A Power-Efficient and Scalable Load-Store Queue Design

F. Castro[1], D. Chaver[1], L. Pinuel[1], M. Prieto[1], M. C. Huang[2], F. Tirado[1]

[1] ArTeCS Group, Complutense University of Madrid,
Madrid, Spain
fcastror@fis.ucm.es
{dani02, mpmatias, ptirado, lpinuel}@dacya.ucm.es

[2] University of Rochester,
Rochester, New York, USA
{Michael.Huang}@ece.rochester.edu

Abstract. The load-store queue (LQ-SQ) of modern superscalar processors is responsible for keeping the order of memory operations. As the performance gap between processing speed and memory access becomes worse, the capacity requirements for the LQ-SQ increase, and its design becomes a challenge due to its CAM structure. In this paper we propose an efficient load-store queue state filtering mechanism that provides a significant energy reduction (on average 35% in the LSQ and 3.5% in the whole processor), and only incurs a negligible performance loss of less than 0.6%.

1 Introduction

As the performance gap between processing and memory access widens, load latency becomes critical for performance. Modern microprocessors try to mitigate this problem by incorporating sophisticated techniques to allow early execution of loads without compromising program correctness.

Most out-of-order processors include two basic techniques usually denoted as *load bypassing* and *load forwarding*. The former allows a load to be executed earlier than preceding stores when the load effective address (EA) does not match with any of the preceding stores. If the EA of any preceding store is not resolved when the load is ready to execute, it must wait. When a load aliases (has the same address) with a preceding store, *load forwarding* allows the load to receive its data directly from the store.

More aggressive implementations allow *speculative execution of loads* when the effective address of a preceding store is not yet resolved. Such speculative execution can be premature if a store earlier in the program order overlaps with the load and executes afterwards. When the store executes, the processor needs to detect, squash and re-execute the loads. All subsequent instructions, or at least, the dependent instructions of the load need to be re-executed as well. This is referred to as load-store replay [1]. To mitigate the performance impact of replays, some designs incorporate predictor mechanisms to control such speculation [2]. One commercial example of this technique can be found in Alpha 21264 [1] (and its follow-on generation, Alpha 21364). The processor uses a simple 1-bit PC-indexed table to predict whether a load will be dependent upon a previous store or not. Loads predicted to be independent will execute as soon as their effective address is available, while other loads wait until all prior stores are resolved.

Modern processors with support for glueless multiprocessor systems, such as Intel's Itanium or IBM's POWER 4, also have to include support for memory consistency in shared memory multiprocessors systems [3, 4], which makes the memory system even more complex.

These complex mechanisms come at the expense of high energy consumption. Our investigated design accounts for around 8% of the total consumption. Furthermore, it is expected that this consumption will grow in future designs, because structures buffering in-flight load and stores (LSQ) need to be scaled up to buffer more load and stores to allow the processor to effectively tolerate longer memory latencies. Driven by this trend, our main objective in this paper is to design an efficient and scalable LSQ structure, which could save energy without sacrificing performance.

The rest of the paper is organized as follows. Section 2 and 3 present the conventional design and our alternative LSQ mechanism respectively. Section 4 describes our experimental framework. Section 5 analyzes experimental results. Section 6 discusses related work. Finally, Section 7 presents our conclusions.

2 Conventional Design of Load-Store Queue

Conventional implementations of the LSQ are usually based on two separate queues, the load queue (LQ) and the store queue (SQ) (see Figure 1). Each entry of these queues provides a field for the load or store effective address and a field for the load or store data. For example, the Alpha 21264 microprocessor [1] incorporates 32+32 entries for the load and store queues.

Techniques such as *load forwarding and load bypassing* increase the complexity of these structures given that they require associative search in the store queue to detect aliasing between loads and preceding stores. *Speculation* increases complexity even more since associative search is also necessary in the load queue to detect premature loads: when the EA of a store is resolved, an associative search in the load queue is performed; if a match with a later load that has already been issued is found, execution from the conflicting load is squashed.

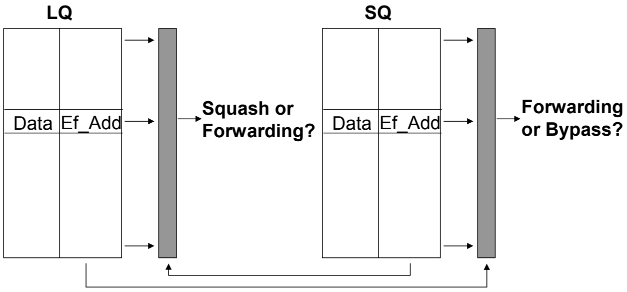


Fig. 1. Conventional load-store queue design. Each load instruction performs an associative search of the SQ to determine if load forwarding or load bypassing is necessary. Each store also performs an associative search of the LQ to forward data for a later load with the same address, which can happen if the load is waiting for a memory port to be free, or to squash its execution if it is already executing.

3 Efficient Load-Store Queue Management

3.1 Rationale

While the techniques outlined above such as *load forwarding* and *load bypassing* improve performance, they also require a large amount of hardware that increases energy consumption. In this paper, we propose a more efficient LSQ design that allows for a more efficient energy usage. Besides, the size of the structure scales significantly better than the conventional approach described before.

The new approach is based on the following observations:

1- Memory dependencies are quite infrequent. Our experiments indicate that only around 11% of the load instructions need a *bypass*. This suggests that the complex disambiguation hardware available in modern microprocessors is often being underutilized.

2- On average, around 73% of the memory instructions that appear in a program are loads. Therefore, their contribution to the dynamic energy spent by the disambiguation hardware is much greater than that of the stores. This suggests that more attention must be paid to handling loads.

3.2 Overall Structure

As shown in Figure 2, the conventional LQ is split into two different structures: the Associative Load Queue (ALQ) and the Banked Non-associative Load Queue (*BNLQ*). ALQ is similar to a conventional LQ, but smaller. It provides fields for the effective address and the load data, as well as control logic to perform associative searches. The *BNLQ* consists of a simple buffer to hold the load effective address and the load data. An additional mechanism, denoted as Exclusive Bloom Filter (EBF), is added to assure program correctness. To distribute load instructions into these two queues, we employ a dependence predictor. We describe the operation of each component in the following.

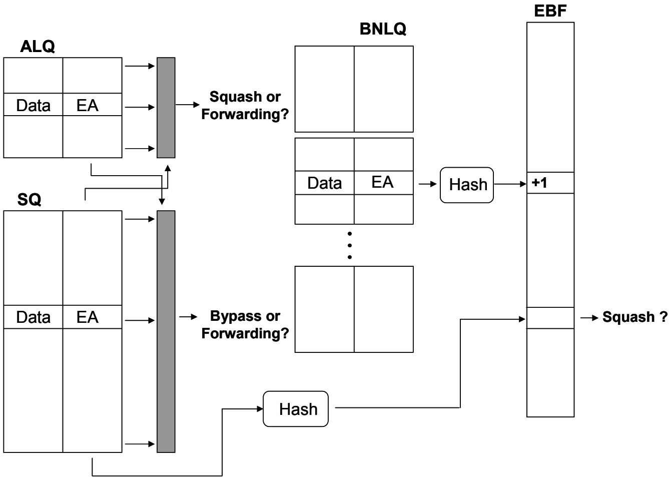


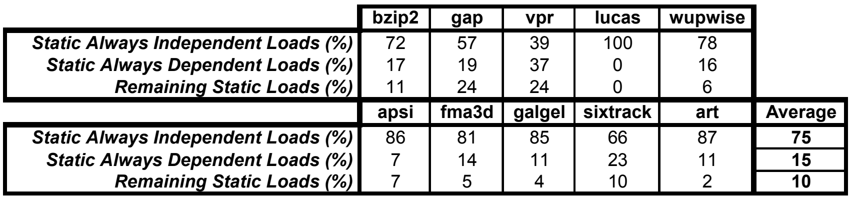
Fig. 2. Effective LQ-SQ Management. LQ is divided into two queues, an associative load queue (ALQ) for those predicted-dependent loads and a Banked Non-associative Load Queue (*BNLQ*) for those predicted-independent loads. The ALQ and the SQ work similarly to a conventional LQ-SQ. To ensure premature loads are detected even if they are sent to BNLQ, an Exclusive Bloom Filter (EBF) is used.

3.3 Distribution of Loads and Dependency Prediction

The distribution of loads between the ALQ and the BNLQ is a key aspect to be considered. Our distribution is based on a broadened notion of dependency. We classify as *dependent-loads* those loads that while in-flight, happen to have the same effective address as any store from the SQ. This would include loads that overlap with younger in-flight stores as well. A more conventional definition of dependent load would only consider those loads that receive their data directly from an earlier, in-flight store – loads that overlap with older, in-flight stores. This broadened definition is due to the imprecise (and hence more energy-efficient) disambiguation mechanism used for handling independent loads as we will explain later. Using our alternative definition, 23% of loads (on average) are classified as dependent.

After decoding a load instruction, it will be suggested by either a *dynamic dependency* predictor or *profiling-based* information, into which queue the load should be accommodated. According to this information and taking into account the queue occupancy, the load is allocated in the ALQ or the BNLQ. Occupancy information is used to improve the balance among both queues. In our current design, if an independent load arrives and the BNLQ is full, it is accommodated in the ALQ, since an independent load can always be allocated into the associative queue without compromising program correctness. Although it is out of the scope of this paper, more sophisticated distribution policies can be devised. For example, distribution could also take into account the occupancy of the different BNLQ banks.

Table 1. Static load distribution. We show for each benchmark the percentage of static loads that are independent for all its dynamic instances, the percentage of static loads that are dependent for all dynamic instances, and the rest of the static loads in the program.



	bzip2	gap	vpr	lucas	wupwise	apsi	fma3d	galgel	sixtrack	art	Average
Static Always Independent Loads (%)	72	57	39	100	78	86	81	85	66	87	**75**
Static Always Dependent Loads (%)	17	19	37	0	16	7	14	11	23	11	**15**
Remaining Static Loads (%)	11	24	24	0	6	7	5	4	10	2	**10**

In a profiling-based system, every static load is tagged as dependent or independent based on profile. This way, load dependency prediction is tied to the static instructions. Results in Table 1 show that this is a reasonable approach given that the runtime dependency behavior of static load instructions remains very stable. Only 10% (on average) of the load instructions in our simulations change their dependency behavior during program execution. Nevertheless, apart from profiling, load annotation involves some changes in the instruction set so that it is possible to distinguish between dependent and independent load instructions.

We have chosen a dynamic approach where dependency prediction is generated dynamically. To store the prediction information, we can augment the instruction cache or use a dedicated, PC-indexed table. We have opted to use a PC-indexed prediction table as in the Alpha 21264 [1]. All loads are initially considered as independent. However, as will be explained below, this initial prediction is changed if a potential dependency is detected. Once a prediction has been changed, this prediction will hold during the rest of the execution. This decision, which is based on the stable behavior observed in Table 1, simplifies the implementation and has little impact on the prediction accuracy. Unlike the profiling approach, this alternative needs neither change in the instruction set architecture nor profiling. However, as mentioned above, it requires extra storage and a prediction training phase.

Our results indicate that both strategies provide similar performance. In this paper, we have only explored the dynamic version given that its extra cost (in terms of design complexity and power) is insignificant.

3.4 Load-Store Replay and Dependency Predictor Update

Once the memory instructions are distributed to their corresponding queues, it is necessary to perform different tests to detect violations of the memory consistency.

As described above, in a conventional LQ-SQ mechanism, these violations are detected using associative searches. Our proposed mechanism follows the same strategy for those loads accommodated in the ALQ. However, for those loads accommodated in the BNLQ, an alternative mechanism needs to be incorporated to detect potential violations. As in [5], our implementation adds a small table of two-bit counters, denoted as EBF (Figure 2). When a load of the BNLQ is issued, it indexes the EBF based on its effective address and increases the corresponding counter. When the load commits, the counter is decremented.

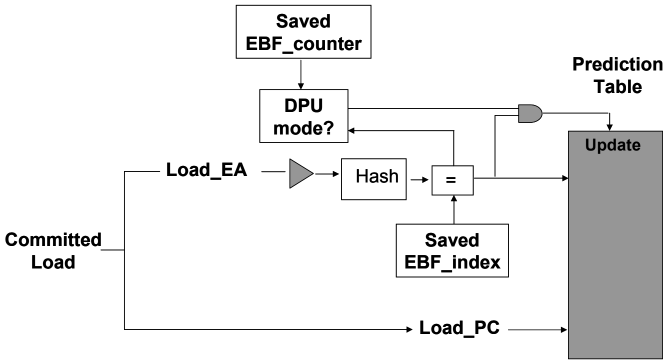


Fig. 3. Dependence predictor update in the DPU mode when loads commit.

When stores are issued, miss-speculated loads accommodated in the ALQ are detected by performing an associative search, as in a conventional mechanism. Miss-speculated loads in the BNLQ are detected by indexing the EBF with the store's EA. If the corresponding counter is greater than zero, a potentially truly dependent load is in-flight. In this case, our mechanism conservatively squashes execution from the subsequent load and triggers a special mode, denoted as DPU (*dependency predictor update mode*), to update the dependency prediction information. Note that any in-flight load that aliases with the store in the EBF causes a squash, independent of its relative position to the store. This is why we use a broadened notion of dependency.

Two different hashing functions (H0 and H1) have been proposed in [5] for indexing the EBF. H0 uses lower order bits of the memory instruction address to index into the hash table, whereas the H1 uses profile to index the EBF using those bits in the addresses that were the most random. We have opted to employ H0, given that H1 does not provide any significant improvement. In fact, for a large enough EBF, H0 outperforms H1 in our simulations.

When a store triggers the DPU mode, its corresponding EBF index and EBF counter are saved in special registers (see Figure 3). During the DPU mode, when a load commits,

its prediction is changed to dependent if its hashed EA matches with the EBF index value. Based on the saved EBF counter value, this mode is held until all the in-flight loads that aliased with the trailing store have committed.

4 Experimental Framework

We evaluated our proposed load-store queue design on a simulated, generic out-of-order processor. The main parameters used in the simulations are summarized in Table 2. As the evaluation tool, we employed a heavily modified version of SimpleScalar [6] that incorporates our LQ-SQ model and a tuned Wattch framework [7] that models the energy consumption in the proposed LQ-SQ mechanism.

Table 2. System Configuration

Processor	**Caches and Memory**
8-issue out of order processor **Register File:** 256 integer physical registers 256 floating-point physical registers **Units:** 4 integer ALUs, 2 integer Mult-Dividers, 3 floating-point ALUs, 1 floating-point Mult-Dividers **Branch Predictor:** Combined, bimodal: 8K entries 2level, 8K entries, 13 bits history size, meta-table: 8K entries BTB: 4K entries RAS: 32 entries **Queues:** I-Queue: 128 entries FP-Queue: 128 entries	**L1 data cache:** 32-KB, 4-way, LRU, latency: 3 cycles **L2 data cache:** 2-MB, 8-way,LRU, latency: 12 cycles **L1 instruction cache:** 64-KB, 2-way, LRU, latency: 2 cycles **Memory access:** 100 cycles

Standard LQ-SQ	**Proposed LQ-SQ**	**EBF-based LQ-SQ**
LQ: 80 entries	ALQ: 32 entries	Non-Asoc-LQ: 80 entries
SQ: 48 entries	BNLQ: 3x16 entries	SQ: 48 entries
	SQ: 48 entries	EBF: 4K entries
	EBF: 4K entries	

The evaluation of our proposal has been performed selecting several benchmarks from the SPEC CPU2000 suite. In selecting those applications, we tried to cover a wide variety of behavior. We simulated single sim-point regions [8] of one hundred million instructions.

We have simulated three different schemes: a conventional LQ-SQ mechanism (baseline configuration), our proposed alternative and the original *state filtering* scheme proposed in [5]. The associative LQ from the conventional approach provides 80 entries, whereas the ALQ from our proposal provides only 32 entries. Our BNLQ allocates 3 banks with 16 entries in each bank, whereas the non-associative LQ (NLQ) of the original scheme has 80 entries. The EBF provides 4096 2-bit saturating counters. For this reason, if more than 4 issued loads happen to hash into the same entry, a potential issue of a fifth load would stall until any of the others commit (this situation almost never occurs in our experiments).

5 Experimental Results

Tables 3 and 4 report performance gains over the baseline configuration, as well as energy savings achieved in the load-store queue and in the whole processor. The original proposal of Sethumadhavan [5] manages to reduce the energy spent in the LQ-SQ mechanism (Table 3), but both overall energy and performance suffer a significant drop due to squashes caused by dependence violations.

Table 3. Energy savings and performance loss for the original Sethumadhavan's *state filtering* approach. A negative value for performance denotes performance loss over the conventional LQ-SQ, while a positive value for the energy means energy savings.

	bzip2	gap	vpr	lucas	wupwise	apsi	fma3d	galgel	sixtrack	art	Average
∆IPC (%)	-37,3	-9,5	-21,9	-7,3	-14,3	-15,7	-5,6	-5,5	-30,4	-41,6	**-18,9**
∆ LQ-SQ Energy (%)	-3,7	16,5	5,8	13,2	26,1	23,7	30,5	33,4	-2,6	-14,8	**12,8**
∆Energy (%)	-24,0	-5,3	-18,6	-3,4	-5,2	-5,3	-0,7	0,1	-28,3	-46,8	**-13,8**

Our scheme (Table 4) outperforms this previous proposal by reducing dramatically the number of squashes, which allows for a negligible performance loss (less than 0.6% on average). Besides, the splitting of the LQ into a set of smaller tables (ALQ + 3xBNLQ) further improves energy consumption. On average, our implementation saves 35% energy in the LQ-SQ, which translates into an important reduction in the whole processor (around 3.25%).

We should remark that, although the end results are quite satisfactory, our current dependency predictor is far from ideal. On average, about 50% of all loads are classified as dependent, most falsely so due to conflict of hashing function: only 23% of loads will be classified as dependent loads if a hash table with an infinite size is used. This observation suggests that there is still significant room for improvement, which motivates us to include more accurate prediction and bloom filter hashing in future implementations.

Table 4. Energy savings and performance loss of our proposed LQ-SQ design. A negative value for performance denotes performance loss over the conventional LQ-SQ, while a positive value for the energy means energy savings.

	bzip2	gap	vpr	lucas	wupwise	apsi	fma3d	galgel	sixtrack	art	Average
*Δ**IPC (%)***	-0,02	-1,08	-0,47	-0,28	-0,01	-1,92	-0,01	-0,50	-0,05	-1,32	**-0,57**
*Δ **LQ-SQ Energy (%)***	32,9	29,4	31,4	24,6	44,7	45,6	49,2	29,5	32,4	33,2	**35,28**
*Δ**Energy (%)***	3,59	3,66	3,44	1,45	4,37	3,72	4,23	2,77	4,26	2,24	**3,37**

6 Related Work

Recently, a range of schemes have been proposed that use approximate hardware hashing with Bloom filters to improve LSQ scalability [5, 9]. These schemes fall into two broad categories. The first, called *search filtering*, reduces the number of expensive, associative LSQ searches [5], and in partitioned search filtering [9], reduces the number of entries that the hardware must search if a search must be done. In the second category, called *state filtering*, LSQs handle memory instructions that are likely to match others in flight, and Bloom filters (denoted as EBF in [5]) encode the other operations that are unlikely to match. In [5] they report only one simple preliminary design, in which the LQ is completely eliminated and all loads are hashed in the EBF. Experimental results using such scheme suffered a significant performance loss. Our work improves upon prior art and provides an EBF-based design with a negligible performance penalty (1% on average). In addition, we have also extended early studies by evaluating the power consumption of both schemes.

In the LSQ domain, several researchers have proposed alternative solutions to the disambiguation challenges. Park *et al.* [10] proposed using the dependence predictor to filter searches to the store queue. Cain and Lipasti [11] proposed a value-based approach to memory ordering that enables scalable load queues. Roth [12] has proposed combining Bloom filters, address-partitioned store queues, and first-in, first-out retirement queues to construct a high-bandwidth load/store unit. Baugh *et al.* [13] proposed an LSQ organization that decouples the performance-critical store-bypassing logic from the rest of the load-store queue functionality, for which they also used address-partitioned structures.

7 Conclusions and Future Work

In this paper we have presented a *state filtering* scheme, which with a negligible performance penalty allows for a significant energy reduction: around 35% for the LSQ, and close to 3.5% for the whole processor. These experimental results were obtained using a moderate configuration, in terms of instruction window and LSQ sizes. When employing more aggressive configurations, the expected energy savings would be much higher. The key points of our proposal are the following:

– We have designed a simple dependence predictor, specially adapted for using with Exclusive Bloom Filters (traditional predictors are not suitable for this scheme).
– We have explored the asymmetric splitting of LQ/SQ, as well as ALQ/BNLQ.
– Since the BNLQ is not associative, banking is straightforward and provides further energy reductions. This also simplifies *gating*.

Our future research plans include applying *bank gating*, as well as dynamic structure resizing, both based on profiling information. In addition, we also envisage enhancing both the bloom filter hashing and the dependence predictor, since the actual implementation is too conservative and there is a significant room for improvement.

References

1. R. E. Kessler. "The Alpha 21264 Microprocessor". Technical Report, Compaq Computer Corporation, 1999.
2. B. Calder and G. Reinman. "A Comparative Survey of Load Speculation Architectures". Journal of Instruction-Level Parallelism, May-2000.
3. C. Nairy and D. Soltis. "Itanium-2 Processor Microarchitecture". IEEE-Micro, 23(2):44-55, March/April, 2003.
4. J. M. Tendler, J. S. Dodson, J. S. Fields Jr., H. Le and B. Sinharoy. "Power-4 System Microarchitecture". IBM Journal of Research and Development, 46(1):5-26, 2002.
5. S. Sethumadhavan, R. Desikan, D. Burger, Charles R. Moore, Stephen W. Keckler. "Scalable Hardware Memory Disambiguation for High ILP Processors". Proceedings of MICRO-36, December-2003.
6. T. Austin, E. Larson, and D. Ernst. "SimpleScalar: An Infrastructure for Computer System Modeling". Computer, vol. 35, no. 2, Feb 2002.
7. D. Brooks, V. Tiwari, and M. Martonosi. "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations". 28-ISCA, Göteborg, Sweden. July, 2001.
8. T. Sherwood, E. Perelman, G. Hamerly, B. Calder. " Automatically charecterizing large scale program behavior ". Proceedings of ASPLOS-2002, October-2002.
9. S. Sethumadhavan, R. Desikan, D. Burger, Charles R. Moore, Stephen W. Keckler. "Scalable Hardware Memory Disambiguation for High ILP Processors". IEEE-Micro, Vol. 24, Issue 6:118-127, November/December, 2004.
10. I. Park, C. Liang Ooi, T. N. Vijaykumar. "Reducing design complexity of the load-store queue". Proceedings of MICRO-36, December-2003.
11. H. W. Cain and M. H. Lipasti. "Memory Ordering: A Value-Based Approach". Proceedings of ISCA-31, June-2004.
12. A. Roth. "A high-bandwidth load-store unit for single- and multi- threaded processors". Technical Report, University of Pennsylvania, 2004.
13. L. Baugh and C. Zilles. "Decomposing the Load-Store Queue by Function for Power Reduction and Scalability". Proceedings of PAC Conference, October-2004.