to allocate and access entries in the load buffer *without accessing* the load queue. We employ the Non-Issued Load Pointer (NILP) and Load Issue Vector (LIV) to handle this concern. The NILP points to the oldest non-issued load entry. Top points to the oldest load entry, and Tail points to the next available entry for a new-coming load. The pointer starts from Top and cannot go beyond Tail. The NILP moves toward Tail when the load to which it points issues, and the pointer skips over loads that are already issued, so that NILP always points to the oldest non-issued load.

To avoid accessing the load queue for updating the NILP, we employ the Load Issue Vector (LIV). The LIV has the same number of entries as the load queue itself. In each entry, there is a single issue bit and a pointer to a load buffer entry. Each issue bit in the LIV corresponds to an entry in the load queue. The bit is set when the relevant load issues, and the bit is reset when the relevant load commits or is invalidated due to misspeculation.

When a load issues, it checks where the NILP is pointing. If the NILP is pointing to the LIV entry for the load, it means that the load is the oldest non-issued load. Hence, this load searches the load buffer to detect the load-load order violation as soon as it has a valid address. Also, the load sets its LIV bit, which triggers the NILP to move toward Tail. The NILP skips over any entry whose issue bit is already set, so that the NILP will always stop at an LIV entry whose issue bit is cleared. If the NILP is not pointing to the LIV entry for the load, then the load is not the oldest load among non-issued instructions. For this case, if there is an available entry in the load buffer, the load copies its address to the load buffer entry as soon as it has a valid address. The pointer of the LIV entry should point to this newly-allocated load buffer entry and the LIV bit is set.

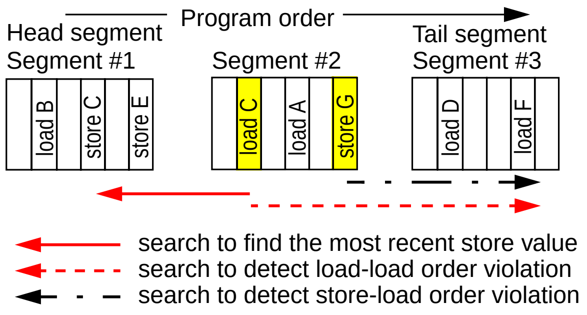
If the load buffer is full, the load stalls until there is an available entry in the load buffer or until the NILP points to the LIV entry for this load, which means that this load can be issued in program order and elides the load buffer. Such a stall mechanism is similar to what the store-set predictor uses to stall a load that has non-issued dependent stores in the pipeline.

Load buffer entries can be released and reused in the following way. As the NILP moves toward Tail, if the NILP encounters an LIV entry with an issue bit that is already set, then the load corresponding to this LIV entry was issued out of order, and the load occupies an entry in the load buffer. The load buffer entry that is pointed to by the LIV entry can now be invalidated, because the corresponding load does not have any older non-issued load and cannot cause a load-load order violation after that point in time. Then, this load buffer entry can be used by later out-of-order-issued loads. At this time, the load relevant to the LIV entry has to search the load buffer for an out-of-order-issued load.

3 Increasing Queue Capacity

We apply segmentation to increase the size of the load/store queue by concatenating multiple smaller load/store queues serially. We allocate a load/store queue entry for each memory instruction within a segment. If a segment runs out of entries to allocate, we move on to the next segment. Figure 5 illustrates the basic idea of the segmented load/store queue. For brevity, we show load and store instructions together in the same queue. We call the segment with the oldest memory instruction the *head segment* and the segment with the youngest memory instruction the *tail segment* in this study.

A memory instruction first performs a memory dependence search on the segment corresponding to its load/store queue entry, and the search extends to other segments as needed. As shown in Figure 5, *load C* (or *store G*) searches its own segment first. If there is no match within the segment, the search continues to other segments. If the goal of the search is to find the most recent value from a matched store, the search goes to the previous segment and continues searching toward the head segment until it finds a match or completes the search. If the goal of the search is to detect a load-load order violation or a store-load order violation, the search continues to the next segment toward the tail segment until it detects a violation or completes the search.



Program order
Head segment
Segment #1
Segment #2
Tail segment
Segment #3
load B
store C
store E
load C
load A
store G
load D
load F
search to find the most recent store value
search to detect load-load order violation
search to detect store-load order violation

Figure 5: Memory disambiguation in the segmented load/store queue.

However, such multiple searches through different segments cause two design challenges. First, searching multiple segments to find the latest store value requires extra cycles that impact load hit latency. This searching also makes the hit latency variable. For high performance, superscalar processors speculatively schedule instructions dependent on the load with the assumption that the load is a cache hit. Variable hit latencies may complicate such a scheduling mechanism. To avoid complicating the scheduler, we forego early scheduling for the instructions that are dependent on the load. However, there is one important exception to this rule: if the load is in the head segment, it does not need to search any previous segment because no previous segment exists, so the hit latency for the load is constant. Therefore, we keep performing early scheduling for the load-dependent instructions if the load is from the head segment.

Second, unlike the conventional load/store queue which limits the total number of searches to the number of ports, the segmented load/store queue may have more simultaneous searches than the number of ports. As long as such searches are distributed among the segments without any conflicts, the segmented load/store queue can pipeline searches and can handle more simultaneous searches than the conventional load/store queue.

For example, assuming a two-ported load/store queue, the conventional queue allows any combination of two memory accesses *in total* at one cycle, whereas the segmented queue allows any combination of two memory accesses *for each segment* at one cycle. In Figure 5, *store C* and *store E* of segment #1 are allowed to search the load queue to detect a store-load order violation at t_1. Meanwhile, *load D* and *load F* of segment #3 can search the load queue to detect a load-load order violation in the same cycle.

3.1 Allocation

Strategies for allocation of a new entry in the segmented load/store queue allow a trade-off between the above issues of latency and bandwidth. We consider two such strategies. One strategy is *no-self-circular,* which spreads out entries across many segments, and the other is *self-circular,* which compacts entries into only a few segments. Spreading out entries provides higher aggregate bandwidth by using many segments simultaneously, but it increases latency by having to search more segments. Compaction has the opposite effect.

The *no-self-circular* treats all segments as a single queue, and allocation of a new entry uses a single head and tail. Allocation moves linearly from one segment to the next even if the current segment has free entries at the top. The *self-circular* treats the segmented load/store queue as ordered segments with a head and a tail for each segment. In this method, allocation of new entries is circular within each segment. Allocation moves to the next segment only if there are no free entries at the top of the current segment. Thus, no-self-circular spreads out allocation across many segments while self-circular tends to restrict allocation to fewer segments.

3.2 Contention

Segmentation introduces a contention problem. Going back to the example in Figure 5, there are two situations that can cause port contention problems in cycle t_2. *store C* and *store E* from segment #1 go to segment #2 to continue the search for a store-load order violation at cycle t_2. In one contention situation, segment #2 may initiate *store G* to search for a store-load order violation. Then the total number of searches required in cycle t_2 in segment #2 will exceed segment #2's search bandwidth. This situation is easily solved by delaying the commit of the store, because the store is not in the pipeline anymore.

The other contention situation occurs if segment #2 initiates *load C* and *load A* to search for a load-load order violation while *store C* and *store E* reach segment #2 in cycle t_2. We cannot simply delay the searches because the loads are already in the memory stage of the pipeline. Fortunately, this situation can happen only when several conditions are *simultaneously* satisfied: (1) more than one load is issued or a store starts the search for a store-load order violation from a segment #A at cycle t_a, and more than one load is issued out of order from a segment #B at cycle t_b, (2) segment #A is closer to the head segment than segment #B, (3) the time t_a is earlier than the time t_b, and (4) the distance between two segments is the same as the time difference between t_a and t_b. However, we found that the average number of out-of-order-issued load instructions in the pipeline is small (< 3) (Section 4.1.2). Therefore, the probability of all four conditions being satisfied at the same time is low.

If contention does occur, we squash *load C* and *load A*, along with all instructions between the issue stage and the execute stage, similar to a flush due to a load miss in conventional pipelines. Alternatively, we can stall the pipeline between the issue stage and the memory stage until the port contention is resolved. Because our previous techniques significantly reduce the load/store queue search bandwidth, the port contention rarely occurs when the segmented load/store queue is combined with the techniques.

Table 1: System configuration parameters.

ROB size	256 entries
Issue queue	64 entries
Issue width	8
Functional units	8 integer, 8 pipelined floating-point
Register file	356 INT/ 356 FP
2-port L1 i-cache & 4-port L1 d-cache	64K 2-way, pipelined 2-cycle hit, 32-byte block
L2 cache	2M 8-way, pipelined 12-cycle hit, 64-byte block
Memory	150 cycles
Store-set predictor with store-load pair	4K-entry SSIT, 128-entry LFST
Branch predictor Mispredict penalty	hybrid GAg & PAg 4K-entries each, 14 cycles

4 Results

We have built a cycle-accurate simulator of an out-of-order superscalar pipeline. Table 1 shows the base configuration for the experiments. Because we vary the number of load/store queue ports from 1 to 4, we assume a four-ported d-cache so that the d-cache does not limit the number of requests to the load/store queue. Because we vary the load/store queue size from 32 to 128, we assume a 256-entry active list so that the active list does not limit the number of in-flight memory instructions.

We use the SPEC2K applications with the reference input set. We skip the first 3 billion instructions and then simulate a total of 500 million instructions. Table 2 shows the applications we use in this study and their base IPCs.

In Section 4.1, we examine the effect of the store-load pair predictor and the load buffer in isolation on the search bandwidth and performance. In Section 4.2, we examine the impact of increasing the capacity of the load/store queue with segmentation. In Section 4.3, we combine the three techniques and examine their impact on processor performance.

Table 2: Applications and their base IPCs

INT		INT		FP		FP	
bzip	2.5	perl	3.0	ammp	1.2	mgrid	2.2
gcc	2.1	twolf	1.5	applu	2.6	sixtrack	2.9
gzip	2.0	vortex	2.2	art	0.3	swim	1.0
mcf	0.3	vpr	1.3	equake	1.1	wupwise	2.9
parser	1.9			mesa	3.3		

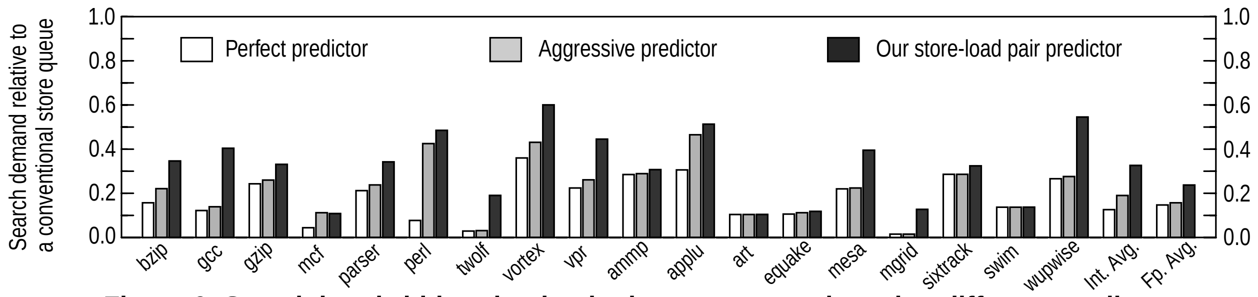


Figure 6: Search bandwidth reduction in the store queue by using different predictors.

4.1 Reducing Search Bandwidth Demand

In this section, we present the results of the store-load pair predictor and the load buffer separately. Then we present the results of combining these two techniques.

4.1.1 Reducing Store Queue Search

In this subsection, we evaluate our technique for reducing the search bandwidth demand on the store queue. We apply the store-load pair predictor to predict whether a load is dependent on any earlier store. A load that is predicted to be independent does not search the store queue, thus reducing the search bandwidth demand on the store queue. In Figure 6, the Y axis shows the search bandwidth demands on the store queue of three different predictors (perfect, aggressive and store-load pair) normalized to the search bandwidth of the base case. The X axis shows our benchmarks and the average of the integer and the floating-point programs separately. The base case is a two-ported conventional load/store queue in which all loads search the store queue. We choose two ports because that is a commonly-used design point in current high-performance processors.

The perfect predictor flags a search for only those loads that are dependent on an in-flight store. The aggressive predictor uses unrealistic hardware to emulate an alias-free version of our store-load pair predictor — i.e., store sets which may conflict in our tables do not do so in the aggressive predictor. Consequently, with the aggressive predictor, a load avoids searching the store queue as much as possible. In comparison, because our store-load pair predictor uses realistic hardware, as described in Section 2.1, it incurs aliasing (only as much as the original store-set predictor). Therefore, our predictor ends up being more conservative in predicting a load to be independent.

Figure 6 shows that the store queue with a perfect predictor (left bar) reduces the search bandwidth of the base case by 86% (demand is 14% of the base case). The aggressive predictor (middle bar) manages to reduce the search bandwidth by 81% for integer benchmarks, and by 84% for floating-point benchmarks. Our store-load pair predictor (right bar) manages to reduce the search bandwidth demand on the store queue by 67% for integer benchmarks and by 76% for floating-point benchmarks compared to the base case.

In Figure 7, the Y axis shows the performance benefit of the predictors compared to the same base case as Figure 6. We see that the perfect predictor does not achieve much improvement, even though it reduces the search bandwidth demand by 86%. Recall that the base case uses two store queue ports, which provide sufficient bandwidth. Therefore, reducing the demand does not translate to performance in this figure. The benefit of reducing the demand will be seen in Section 4.1.3, where we show a 1-ported store queue.

The aggressive predictor performs *worse* than the base case in a few cases by as much as 19%, even though it reduces the search bandwidth demand by more than 80%. The lack of aliasing in the aggressive predictor also implies lack of constructive interference (a similar effect is reported in [2]). Consequently, the aggressive predictor is overly eager to predict a load to be independent of earlier stores. The resulting high misprediction rate causes a large number of squashes that degrade performance, especially for *vortex* and *wupwise.* Our store-load pair predictor is more conservative and does not incur misprediction squashes as frequently as the aggressive predictor. As a result, our store-load pair pre-

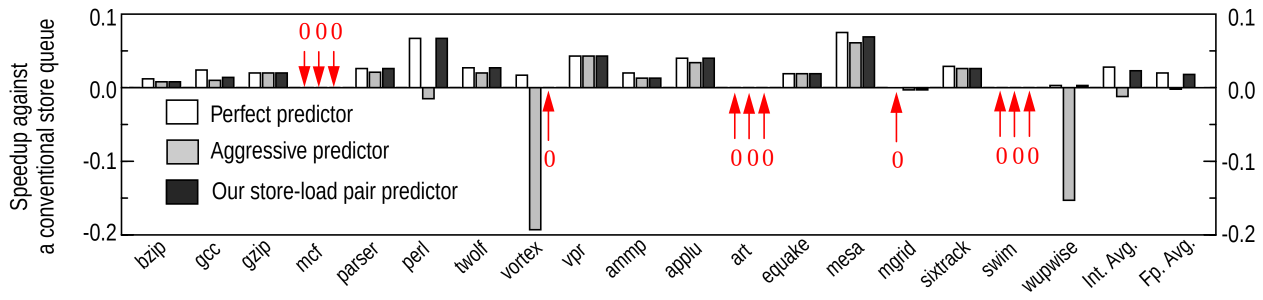


Figure 7: Performance benefit from the search bandwidth reduction in the store queue.

Table 3: Accuracy of the store-load pair predictor.

INT	Mispred.	Squash	FP	Mispred.	Squash
bzip	17.3%	$7.3x10^{-5}$	ammp	4.3%	$3.9x10^{-4}$
gcc	27.8%	$8.4x10^{-4}$	applu	22.2%	$8.0x10^{-5}$
gzip	9.0%	$6.7x10^{-5}$	art	0.0%	$3.6x10^{-5}$
mcf	8.1%	$1.1x10^{-4}$	equake	0.9%	$1.6x10^{-5}$
parser	12.3%	$2.1x10^{-4}$	mesa	15.5%	$2.4x10^{-4}$
perl	15.5%	$7.8x10^{-5}$	mgrid	8.6%	$7.6x10^{-6}$
twolf	5.0%	$2.8x10^{-5}$	sixtrack	4.3%	$1.3x10^{-4}$
vortex	19.1%	$2.2x10^{-3}$	swim	0.0%	$1.4x10^{-5}$
vpr	22.3%	$1.2x10^{-4}$	wupwise	24.7%	$6.6x10^{-5}$

dictor does not incur performance degradation in *vortex* and *wupwise*. Our store-load pair predictor performs comparably to the perfect predictor, and it outperforms the base case on average by 2% and up to 7% for both integer and floating-point benchmarks.

Table 3 shows the accuracy of our store-load pair prediction. The misprediction causes pipeline squashes or unnecessary searches in the store queue. The results show that our predictor successfully avoids squashes without sacrificing the opportunity to save bandwidth.

4.1.2 Reducing Load Queue Search

In this subsection, we evaluate our technique for reducing the search bandwidth demand on the load queue. The load buffer separates the search for detecting a load-load order violation from the load queue. Thus, the load queue only needs to support associative searches by stores to detect store-load order violations. Therefore, we expect the load buffer to reduce substantially the search bandwidth demand on the load queue.

In Figure 8, the Y axis shows the search bandwidth reduction in the load queue by using a load buffer with two entries. The base case is a conventional load queue without the load buffer. In the figure, the search bandwidth in the load queue with the load buffer is normalized to the search bandwidth in the load queue of the base case. The load buffer reduces the search bandwidth demand on the load queue by an average of 74% for integer benchmarks and 77% for floating-point benchmarks. In *mgrid*, the load buffer achieves the

Table 4: Average number of loads issued out of program order.

INT		INT		FP		FP	
bzip	3.4	perl	3.2	ammp	1.2	mgrid	2.9
gcc	0.3	twolf	1.0	applu	1.5	sixtrack	1.0
gzip	0.8	vortex	1.9	art	3.4	swim	0.9
mcf	0.2	vpr	1.5	equake	2.5	wupwise	2.3
parser	0.8			mesa	0.9		

most benefit by reducing the search bandwidth demand by 96%. The reason for this large reduction is that 51% of dynamic instructions in *mgrid* are loads and just 2% are stores. *Vortex* shows the least reduction in the search bandwidth demand. This result is not a surprise when we consider that just 18% of dynamic instructions are loads and 23% are stores. Therefore, even though the load buffer removes the searches required for detecting a load-load order violation from the queue, the load queue still has a significant number of searches from stores for detecting store-load order violations in *vortex*.

Table 4 shows the average number of out-of-order-issued loads in flight every cycle. This average indicates how large the load buffer needs to be (as mentioned in Section 2). Even though these numbers are rather small across benchmarks, the performance impact of out-of-order-issued loads is significant. We can see the impact of these loads when we look at the first two bars in Figure 9.

Figure 9 illustrates the performance benefit of using the load buffer. An N-entry load buffer allows up to N loads to be issued out of program order. We vary N as 1,2 and 4. In addition, we also show two designs that issue loads in order, to justify out-of-order issue of loads. Note that in these designs loads are in order only with respect to each other. In-order issue of loads has two different impacts on performance: (1) the ILP reduces, but (2) the bandwidth pressure on the load queue also reduces (because there is no need to search the load queue for load-load order violations). The *in-order-always-search* load queue (the leftmost bar) fruitlessly searches the load queue to incur not only the loss of ILP but also the bandwidth pressure of an out-of-order issue load queue. On the other hand, the *zero-entry* load buffer (the next bar in white) does not search the load queue and therefore incurs only the loss of ILP without the bandwidth pressure.

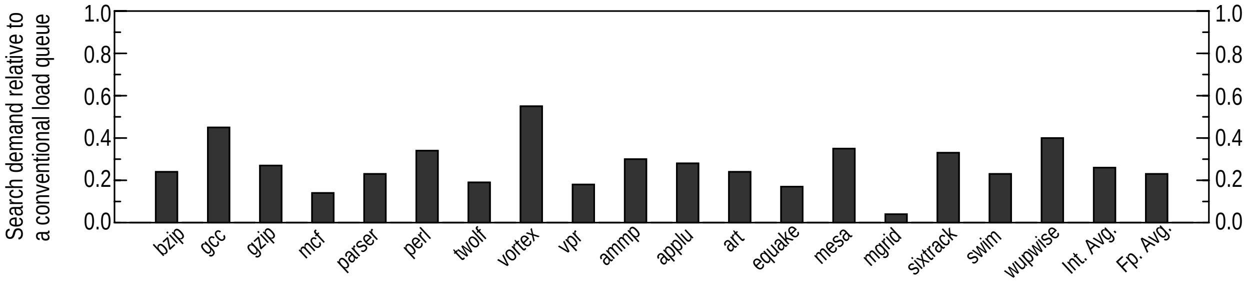


Figure 8: Search bandwidth reduction in the load queue by using the load buffer.

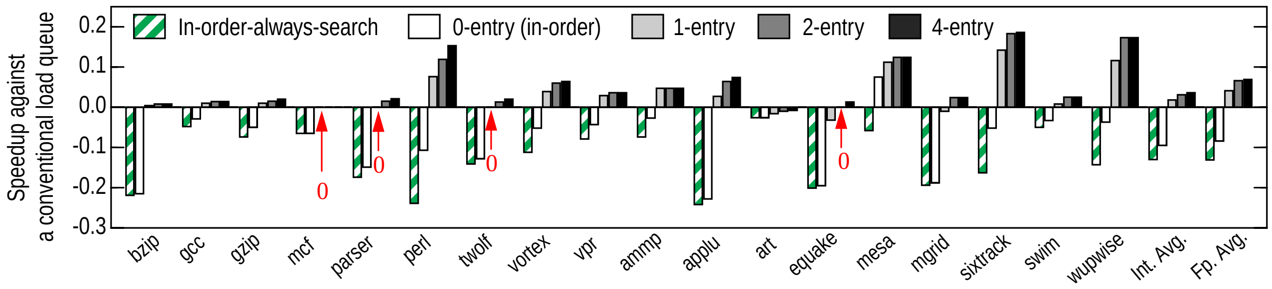


Figure 9: Performance benefit from the search bandwidth reduction in the load queue.

The base case is a conventional load queue without the load buffer.

From the figure, the first two bars show that in-order issue of loads performs poorly. The zero-entry load buffer's bandwidth-pressure advantage allows it to perform better than in-order-always-issue. However, the advantage is not enough to overcome the ILP loss of in-order issue. We also see that by allowing only one load to be issued out of order, we can realize the significant benefit of the load buffer. A four-entry load buffer is close to an infinite load buffer in terms of performance, but we can see most of the benefit with just two entries. The figure shows that the load/store queue with a two-entry load buffer improves performance by an average of 3% and 7% with a maximum of 12% and 18% for integer and floating-point benchmarks, respectively.

4.1.3 Combining the Two Reduction Techniques

In this subsection, we combine the store-load pair predictor with the load buffer to reduce the search bandwidth demand on both the store queue and the load queue. Figure 10 shows the performance of a load/store queue with our two reduction techniques relative to the base case, which is a two-ported conventional load/store queue.

From left to right, the bars represent a one-ported conventional queue, a one-ported queue with our techniques, a two-ported queue with our techniques, and a four-ported conventional queue. The leftmost bar represents the extreme case of low design complexity and low performance, and the rightmost bar represents the extreme case of high design complexity and high performance.

The figure shows that the performance of the conventional one-ported load/store queue drops by 24% from the conventional two-ported load/store queue. This result says that without our techniques, a one-ported load/store queue is not an option because of its poor performance. However, the *one-ported* load/store queue with our search bandwidth reduction techniques achieves an average speedup of 2% and 7% with a maximum of 7% and 25% for integer and floating-point benchmarks, respectively, over the base case, which is the conventional *two-ported* load/store queue. We can also see that the two-ported load/store queue with our techniques performs comparably to the conventional four-ported load/store queue.

4.2 Increasing Queue Capacity

In this section, we show the result of segmenting the load/store queue to increase its capacity. Figure 11 shows the effect of the segmented load/store queue. The base case has a 32-entry load queue and a 32-entry store queue. We segment the load/store queue into four segments with 28 entries in each segment for a total size of 112. We keep the access latency of each segment the same as the base case, but we make each segment smaller than the base case to compensate for any potential overhead that segmentation would introduce. A search within one segment takes one cycle, and each additional segment takes an extra cycle.

In the figure, we show the results of our two methods for allocation, no-self-circular and self-circular. Recall from Section 3.1 that no-self-circular spreads out allocation across many segments while self-circular tends to compact allocation within fewer segments.

On average, the no-self-circular shows no speedup for integer benchmarks and 16% speedup for floating-point benchmarks. Five (bzip, gcc, gzip, parser, and twolf) of the

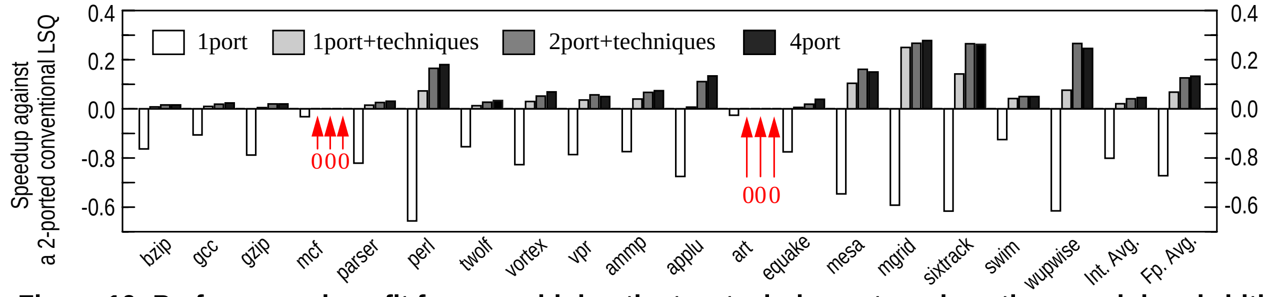


Figure 10: Performance benefit from combining the two techniques to reduce the search bandwidth.

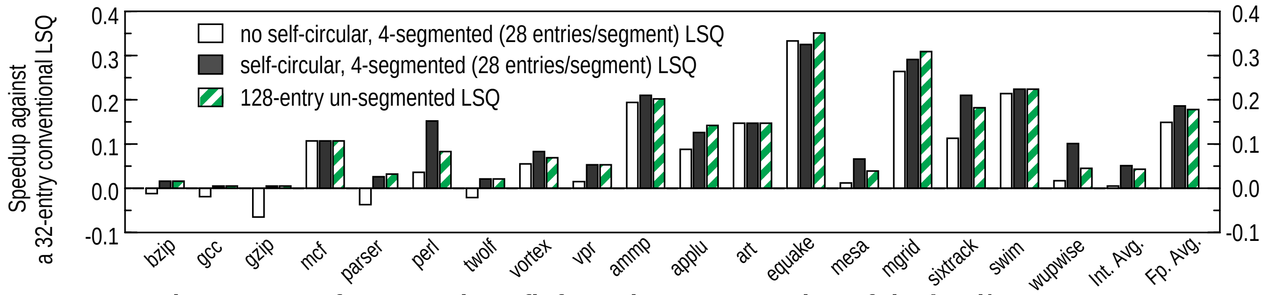


Figure 11: Performance benefit from the segmentation of the load/store queue.

nine integer benchmarks actually perform worse than the base case. We explain this poor performance in Table 5 by showing the average number of entries needed in the load/ store queue. We see that in the five benchmarks (shaded) the average number of entries needed in these benchmarks can fit within one segment. However, no-self-circular spreads their entries across two segments, incurring extra search overhead that does not exist in the base case. Therefore, no-self-circular does not perform as well as the base case when there is less demand for higher capacity of the load/store queue. On the other hand, even though *vortex* does not need a large load queue, its high demand on the store queue explains its speedup using the segmented load/store queue.

The self-circular achieves an average speedup of 5% and 19% with a maximum speedup of 15% and 33% for integer and floating-point benchmarks, respectively. By restricting the entries within one segment as much as possible, self-circular reduces the possibility of spanning load/store queue entries across two segments. Therefore, self-circular manages to outperform no-self-circular. Because segmentation provides higher bandwidth by using many segments simultaneously, self-circular outperforms the unrealistic 128-entry unsegmented load/store queue as well (Section 3). *Equake* and *mgrid* show the most benefit from higher queue capacity due to the fact that 42% and 51% of their dynamic instructions are loads.

Table 6 shows the distribution of the number of searched segments by loads for the latest stores using the self-circular. For example, in *mcf,* 84.4% of the load accesses search only one segment, 9.9% search two segments, 0.3% search three segments, and 5.4% search all four segments to find a latest store value from the store queue. For integer benchmarks, 90% of the load accesses search only one segment, while the same is true for 79% of the load accesses for floating-point benchmarks. This table shows that the extra search cycle due to segmentation is not likely to hurt the average load hit latency (Section 3) because the majority of loads end their searches within one or two segments.

Table 5: Average number of entries needed in the load and store queues.

INT Benchmarks	Avg.(ld/st)	FP Benchmarks	Avg.(ld/st)
bzip	**16 / 6**	ammp	65 / 28
gcc	**7 / 6**	applu	49 / 19
gzip	**14 / 7**	art	49 / 17
mcf	40 / 9	equake	72 / 15
parser	**21 / 9**	mesa	33 / 20
perl	34 / 20	mgrid	90 / 4
twolf	**18 / 6**	sixtrack	60 / 30
vortex	13 / 18	swim	70 / 21
vpr	41 / 15	wupwise	47 / 31

4.3 Combining Search Bandwidth Reduction and Higher Capacity

In this section, we combine the segmented load/store queue with the store-load pair predictor and the load buffer to see the overall benefit from these techniques. Apart from the base processor configuration, we also show a scaled processor to see the benefit of our techniques in the future. To scale the processor, we increase the issue width from 8 to 12, the issue queue size from 64 to 96, and the L1 hit latency from 2 to 3 cycles, while keeping the cache size the same. For both processors, we use a two-entry load buffer, self-circular allocation, and four 28-entry segments. Figure 12 shows the performance of the *one-ported* load/store queues with our three techniques compared to a *two-ported* conventional load/store queue. The white bar shows the speed up of our techniques on the base processor, and the dark bar shows the speed up on the scaled processor.

The figure shows that with today's processor, the one-ported load/store queue with our techniques improves perfor-

Table 6: Distribution of the number of searched segments by loads for the latest stores.

Bench.	1	2	3	4	Bench.	1	2	3	4
bzip	97.8	1.4	0.0	0.8	ammp	74.0	13.8	0.3	11.9
gcc	98.0	1.5	0.0	0.6	applu	78.1	11.4	0.0	10.5
gzip	97.7	1.8	0.0	0.5	art	89.1	5.9	0.1	4.9
mcf	84.4	9.9	0.3	5.4	equake	75.3	13.9	0.1	10.7
parser	93.3	5.0	0.0	1.7	mesa	74.7	14.8	0.2	10.3
perl	81.5	13.5	0.0	5.0	mgrid	94.1	3.6	0.0	2.2
twolf	92.5	6.1	0.0	1.4	sixtrack	71.8	17.0	0.2	11.0
vortex	79.2	13.9	0.3	6.6	swim	81.3	9.6	0.0	9.1
vpr	84.0	13.9	0.0	2.1	wupwise	74.9	15.1	0.4	9.6

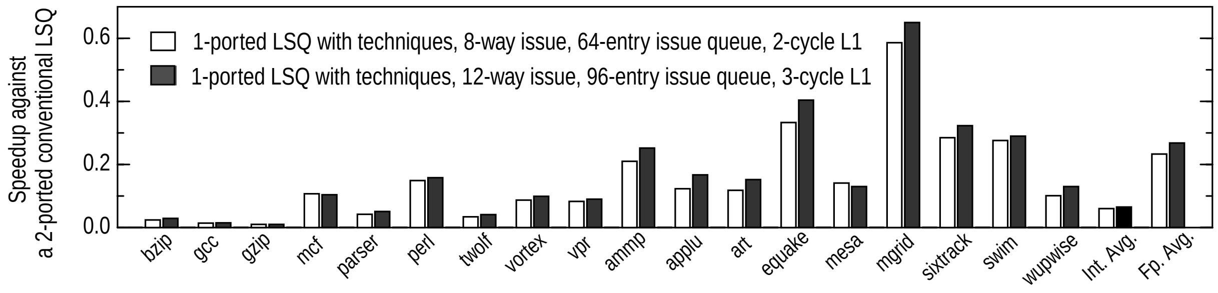


Figure 12: Performance of a one-ported ld/st queue with the three techniques combined.

mance on average by 6% and 23%, and up to 15% and 59% for integer and floating-point benchmarks, respectively, over the two-ported conventional load/store queue. As a processor scales, there are more in-flight instructions in the pipeline, hence there is more pressure on the load/store queue. With a scaled system, we can see that our techniques further improve performance, especially for floating-point benchmarks. Floating-point benchmarks have more instruction-level parallelism than integer benchmarks. Therefore, floating-point benchmarks can utilize the extra hardware to overlap more instructions so that these programs put more pressure on the load/store queue.

5 Conclusions

A modern load/store queue holds in-flight memory instructions and supports simultaneous searches to honor memory dependences and memory consistency models. In this paper, we proposed three techniques to scale the load/store queue; two of these techniques — the load-store pair predictor and load buffer — reduce the search bandwidth demand on the load/store queue, and the other technique — segmentation — increases the capacity of the load/store queue. Using SPEC2K benchmarks, we showed that our predictor reduces the search bandwidth demand by 72% in the store queue and a two-entry load buffer reduces the search bandwidth demand by 76% in the load queue. Then, we showed that a load/store queue that combines these two techniques needs only one port to outperform a conventional load/store queue with two ports. We also showed that our segmentation improves performance on average by 5% and 19%, and up to 15% and 33%, for integer and floating-point benchmarks, respectively. Finally, our results show that a *one-ported* load/store queue using all our techniques combined improves performance on average by 6% and 23%, and up to 15% and 59%, for integer and floating-point benchmarks, respectively, over the *two-ported* conventional load/store queue.

Our techniques not only reduce the complexity of the load/store queue design but also improve performance by reducing the search bandwidth demand and increasing the capacity. Our techniques will become more important as the pressure on the load/store queue increases with the scaling of processors in the future.

Acknowledgements

We would like to thank Chad Scarbrough and the anonymous reviewers for feedback on earlier drafts of this paper. This research is supported in part by NSF CAREER award 9875960-CCR and Instrumentation grant CCR-9986020.

References

[1] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. Technical Report 9512, Department of Electrical and Computer Engineering, Rice University, September 1995.

[2] George Z. Chrysos and Joel S. Emer. Memory dependence prediction using store sets. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 142–153, June 1998.

[3] Compaq Computer Corporation. *Compiler Write's Guide for the 21264/21364*, January 2002.

[4] Daniele Folegnani and Antonio Gonzalez. Energy-effective issue logic. In *Proceedings of the 28th International Symposium on Computer Architecture*, pages 230–239, June 2001.

[5] Alvin Lebeck, Jinson Koppanalil, Tong Li, Jaidev Patwardhan, and Eric Rotenberg. A large, fast instruction window for tolerating cache misses. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 59–70, June 2002.

[6] Andreas I. Moshovos, Scott E. Breach, T.N. Vijaykumar, and Gurindar S. Sohi. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 181–193, June 1997.

[7] Il Park, Michael D. Powell, and T. N. Vijaykumar. Reducing register ports for higher speed and lower energy. In *Proceedings of the 35th International Symposium on Microarchitecture*, pages 171–182, November 2002.

[8] Steven E. Raasch, Nathan L. Binkert, and Steven K. Reinhardt. A scalable instruction queue design using dependence chains. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 318–329, June 2002.

[9] Silicon Graphics, Inc. *MIPS R10000 Microprocessor User's Manual version 2.0*, October 1996.

[10] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le Jr., and B. Sinharoy. Power4 system microarchitecture. *IBM Journal of Research and Development*, 46(1), October 2002.