A Two-Level Load/Store Queue Based on Execution Locality

Miquel Pericàs[†⋆], Adrian Cristal[⋆], Francisco J. Cazorla[⋆],
Ruben González[†], Alex Veidenbaum[◇], Daniel A. Jiménez[‡] and Mateo Valero[†⋆]

[†]Universitat Politècnica de Catalunya, {mpericas,gonzalez,mateo}@ac.upc.edu
[⋆]Barcelona Supercomputing Center, {adrian.cristal,francisco.cazorla}@bsc.es
[‡]The University of Texas at San Antonio, djimenez@acm.org
[◇]The University of California at Irvine, alexv@ics.uci.edu

Abstract

Multicore processors have emerged as a powerful platform on which to efficiently exploit thread-level parallelism (TLP). However, due to Amdahl's Law, such designs will be increasingly limited by the remaining sequential components of applications. To overcome this limitation it is necessary to design processors with many lower–performance cores for TLP and some high-performance cores designed to execute sequential algorithms. Such cores will need to address the memory-wall by implementing kilo-instruction windows.

Large window processors require large Load/Store Queues that would be too slow if implemented using current CAM-based designs. This paper proposes an Epoch-based Load Store Queue (ELSQ), a new design based on Execution Locality. It is integrated into a large-window processor that has a fast, out-of-order core operating only on L1/L2 cache hits and N slower cores that process L2 misses and their dependent instructions. The large LSQ is coupled with the slow cores and is partitioned into N small and local LSQs, one per core.

We evaluate ELSQ in a large-window environment, finding that it enables high performance at low power. By exploiting locality among loads and stores, ELSQ outperforms even an idealized central LSQ when implemented on top of a decoupled processor design.

1 Introduction

Placing multiple cores on a die is an effective way to increase the raw execution bandwidth of the chip. In addition to reduced design complexity, multicores enable performance at a power budget that is superior to traditional scaling.

Nevertheless, multicores have some problems. Applications are difficult to parallelize. In the extreme case, sequential algorithms might not allow any high-level parallelization, yet might still have a significant amount of instruction-level parallelism (ILP). It is important to maximize the opportunities to exploit both thread-level parallelism (TLP) and ILP to maximize system performance. One approach is to design a heterogeneous processor with large cores capable of holding thousands of instructions to exploit ILP and small cores to exploit TLP.

Past research has focused on designing large window processors with proposals such as the WIB [1], CFP [2, 3], TRIPS [4] or KILO-Instruction processors [5, 6]. For many applications, a large window processor can exploit the copious ILP that remains hidden from current microarchitectures.

In this paper we focus on one of the most critical components of a large-window processor: the Load/Store Queue (LSQ). There have been many previous design proposals for the LSQ. However, in architectures that can handle thousands of in-flight instructions, most techniques fail to deliver performance. In Section 2 we analyze the causes for this failure.

To overcome the bottlenecks we apply the concept of *Execution Locality* to our LSQ design. Execution Locality is the observation that instructions tend to execute in bursts separated by cache-missing loads. It has been applied to the design of large-window processors [6, 7]. However, the authors have not proposed a workable LSQ for this architecture.

Based on Execution Locality we design an LSQ with two-level disambiguation, dividing the non-completed instructions into two parts depending on whether they are miss-dependent or not. The instruction that divides both groups is the oldest instruction that does not depend on a cache miss. This instruction and all younger ones belong to the *High-Locality* part of the window. Older instructions belong to the *Low-Locality* part of the window. Low locality instructions are further partitioned into epochs implemented in different banks. A two-level disambiguation scheme is implemented based on these epochs. On issue, loads and stores first search the local epoch for matches, then the global level. The implementation makes local searches much more power efficient than global searches and profits from store-load locality.

This paper makes the following contributions:

- A Load/Store Queue based on *Execution Locality*, the ELSQ, is proposed. (Section 3.2)
- Exploiting locality, several restricted disambiguation schemes are proposed that can considerably reduce the implementation complexity. (Section 3.3)
- The LSQ further classifies low-locality memory instructions into *epochs* based on their age. Epochs are the building blocks for the proposed two-level disambiguation scheme. (Section 3.4)
- Several filtering schemes are proposed to reduce activity in global disambiguation. (Section 3.4)
- The energy-efficiency is analyzed. (Section 6)

2 Background

2.1 Memory Handling for Large Windows

Building a scalable load store queue (LSQ) is challenging. LSQs are more difficult to implement compared with normal instruction queues due to their higher number of states and functionalities.

In a normal instruction queue there are only two states: *Ready* or *Not Ready*. The functionality of the LSQ is more complex. Issuing loads need to search the Store Queue (SQ), while stores need to search the Load Queue (LQ). The overall functionality that needs to be supported in an uniprocessor environment is as follows:

Store-Load Forwarding: When a load issues, in addition to accessing the data cache, it also needs to search the store queue for older stores matching the load address. If there is a match, the load should use the data from the store queue instead of the cache. Store-Load Forwarding involves two special cases. First, the matching store might still be waiting for data. In this case it is common to periodically reissue the load every few cycles until the data is available. Second, the load access might only partially match the store. In this case, special action should be taken to recover the correct value. Some implementations squash the load and do not issue it again until the store has committed its data to the cache [8].

Store-Load Ordering: When a store issues, it is necessary to check whether a younger load with a matching address has already executed, potentially violating program semantics. In general, store-load violations squash the instruction window starting from the violating load. Fortunately, these violations are rare.

Commit: At commit, stores update the memory in program order, maintaining program semantics. All loads and stores need to be buffered during their whole lifetime.

Due to the age-based operation of LSQs it is typical to implement age-indexed LSQs. In this scheme, when a memory instruction is decoded, an entry is allocated at the tail of the LQ or SQ. The size of the LSQ needs to be balanced so that it does not overly constrain the instruction window. We now introduce two relevant solutions that have been proposed for the LSQ.

Hierarchical Store Queues One solution that has been proposed to overcome the problem of the Store Queue is to use hierarchical store queues (HSQ) [3]. In this scheme, the SQ has two parts: A small and fast first-level store queue stores the X youngest stores in the window and a large and slower second-level SQ stores all older stores. This scheme optimizes loads that are ready soon after decode and forward from close stores, but penalizes loads that depend on chasing pointers or forward from distant stores. The hierarchical store queue also suffers higher complexity due to the way it manages checkpoints. Second-level stores are tracked by hashing into a set of counters. Checkpoint recovery consists of decrementing the counters, one by one, for every squashed store. This is costly and takes extra time.

Load Re-Execution The largest group of techniques addressing the load queue are those related to load re-execution [9, 10] with the goal of making the LQ non-associative, thus solving the LQ scalability problem. In these schemes, when a store issues, it does not search the LQ for violations. Instead, when the load commits, the load re-executes and checks whether it obtained the correct value. Research has concentrated on reducing the number of loads that need to re-execute. Lipasti et al. [9] propose re-executing only loads that issue while there are older stores with unknown addresses in–flight. Store Vulnerability Windows (SVW) [10] is another way to decrease re-executions. SVW uses a Bloom filter to determine whether a re-execution might be necessary, substituting data access with filter access and possible re-execution.

Large instruction windows can increase the number of necessary re-executions, making this technique less applicable. For example, using SVW with a 10-bit SSBF, a conventional out–of–order processor with a 64-entry window observes an average of 1 re-execution every 715 instructions for SPEC FP. The same execution using a large window processor with about 1500 in-flight instruction results in 1 re-execution every 95 instructions.

2.2 Execution Locality

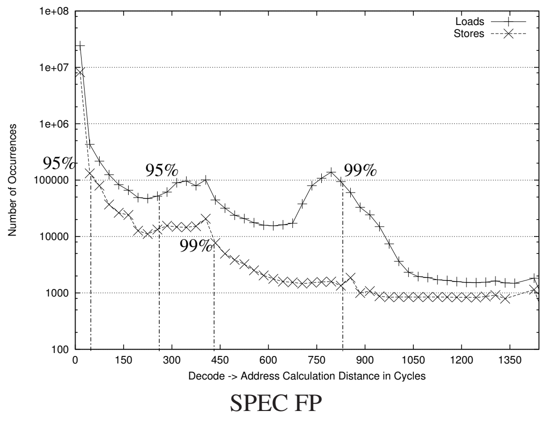
A large impediment for high performance in current microprocessors is the memory wall, i.e. the discrepancy between memory access times and processor clock cycle. Given the current processor/memory speeds and ROB sizes, every time a main memory access occurs the processor spends around 90% of the service time idle because the ROB fills.

The impact of the memory wall on performance has been studied comprehensively [11]. Applications with large working sets and little locality suffer a large penalty in terms of execution speed. For applications with high memory-level parallelism, much of the performance can be recovered by increasing the size of the instruction window. Since parallel memory accesses do not need to wait for previous accesses to finish, memory access latencies can be hidden.

Technologically it is infeasible to design large window processors using traditional scaling techniques. Register files, issue queues (wake-up and select machinery) and LSQs are all hard-to-scale structures. Attempting to scale them reveals power density and access time constraints. Thus, a different approach needs to be explored.

Almost all large–window processor designs that have been proposed rely on the observation that the instruction window does not need to grow unless a long latency event such as an L2 cache miss occurs. Most proposals operate by detecting the presence of cache misses and extracting dependent instructions. Two techniques have been proposed. One group of proposals extracts instructions at issue, using the select and wake-up logic [1, 2]. Another set of large-window designs makes use of the concept known as *Virtual ROB*, in which long-latency instructions are extracted when they reach the head of a partial ROB [5, 6].

In an environment where main memory accesses have high latencies relative to the processor frequency, dependent instruction chains execute quickly relative to memory accesses. Unless the working set is small and fits into the cache, the execution of such applications consists of small bursts of in-



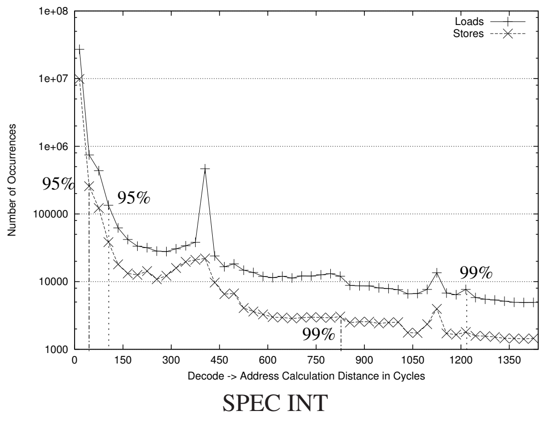


Figure 1. Floating Point (left) and Integer (right) Decode→Issue Distribution for 100 million committed instructions

structions followed by intervals in which the processor waits for data. This is the observation behind *Execution Locality* [6].

This concept has been used to propose decoupled processor designs. A first core executes these *high locality* instructions just after decode. Analysis shows that most instructions belong to this category. This first core is called the *Cache Processor* while the cache-miss dependent instructions are processed by a second core, the *Memory Processor* [6]. Cache-miss dependent instructions are said to have *low locality*.

Figure 1 shows how the concept of execution locality applies to address calculation. The plots classify loads and stores depending on the latency between instruction decode and address calculation. Each data point represents the number of loads or stores that have a similar decode→issue latency measured in cycles. Similarity is grouped in blocks of 30 cycles. The test ran SPEC CPU 2000 on a 4-way out-of-order processor with a large window (up to 4096 in–flight instructions) and a memory subsystem with L1, L2 and main memory with distances of 1, 10 and 400 cycles, respectively. The numbers are averages for 100 million committed instructions over all benchmarks. The plots show the latencies within which 95% and 99% of all loads/stores are covered. For SPEC2000, around 91% of all loads and 93% of all stores calculate their addresses within 30 cycles after decode. The figures show that most address calculations do not depend on cache misses, explaining the prefetching effect achieved by large-window processors. For address calculations that depend on cache misses, loads are more frequent than stores. Few stores have address calculations depending on a cache miss, and almost none depending on multiple cache misses.

3 Epoch-based Load Store Queue

3.1 Generic Processor Model

We first explain the LSQ model in the context of a traditional superscalar with a microarchitecture resembling that of a MIPS R10000 [12]. This processor features a reorder buffer and a centralized physical register file. Logical registers are renamed during decode and checkpoints are taken at branches. In Section 4 we will show how ELSQ can be integrated into a microarchitecture based on execution locality.

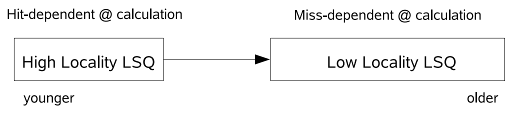


Figure 2. Basic Scheme of a two level LSQ based on Execution Locality

3.2 Epoch-based Load/Store Queue

We propose to partition the LSQ based on *Execution Locality* [6]. For high-locality memory references we keep a first high-locality LSQ, while low-locality references occur in the low-locality LSQ. Low-locality address calculations are more latency tolerant. However, store-load forwardings from low-locality stores to high-locality loads are critical.

The partitioning that we propose enables fast access time and reduces power for high-locality memory instructions. In any given cycle, the number of these instructions is relatively small and moderate sized queues are sufficient to track them. Thus, the technique resembles schemes that partition the queue by using address interleaved LSQ banks. However, the conceptual differences imply completely different logic designs. Address-interleaved LSQs require mechanisms to test the ordering between memory instructions that reside in different banks. In our model, memory instructions are physically ordered among the queues so that low-locality instructions are older than high-locality instructions. This idea is illustrated in Figure 2.

Loads and Stores are sequentially moved from the high-locality queue (HL-LSQ) to the low-locality queue (LL-LSQ) either when it is known that the address calculation is cache miss-dependent or whenever the low-locality queues are active. This is implicitly represented by the arrow in Figure 2. When the LL-LSQ is not active it can be kept in a low-power mode. This is beneficial to our design since the processor runs in *high-locality mode* for a large amount of time.

Consider the code segment annotated with cache behavior shown on the left of Figure 3. The right side of Figure 3 shows how execution proceeds. As long as address calculations do not depend on cache misses, address computation and issue

```
1) MOVI R1, X   ; R1 <- X

2) LD R2, 0(R1) ; R2 <- Mem[R1]
                  (Cache Hit)

3) LD R3, 0(R2) ; R3 <- Mem[R2]
                  (Cache Miss)

4) LD R4, 0(R3) ; R4 <- Mem[R3]
                  (Cache Miss)

5) ST R4, 0(R1) ; Mem[R1] <- R4
```

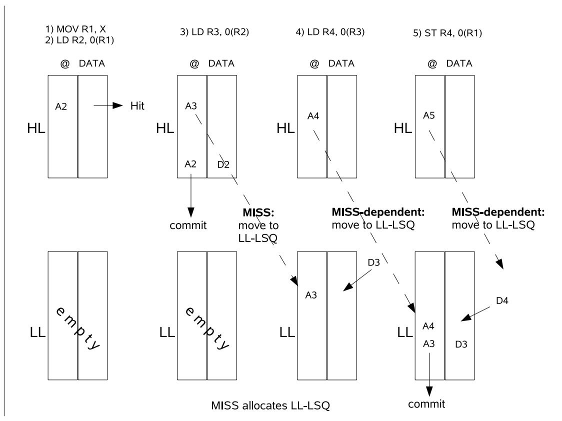


Figure 3. Execution (right) of example code (left) in a locality based Load/Store Queue

proceed in the first queue (HL-LSQ). If the address calculation does depend on a cache miss, then the instruction migrates to the second queue (LL-LSQ) before address calculation and issue proceed. Loads that obtain their address in the HL-LSQ but miss in the cache are also migrated to the LL-LSQ. Migration from HL-LSQ to LL-LSQ follows a scheme based on the Virtual ROB [5]. The goal of these techniques is to maintain instructions separated in two queues based on age and locality.

3.3 Restricted Disambiguation Models

The scheme so far presented allows loads and stores to disambiguate either in the HL-LSQ or in the LL-LSQ. However, disambiguation also needs to occur between locality levels. As we will see, some support logic is needed to make this work. This logic can be simplified if we restrict the disambiguation capabilities. We consider four disambiguation models:

- **Full Disambiguation**: In this model, loads and stores are allowed to disambiguate in both the HL-LSQ and the LL-LSQ. This model requires associative queues in both locality levels for loads and stores.

- **Restricted SAC**: Store Address Calculation (SAC) is restricted mainly to the HL-LSQ. If a store address depends on a long-latency register the store is allowed to migrate, but no later memory reference can be migrated until the store address calculation completes. This model simplifies disambiguation by removing LL-LSQ searches for store-load violations. The model benefits from the fact that store addresses are usually calculated in the HL-LSQ and rarely occur in the LL-LSQ (see Figure 1). Thus, stalls will be infrequent.

- **Restricted LAC**: In this model, Load Address Calculation (LAC) is restricted to the HL-LSQ. The benefits in terms of logic are less than those of the restricted SAC model as store-load forwardings will still require searching for stores in both HL-LSQ and LL-LSQ. Moreover, loads tend to have many more long-latency address calculations than stores so performance is likely to degrade more noticeably.

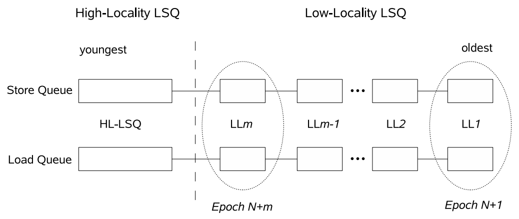


Figure 4. LSQ with banked LL-LSQ

- **Restricted LAC/SAC**: In this model both loads and stores are restricted. Disambiguation resources are considerably simplified. However, the store window may be large so a solution for the LL-LSQ is still necessary.

For common parameters like a 10-cycle L2 cache and a 4-way processor, the HL-LSQs need not be larger than 24-32 entries, so a conventional-sized queue is enough.

3.4 Hardware Disambiguation Schemes

Since the LL-LSQ holds all low-locality loads and stores, it may need to buffer hundreds of memory references. We address the problem of the large LL-LSQ by banking the structures. To keep the sequence of memory instructions we bank based on age, not address. The number of banks is a design parameter. For the implementation we want to have as few banks as possible to minimize complexity, but enough banks for each to have smaller size. Figure 4 shows the partitioning.

Despite the multiple structures, this scheme still represents a sequential window of memory instructions. Since each partition of the LL-LSQ contains a sequential portion of loads and stores of the instruction window –which we call a *memory epoch*– we call our LSQ scheme the *Epoch-based Load Store Queue* or ELSQ, in short. Note that instructions never travel between epochs. We will use the banked scheme as the basis and on top of it implement a scalable disambiguation scheme for ELSQ.

Implementing an eager disambiguation scheme consists of implementing store-load forwarding at load issue and store-load violation detection at store issue. We now describe how this task is accomplished in the Epoch-based LSQ. The ELSQ uses a two-level disambiguation. The first level is *Local Disambiguation*. This disambiguation occurs within the epoch and involves no global searches. Loads search the local epoch's store queue for matches while stores search the local epoch's load queue for violations. If a load finds a matching store, the procedure stops as there is no need to perform a global search. In this case the power of the search is reduced to a single epoch. The scheme benefits from the fact that the majority of store-load forwardings happen among close store-load pairs. A similar procedure is applied to stores when they find a local violation.

If the local search does not hit a global search is conducted. *Global Disambiguation* provides the overall integration necessary for correctness. Its goals are the same as local disambiguation, i.e. having loads get the correct value from matching stores and having stores check loads for violations.

We propose two filtering schemes to avoid unnecessary searches. The goal of these schemes is to minimize the number of searches with a minimal hardware budget. One constraint that the filters need to satisfy is that the access time must be no longer than the time it takes to search the local store queue or the L1 cache access latency. If the filter cannot satisfy this condition load execution time will grow with a noticeable performance penalty. For the ELSQ we study two filters: one based on L1 cache lines (*Line Filter*) and one based on Bloom filters (*Hash Filter*).

Line-based Filter In the first filter two bit–vectors (one for loads and one for stores) with as many entries as the total number of epochs are associated with every L1 cache line. The full collection of bit–vectors forms a table that we call the *Epoch Resolution Table* (ERT). There are two cases in which the ERT is updated: First, when a memory instruction with a known address is inserted into an LL-LSQ epoch, it sets the bit corresponding to its epoch and cache line in the ERT; and second, when an instruction obtains its address while in the LL-LSQ.

Global disambiguation proceeds as follows. In parallel with *local* store queue search during load issue, the cache line and the ERT store bit–vector are accessed. The value from the cache is used only when there is no active bit in the ERT store bit–vector. If an active bit is found it means that there is a possible match with a store and a remote search is conducted. A load pays an additional latency penalty while waiting for the search, even if it does not result in a match. The information from the bit-vector contains the epoch to which the likely matching store belongs. Using this information the load accesses the LL-SQs searching for the store. It searches the epochs that were active in the bit vector, one at a time, starting from the most recent one. This considerably reduces the energy required for the searches.

For this scheme to work it is necessary that the address-known memory instructions in the low-locality queues have an associated bit in the cache ERT. Thus, the system requires that all referenced lines be allocated in the L1 cache. Note that the data need not be available. When a new address appears in the LL-LSQ it is necessary to allocate the line and lock it in the cache. Locking is necessary because a replacement would break the disambiguation mechanism. If the new line cannot be allocated because all the lines in the set are already locked then special action needs to be taken. If the address belongs to an instruction that is being inserted from the HL-LSQ, then the insertion procedure is simply stalled. However, when the address is due to a memory reference that issued in the LL-LSQ the situation is more complex. The problem is that the loads that are locking the set may be younger than the load that issued. Stalling would result in a deadlock. As a solution, when this happens we proceed to squash the instruction window starting from the load that tried to lock the line and restart execution. This is supported by the recovery logic.

Locking cache lines does not involve any additional structures as the replacement algorithm can take care of everything. It will only replace lines for which there are no active bits in the ERT.

Address Hashing based Filter To avoid the complexity resulting from modifying the algorithm to handle cache-line locking we also study a more conventional method based on Bloom filters [13]. In this method, the ERT is indexed using a hash consisting of a set of the lower–bits from the address. Thus, when a memory instruction is inserted into the LL-LQs or computes its low-locality address it takes n bits of the address, it indexes the ERT and activates the bit corresponding to its epoch. This scheme is decoupled from the L1 cache. The access time to the ERT will depend on its size, so this is a parameter to take into account.

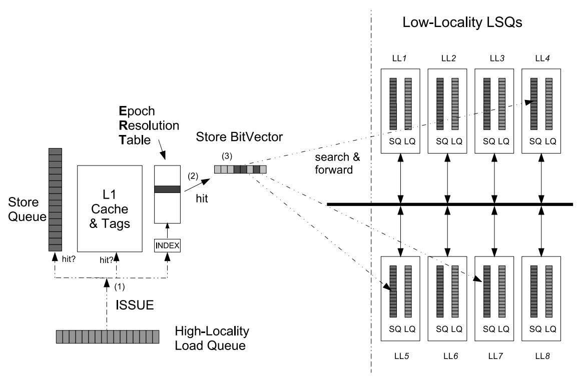
We have described the two global disambiguation schemes using the *Full Disambiguation* configuration. Using restricted schemes may simplify the hardware considerably. Restricted SAC, for instance, eliminates the need for the *Loads–ERT*. The *Stores–ERT*, however, is always necessary for operation.

Figure 5 shows the operation of store-load forwarding for both high-locality and low-locality loads forwarding from a low-locality store. The access to the ERT is guarded by a structure with the label INDEX. This structure reads the address and decides where in the ERT to index, depending on the filtering mechanism. Note that the figure only shows the store-search part of the forwarding process. The data needs to be sent back to the load, following the same path backwards.

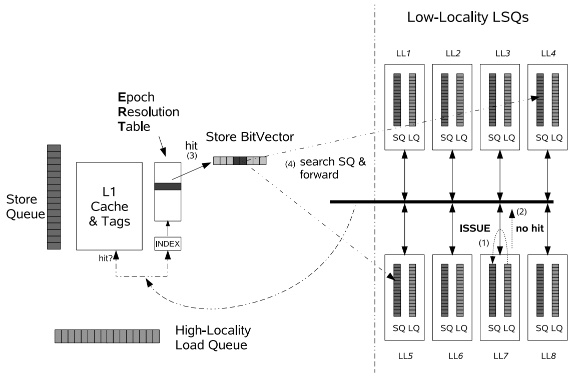
In both filtering schemes, when an epoch commits, the stores are sent to memory and the two columns that represent this epoch in the ERT are cleared. This way, lines in the line-based ERT get automatically unlocked by the bit handling mechanism. This method is notably simpler than the HSQ [3] method that requires counters to be decremented one-by-one for every store in the checkpoint.

3.5 Non-Associative Load Queue with Load Re-Execution

The methods we have introduced work well as a way to reduce search activity in the Load and Store Queues. A different approach is to attempt complete removal of parts of the LSQ. As we have mentioned before, much research has concentrated on removing the Load Queue and maintaining program semantics via *load re-execution* [9, 10]. Load Re-execution consists of executing an optimized load again during the commit stage to check for the validity of the optimization.



(a) High Locality Load



(b) Low Locality Load

Figure 5. Operation of store-load forwarding for both High-Locality and Low-Locality Loads hitting in the ERT

Only loads that may have incurred a store-load ordering violation should re-execute. It is important to take this into account, because cache access bandwidth is limited and expensive. Many techniques to reduce the number of re-executing loads have been researched. For instance, Roth proposed tracking whether the load is vulnerable to any recently committed store [10]. An alternative way to reduce the re-execution rate is to track whether there are stores in flight with unknown addresses and, in the case of store-load forwarding, see if they are younger than the store that is forwarding. If this is not the case then there is no need to re-execute. This is called the *no–unresolved–store–filter* [9].

These techniques can be added to ELSQ to make the Load Queue non-associative. However, care needs to be taken with the *no–unresolved–store–filter*. The filtering that guards searches in the LL-SQ does not track stores with unknown addresses. As a solution, it is possible to track which epochs contain address-unknown stores by adding a new ERT table, and adding counters in the epochs to track unresolved stores. The additional ERT table would need to be accessed by all executing loads (except locally forwarded loads). This is a trade-off that needs to be analyzed in relation with performance (see Section 5.6).

3.6 Coherence and Consistency

The epoch-based LSQ is designed with traditional memory semantics in mind. Externally the system sees a huge load-store queue. The cache subsystem features only one L1 data cache and there is only one L2 cache between L1 and main memory. ELSQ does not modify the problem of maintaining caches coherent, which can be solved using either *snooping* or *directory-based* schemes.

ELSQ is designed to support total store ordering. Most current processors operate under this model or use weaker models. Thus, ELSQ can be implemented on most architectures.

4 Integration with Locality-based Processor

The processor model that has been used so far is based on conventional technology. It is well understood and can conceptually work together with ELSQ, but in a real implementation it is not valid since it cannot scale to our goal of handling thousands of in-flight instructions.

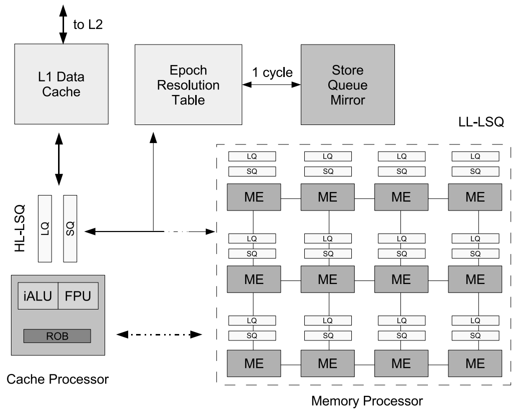


Figure 6. Integration of ELSQ on top of FMC

We have integrated ELSQ on top of a novel microarchitecture known as FMC (Flexible MultiCore) [7]. As with previous execution locality designs, it consists of a Cache Processor (CP) that processes high locality instructions and a Memory Processor (MP) that processes remaining low-locality instructions. In FMC, the MP is partitioned into small sequential engines (called *memory engines, ME*) with the goal of providing reconfigurable heterogeneity. The natural way to complete the integration is to establish a one-to-one relationship between the memory engines and the ELSQ concept of epochs.

The overall organization and interconnect of this architecture is shown in Figure 6. FMC uses a mesh network to interconnect the different memory engines. By mapping ELSQ, every epoch is mapped to a memory engine. As a result the LL-LSQ is distributed along the memory engines. Access to the memory engine network is provided by a bus that interconnects the CP and the MP. In our model, every access from the CP to the MP or back results in a one-way penalty of 4 cycles. Traveling among memory engines works at the speed of one hop per cycle.

For the ELSQ, it is critical that store-load forwarding is as fast as possible. Often high-locality loads forward from low-

Parameter	Value	Parameter	Value
Fetch/Decode BW	4 insts per cycle	OoO-64 Integer IQ Entries	40
CP ReOrder Buffer Size	64	OoO-64 FP IQ Entries	40
ME Max Instructions	128	OoO-64 Scheduling Policy	Out-of-Order
ME Max Loads	64	OoO-64 INT/FP Registers	96/96
ME Max Stores	32	Number of Cache Ports	2 read/write ports
CP Integer Issue Queue Entries	40	L1 Cache Configuration	32KB 4-way, 32-byte lines
CP FP Issue Queue Entries	40	L1 Cache Latency	1 cycle
CP Scheduling Policy	Out-of-Order	L2 Cache Configuration	2MB 4-way Assoc.
CP INT/FP Registers	96/96	L2 Cache Latency	10 cycles
ME Issue Queue Entries	20	Main Memory Access Time	400 cycles
ME Scheduling Policy	In-Order MultiScan [7]		
ME Issue Width	2-way		

Table 1. Default Processor Parameters

locality stores. If this operation performs a round-trip every time (> 8 cycles) the penalty may be noticeable. To alleviate this problem we suggest a final addition to the ELSQ: implementing a *Store Queue Mirror* (SQM). The SQM is a replica of the LL-SQs located next to the ERT. It is updated when a store address appears in the Memory Processor. Accessing the SQM in the Cache processor costs only one additional cycle after ERT access. Figure 6 shows the location and interconnect of the SQM. When implemented, the SQM also acts as the buffer for stores before they commit. Thus, the SQM does not incur any additional power due to network trips.

4.1 Exceptions and Recovery

Maintaining correct state when exceptions occur is another important issue in the design of ELSQ. Being able to recover at any point of the LSQ is a complex issue even for smaller designs. In ELSQ we simplify this issue by relying on checkpoints. ELSQ considers checkpointing only for the low-locality LSQ, which holds many more instructions. The Cache Processor checkpoints branches so recovery may proceed like in a MIPS R10000 processor [12]. For LL-LSQ recovery, a checkpoint is associated with every epoch. When an exception occurs, the processor restarts execution starting from the instruction that initiates the epoch. To keep the state of the ELSQ consistent with program semantics, all loads and stores belonging to this epoch and to younger epochs –including the HL-LSQ– are squashed. This means that some correct path instructions get squashed. Nevertheless, low-locality recoveries are much less frequent than high-locality ones. Using checkpointing for the ELSQ is similar to the use of checkpointing for large window processors such as [3, 2, 6, 7].

5 Evaluation

We now evaluate the performance of the ELSQ. First we analyze global performance issues, establishing the epoch size and comparing the overall performance (Sections 5.2 and 5.3). We then proceed by focusing on the store queue, analyzing the performance of the filtering schemes (Section 5.4). Finally, we analyze the load queue and evaluate the restricted disambiguation and re-execution schemes (Sections 5.5 and 5.6).

5.1 Simulation environment

The ELSQ is modeled on top of an execution-driven simulator that models the FMC microarchitecture. Conventional speculative out-of-order processors are simulated by disabling the Memory Processor part of the simulator. For ELSQ, both Line-based and Hash-based global disambiguation may be simulated. An unlimited conventional LSQ is also modeled. We also implement a model of load re-execution using Store Vulnerability Windows [10] and the *no–unresolved–store–filter* [9]. Table 1 shows the default values that apply for the different microarchitectures unless explicitly stated.

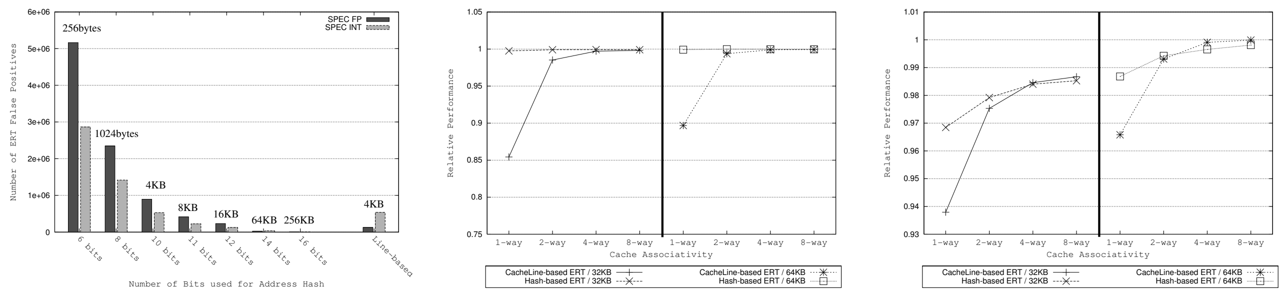
The simulator runs code compiled for the Alpha architecture. We used the Alpha binaries for SPEC CPU 2000 available from the simplescalar web page. The simulation methodology is as follows: For each of the 26 benchmarks a simulation point of 100 million instructions is obtained. The simulator executes these points and yields statistics which are then averaged with arithmetic mean.

5.2 Tuning Epoch Size

First we establish the sizes of checkpoints and epochs. The number of epochs is the same as the number of checkpoints. We choose 16 epochs since this value has been shown to work well for the FMC architecture [7]. We set the maximum number of low-locality instructions per epoch to be 128. This number includes integer ops, floating point ops, control ops and address calculations. Using this size for the checkpoint and a total of 16 epochs, we find that the maximal IPC for SPEC FP that can be reached lies at 2.99. We will now size the LSQ trying to stay within 1% of the unlimited LSQ performance. Setting the Load and Store Queue to 64 and 32 entries yields a average slow-down of 0.9% (7% worst-case). This seems a good trade-off from a power perspective. For this sizing procedure SPEC FP was used since at large window sizes it is more sensitive to variations than SPEC INT.

5.3 Performance of Epoch-based LSQ

We now evaluate the performance of the large window scheme. We evaluate both the cache-line-based and the hash-based ERT schemes. For a fair comparison we model the size of the hashing-based ERT to be the same as that of the cache line–based ERT. For the 32KB 4-way L1 that we use the size of the ERT amounts to 4KB of storage (2KB for the Load–ERT and 2KB for the Store–ERT). This translates to a 10-bit address hashing. For each ERT scheme we evaluate the impact of implementing the Store Queue Mirror. Finally, we also model a single-cycle unlimited–size centralized Load Store Queue. This LSQ is located in the Cache Processor to



(a) Filter Accuracy (b) SPEC FP Performance (c) SPEC INT Performance

Figure 8. Performance of the filtering schemes: (a) varying ERT size, (b-c) varying L1 cache

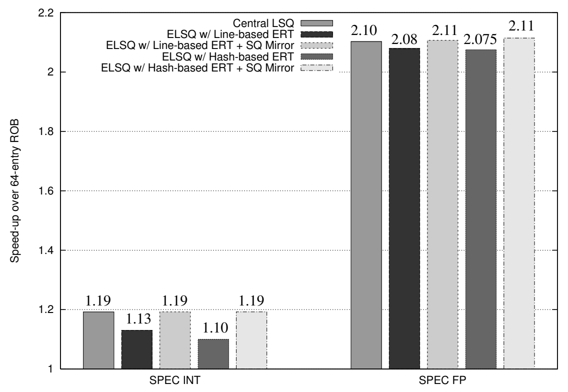


Figure 7. Speed-up of large window LSQ schemes over conventional 64-entry ROB

minimize store-load forwarding occurring in the high-locality stream. However, loads that execute in the Memory Processor suffer the corresponding round-trip penalty. The results are shown in Figure 7 as speed-ups over a conventional processor with a 64-entry ROB that yields IPCs of 1.55 and 1.42 for SPEC INT and SPEC FP, respectively. Note that this processor size is not representative of current technologies. It is chosen because it features the same parameters as the Cache Processor in the FMC architecture and emphasizes the impact of large windows.

The figure shows that for SPEC FP the performance is quite good for all schemes. The presence of the SQM improves performance by about 1% and provides a performance that is slightly larger than that of the centralized queue. This small performance gain comes from local forwardings in the LL-LSQ that require a full round-trip in the case of the ideal centralized queue. SPEC INT, on the other hand, is much more sensitive to the store-load forwardings from low-locality stores to high-locality loads. The presence of the SQM has thus a big impact on the performance, providing up to 8% more performance. Once the SQM is implemented, ELSQ performs at the same speed as the idealized queue.

5.4 Performance of Global Disambiguation

In this section we evaluate the performance of the filtering mechanisms for global memory disambiguation. The efficiency of the mechanism has in principle little impact on IPC. Different filtering schemes affect the number of searches that happen in the LL-LSQ, either for store-load forwarding or ordering violations (in the latter case it affects also the number of searches in the HL-LSQ). The schemes will have an impact on area, complexity and power, but the impact on performance is small. The Line–based scheme could be a little bit of an exception here, since it requires to lock cache-lines. However, for the 4-way 32KB L1 cache that we use, the performance penalty still lies only around 0.4% and can be safely ignored.

The main effect that needs to be evaluated is the number of false positives that are generated. A false positive happens when the ERT directs the load or store to search in an epoch where a matching address is actually not present. Such a search is useless and wastes power. Thus, goal of the ERT scheme should be to minimize these searches. For the address-hash based scheme this goal can be achieved by incrementing the number of bits. Doing so, however, increments the hardware budget of the scheme. Choosing the best scheme involves a trade-off. Figure 8 (a) shows the average number of false positives for 100 million committed instructions in SPEC FP and SPEC INT together with the estimated hardware budget. The hardware budget is estimated by taking into account that we need two ERT tables (one for loads and one for stores) and that every entry stores 16 bits. The figure shows that ERTs of at least 4KB are necessary to have less than 1 false search every 100 instructions. Note that 4KB here means 2KB for the Load-ERT and 2KB for the Store-ERT. The figure also shows that, using 32-byte lines, the line-based scheme requires about half the hardware budget for similar accuracy. Finally, it also shows that the filtering performance depends a lot on how well the filter maps to the application behavior. The line-based scheme performs much better on SPEC FP while the hashing scheme seems to have better performance on integer applications.

We also evaluate the impact of modifying the L1 cache on the Line-based ERT scheme. This scheme depends on the configuration of the cache since the ERT requires long-latency address calculations to have the corresponding cache lines *locked* in the cache. Intuitively this means that high-associativity caches may be necessary since line conflicts are handled via processor stalls or squashes. This section will evaluate how large the L1 cache need to be to minimize the losses. We eval-

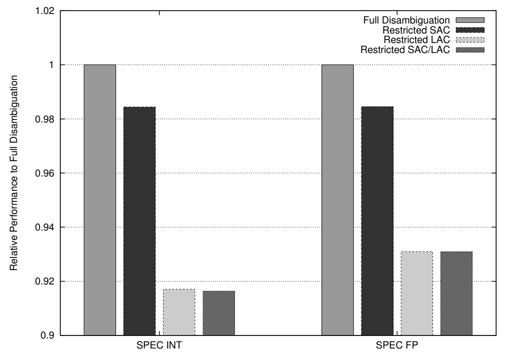


Figure 9. Relative performance of restricted disambiguation models

uate a series of cache configurations of 32KB and 64KB, with associativity ranging from 1 to 8 ways. To compare we also add a hash-based ERT architecture. The interleaving is set so that hardware budgets for ERT are the same in both schemes. Thus, for the 32KB cache 10 bits are used and for the 64KB cache 11 bits are used. Figure 8 shows the results for this test relative to the highest scored performance. The figure shows that an associativity of 4 recovers the lost performance for both SPEC INT and SPEC FP. It also shows that the L1 cache size and associativity has a much higher effect on performance for SPEC INT than SPEC FP.

5.5 Restricted Disambiguation Models

We now analyze the performance of the restricted disambiguation schemes introduced in Section 3.3. Figure 9 shows the performance of the four disambiguation models. Full disambiguation has been chosen as the baseline against which comparison is being made. The figure shows that restricted LAC involves a higher penalty than restricted SAC. The reason for this has to be found, as shown in Figure 1, in the fact that many more loads with low locality address calculation exist than stores. Finally, when both stores and loads are restricted, performance is similar to just using restricted LAC. This is a result of low locality loads being much more frequent than stores. Thus the stalls arising from restricting their address calculation have a much higher penalty.

The performance of all four schemes is quite good. In particular, restricted SAC yields a slowdown that is below 2% for both SPEC FP and SPEC INT. Looking into the benchmarks further reveals that the slowdown is due to peculiarities of particular applications. For example, in the SPEC FP case, all the slowdown is attributable to `equake`, which suffers around a 30% performance loss. Much of the execution of the simulation point is covered by the `smvp()` function, which involves heavy multilevel pointer dereferencing, for both loads and stores.

Restricted SAC has a notable implementation advantage: it eliminates the need for a large associative Load Queue. Since stores may only compute their address in the Cache Processor, only loads in the small high-locality load queue (HL-LQ) may incur ordering violations. As a result, global disambiguation for load violation is no longer necessary, which eliminates the need for the Load-ERT. Note that, unfortunately, the converse does not hold. Restricted LAC does not allow to remove the store queue, so the global disambiguation scheme devised previously would still need to be implemented. Overall this means that restricted LAC is probably not a good idea.

5.6 Large Window Load Re-Execution

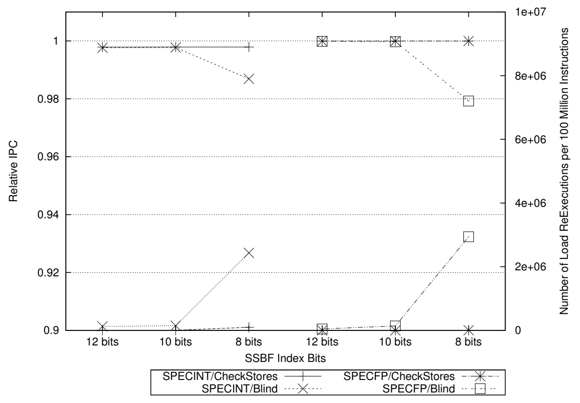
Finally, we analyze how Re-Execution performs in this context. We implement the technique of Store Vulnerability Windows [10] and remove the Load Queue. We do not modify any other structure. The implementation of SVW comes in two variants:

CheckStores: In this variant, when a load issues and forwards from the store queue, it checks if any stores with unknown address exist between the store-load pair. If so, the load re-executes during commit. This is the *no–unresolved–store–filter* [9]. As will be seen, doing so improves performance, but it adds complexity to the store-load forwarding machinery. Complexity is increased because it is necessary to implement a mechanism that tracks unresolved stores.
Blind: In this variant, we do not check whether stores with unknown addresses exist. Instead, we use only the SVW filtering mechanism to decide whether a load needs to be re-executed or not.

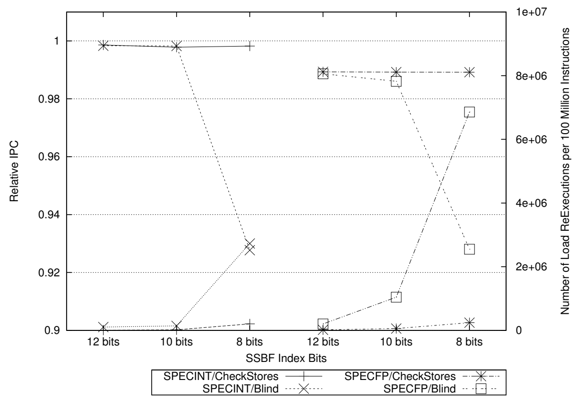
We evaluate the performance of the SVW scheme for both IPC and increase in cache activity. In the evaluation we take into account the fact that re-executing loads forces subsequent stores to commit after the cache access completes. In most cases this will be the next cycle (L1 access latency), but some loads re-execute from the L2 and, in some very rare cases, the load may re-execute from main memory. This behavior can affect performance when the re-execution rate is high.

Evaluating SVW in the context of large-window architectures is especially interesting as large windows are much more likely to create ordering violations compared to small windows. Figure 10 compares the performance of SVW on the FMC – emulating a window of around 1500 instructions – (right) and a smaller 64-entry ROB processor with a conventional out–of–order architecture (left). The small processor is provided to show how the number of re-executions increases. The figure shows three configurations for the Store Sequence Bloom Filter (SSBF), ranging from 8 to 12 bits.

From the results we see that using 12 bits has very good performance in all schemes. The resulting table might, however, be a little larger and more power-hungry than desired. Using 10 bits for the SSBF is still a good option. Performance is almost unaffected except for SPEC FP when the *no–unresolved–store–filter* is not used. The additional re-executions will increase the energy dissipated by the cache, but it needs to be taken into account that the CheckStores mechanism implements additional structures that are accessed by all loads issue while the processor is in *low-locality mode*. This will add to the power consumption. Finally, even a SSBF with 8 bits works nicely if the filter is implemented. Otherwise, the performance for SPEC FP starts to degrade considerably (∼7% vs. 1%).



(a) 64-entry ROB OoO Processor



(b) FMC Processor Model ($\sim$1500 in-flight instructions)

Figure 10. Performance of Store Vulnerability Windows on SPEC CPU 2000 relative to a model featuring Load Queue. The plot shows how Re-Execution is a very window-size dependent technique

6 Energy Considerations

Our goal in introducing the ELSQ is to provide a large, high-performance LSQ that operates with little additional power and low complexity. We now analyze the power characteristics of the ELSQ. Increasing the size of a standard load-store queue would increase the energy consumption excessively, precluding its implementation. However, this is not true for the ELSQ. Although ELSQ keeps many queues, most of them are not active at any given moment and do not perform searches. In general, when the ERT returns a positive match, only one low-locality queue is searched. This happens because of the highly accurate filtering methods as was shown in Figure 8.

First, let us evaluate how much time the processor spends in *high-locality mode*. During this mode the Memory Processor does not need to track instructions because no cache misses have occurred recently (i.e. the Memory Processor is empty). Since the processor does not use low-locality resources, the LL-LSQ together with the ERT and the associated logic can be kept in a low power mode. Figure 11 shows the percentage of time in which the processor is in the high-locality mode as a function of the L2 data cache size.

The figure shows that even with a small 1MB cache, one third of the time the processor only uses the small HL-LSQ machinery that tracks only 32 loads and 24 stores. This is similar in size to what today's processors use. As the data cache grows to 8MB the percentage of time during which the LL-LSQ runs in minimal-power mode averages 50%. Note that, when the Memory Processor is active, not necessarily all epochs queues are allocated. For the 2MB L2 cache an average of 5.73 epochs are allocated for SPEC FP while for SPEC INT this number drops to 4.77. Furthermore, if a designer wants to increase the efficiency of the queues, the Memory Processor can be shared between threads [7].

To estimate the dynamic power requirements of the implementation we track the utilization of the structures including

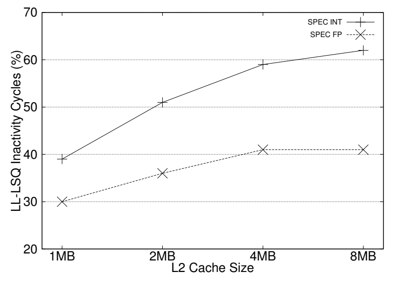


Figure 11. Percentage of time in which the FMC is in high-locality mode

accesses to the queues and the number of ERT lookups and network roundtrips of the searches (with or without data). Table 2 shows the average number of events for 100 million committed instructions for each of the SPEC FP and SPEC INT simulation points. Several large window and two small window configurations are evaluated. The structures of the 64-entry processor match the sizes of the cache processor allowing us to better understand the power behavior of the processor. The SVW implementation uses an SSBF with 10 bits and it does not implement the *no–unresolved–store–filter*.

There are several interesting observations to make from these tables. The structures that receive the most searches are the High-Locality Store Queue and the Epoch Resolution Table. The number of accessing instructions ranges from $25*10^6$ (FMC-Hash HL-SQ access in SPEC FP) to $37 * 10^6$ (FMC-Line HL-SQ access in SPEC INT). For the modeled 4-way fetch/decode architecture, the ERT and the HL-SQ will need to be dual ported, while the high locality load queues, which are accessed by around $8 * 10^6$–$13 * 10^6$ instructions, may be designed as single-ported associative structures. The low-locality load/store queues see fewer instructions, between 0 and $10 * 10^6$, distributed to 5-6 subqueues (about 0-2 $* 10^6$ each, on average). Thus a single port is also enough for these

	Configuration	HL-LQ	HL-SQ	LL-LQ	LL-SQ	ERT/SSBF	RoundTrips	Cache	Speed-Up
SPEC FP	OoO-64	8.692	27.006	0	0	0 / 0	0	33.375	1.0
	OoO-64-SVW	0	27.006	0	0	0 / 26.591	0	34.135	0.997
	FMC-Line	8.761	25.929	0.119	8.902	27.521 / 0	1.561	31.862	2.09
	FMC-Hash	8.618	25.531	0.123	9.893	27.281 / 0	1.701	31.662	2.10
	FMC-Hash-SVW	0	26.010	0	9.795	27.453 / 26.591	1.546	32.971	2.08
	FMC-Hash-RSAC	8.732	25.815	0	9.378	27.037 / 0	1.468	31.610	2.07
SPEC INT	OoO-64	11.326	32.387	0	0	0 / 0	0	37.328	1.0
	OoO-64-SVW	0	32.387	0	0	0 / 29.769	0	38.081	0.998
	FMC-Line	13.356	37.703	0.115	10.348	34.327 / 0	0.544	39.961	1.196
	FMC-Hash	13.354	37.615	0.114	9.445	34.250 / 0	0.541	39.887	1.195
	FMC-Hash-SVW	0	37.602	0	9.606	34.130 / 29.769	0.438	39.948	1.190
	FMC-Hash-RSAC	12.867	36.294	0	8.056	32.624 / 0	0.354	39.291	1.176

Table 2. Number of access to LSQ components (in millions) for SPEC FP (top) and SPEC INT (bottom)

subqueues. Finally, note that network roundtrips can be implemented efficiently [14].

When control speculation works well, as in SPEC FP benchmarks, using ELSQ offers a good power–performance balance. For example, while OoO-64 performs 27 millions queue accesses to two-ported structures, FMC-Hash performs 25.5 million accesses to two-ported queues and 10 million accesses to single-ported queues (of similar size). This is an acceptable increase in power consumption. It is also necessary to account for the 27 million ERT accesses. The ERT is a 2KB SRAM with a similar access rate to a L1 cache, but its power consumption is much lower. Using CACTI-4.2 with a target technology of 70nm, the read energy for the ERT is 0.00195nJ while the read energy for the L1 cache amounts to 0.0958nJ. Thus, the read energy consumption of the ERT is only 2% that of the L1 Cache. With restricted disambiguation models additional gains can be achieved. RSAC reduces the number of accesses to the ERT, as stores do not access the ERT, and therefore it also reduces the number of round-trips. This is in addition to the benefit of removing the Load Queues from the Memory Processor.

Finally, unlike in SPEC FP, the number of LSQ access in SPEC INT grows with the aggressiveness of the processor. This is an effect of poor control path speculation. In integer programs, correctly speculating past multiple branches is difficult, so the processor instruction window grows, but many instructions are wrong–path. New control speculation mechanisms will be necessary to overcome this limitation.

Comparing the line-based filter and the address hash filter shows that both have similar behavior. FMC-Line reduces accesses to the LL-SQ and also reduces round-trips. On the other hand, stores in the MP need to lock the line. For SPEC FP, about 5.2 million stores access their cache line and lock it. Line locking and overflow squashes do not have a noticeable impact on performance. However, the higher implementation complexity makes this technique a less suitable candidate for implementation.

Finally FMC-Hash-SVW and FMC-Hash-RSAC are compared. This will tell us which of both methods is better for load queue simplification. Energy-wise, RSAC has some nicer properties than SVW. It reduces cache accesses (4% and 0%), round trips (5% and 19%), LL-SQ accesses (4% and 16%) and HL-SQ accesses (1% and 4%), for SPEC FP and SPEC INT, respectively. Other operations are not directly comparable: the access frequency of the SSBF (1024-entry RAM) is three times that of the HL-LQ (32-entry CAM). On the other hand SVW has marginally better performance than RSAC (0.5% and 1.2%). Without taking HL-LQ and SSBF accesses into account, we conclude the performance-power behavior is better for RSAC than SVW. Another topic that needs to be taken into account is implementation complexity. Implementing RSAC is simple: stores that do not have computed address at the head of the HL-LSQ stall migration. SVW, on the other hand, makes the whole Load Queue non-associative but requires the implementation of an additional table (SSBF) and some logic to implement the vulnerability windows.

7 Related Work

In addition to the schemes presented in section 2, several other important contributions have appeared in literature.

One proposal that shares similarities with our Epoch-based LSQ is the Address Resolution Buffer (ARB) [15], a work developed in the context of Multiscalar [16]. The main similarity is the use of local and global disambiguation levels, where the global level tracks groups of instructions and the lower level individual instructions. Despite this similarity, ELSQ controls global disambiguation via an Epoch Resolution Table, a concept inspired in directory-based cache coherence schemes [17, 18].

Several researchers have attempted to improve LSQ efficiency by introducing innovations into the traditional structure of the Load Store Queue. Sethumadhavan et al. [19] propose using hardware hashing to attack the issues of performance, power and latency. Single-bit hash tables are implemented via bloom filters with the goal of filtering unnecessary searches.

Park et. al [20] propose several optimizations to reduce search frequency on the LSQ. These include using a store-set predictor [21] to reduce the search requirements, implementing a load buffer for out-of-order loads that reduces the number of load queue searches and increasing the size of the LSQ by using segmentation.

Finally, Sethumadhavan et al. [22] propose a load-store queue architecture in which entries in the LSQ are not allocated at decode, but at issue. This technique reduces the size of the queues, but requires new methods to handle overflows.

Several works have developed high-performance LSQs on top of the basic technique of re-execution [9]. One group of optimizations is based on speculative memory bypassing

(SMB) [23]. The goal of this optimization is to reduce the pressure on the Store Queue. However, an aggressive configuration that predicts all store-load forwardings [24, 25] yields a non-associative store queue.

8 Conclusions

In this paper we have proposed a new load-store queue that aims at satisfying the memory requirements for a large-window processor relying on the total store order consistency model. The queue is based on two basic concepts: Execution Locality and epoch partitioning with two-level disambiguation. Epoch partitioning gives the queue its name: Epoch-based Load/Store Queue (ELSQ).

ELSQ has several important features. First, it allows to implement restricted disambiguation models that can considerably simplify the implementation. Second, a two–level disambiguation can be implemented efficiently. Using simple filtering schemes the activity of global disambiguation is reduced. Finally, local searches in the epochs exploit locality in store-load forwardings.

The queue has been simulated on top of FMC, a large window processor based on Execution Locality. In this environment, ELSQ has been shown to sustain high performance at only a slight cost in terms of power and complexity. Our evaluation also shows that techniques based on load re-execution are less competitive in terms of power, performance and complexity compared with restricted store address calculation.

Acknowledgements

This work has been supported by the Ministry of Science and Technology of Spain under contracts TIN–2004–07739–C02–01 and TIN–2007–60625 and by the Framework Programme 6 HiPEAC Network of Excellence (IST–004408). Daniel A. Jiménez is supported by NSF CAREER award CCF–0545898. Alex Veidenbaum has been supported in part by the NSF under grant CNS–0220069.

References

[1] A. R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg, "A large, fast instruction window for tolerating cache misses," in *ISCA-29*, 2002.

[2] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton, "Continual flow pipelines," in *ASPLOS-11*, 2004.

[3] H. Akkary, R. Rajwar, and S. T. Srinivasan, "Checkpoint processing and recovery: Towards scalable large instruction window processors," in *MICRO-36*, 2003.

[4] K. Sankaralingam, R. Nagarajan, R. McDonald, R. Desikan, S. Drolia, M. Govindan, P. Gratz, D. Gulati, H. Hanson, C. Kim, H. Liu, N. Ranganathan, S. Sethumadhavan, S. Sharif, P. Shivakumar, S. W. Keckler, and D. Burger, "Distributed Microarchitectural protocols in the TRIPS Prototype Processor," in *MICRO-39*, December 2006.

[5] A. Cristal, D. Ortega, J. Llosa, and M. Valero, "Out-of-order commit processors," in *HPCA-10*, 2004.

[6] M. Pericàs, A. Cristal, R. Gonzalez, D. A. Jimenez, and M. Valero, "A Decoupled KILO-Instruction Processor," in *HPCA-12*, February 2006.

[7] M. Pericàs, R. Gonzalez, F. J. Cazorla, A. Cristal, D. A. Jimenez, and M. Valero, "A Flexible Heterogeneous Multi-Core Architecture," in *PACT-16*, September 2007, pp. 13–24.

[8] J. Tendler, S. Dodson, S. Fields, and B. S. H. Le, "Power4 System Microarchitecture," *IBM Journal of Research and Development*, vol. 46, no. 1, 2002.

[9] H. W. Cain and M. H. Lipasti, "Memory ordering: A value-based approach," in *ISCA-31*, June 2004.

[10] A. Roth, "Store Vulnerability Window (SVW): Re-Execution Filtering for Enhanced Load Optimization," in *ISCA-32*, June 2005.

[11] T. Karkhanis and J. E. Smith, "A day in the life of a data cache miss," in *Proc. of the Workshop on Memory Performance Issues*, 2002.

[12] K. C. Yeager, "The MIPS R10000 superscalar microprocessor," *IEEE Micro*, vol. 16, pp. 28–41, Apr. 1996.

[13] B. H. Bloom, "Space/Time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13(7), pp. 422–426, July 1970.

[14] A. Kumar, P. Kundu, A. P. Singh, L.-S. Peh, and N. K. Jha, "A 4.6Tbits/s 3.6GHz Single-cycle NoC Router with a Novel Switch Allocator in 65nm CMOS," *ICCD-2007*, October 2007.

[15] M. Franklin and G. S. Sohi, "ARB: A Hardware Mechanism for Dynamic Reordering of Memory References," *IEEE Transactions on Computers*, vol. 45, pp. 552–571, May 1996.

[16] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, "Multiscalar processors," in *ISCA-22*, June 1995.

[17] C. K. Tang, "Cache Design in the Tightly Coupled Multiprocessor System," in *AFIPS Conference Proceedings, National Computer Conference*, June 1976, pp. 749–753.

[18] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz, "An Evaluation of Directory Schemes for Cache Coherence," in *ISCA-15*, May-June 1988, pp. 280–289.

[19] S. Sethumadhavan, R. Desikan, D. Burger, C. R. Moore, and S. W. Keckler, "Scalable hardware memory disambiguation for high ILP processors," in *MICRO-36*, 2003.

[20] I. Park, C. L. Ooi, and T. N. Vijaykumar, "Reducing design complexity of the load/store queue," in *MICRO-36*, 2003.

[21] G. Z. Chrysos and J. S. Emer, "Memory Dependence prediction using store sets," in *ISCA-25*, June 1998, pp. 142–153.

[22] S. Sethumadhavan, F. Roesner, J. S. Emer, D. Burger, and S. W. Keckler, "Late-Binding: Enabling Unordered Load-Store Queues," in *ISCA-34*, June 2007.

[23] A. Moshovos and G. S. Sohi, "Streamlining Inter-operation Memory Communication via Data Dependence Prediction," in *MICRO-30*, December 1997.

[24] S. Subramaniam and G. H. Loh, "Fire-and-Forget: Load/Store Scheduling with No Store Queue at All," in *MICRO-39*, December 2006.

[25] T. Sha, M. Martin, and A. Roth, "NoSQ: Store-Load Communication without a Store Queue," in *MICRO-39*, December 2006.