SAMIE-LSQ: Set-Associative Multiple-Instruction Entry Load/Store Queue

Jaume Abella and Antonio González

Intel Barcelona Research Center
Intel Labs, Universitat Politècnica de Catalunya
Barcelona (Spain)

Computer Architecture Department
Universitat Politècnica de Catalunya
Barcelona (Spain)

{jaumex.abella, antonio.gonzalez}@intel.com

Abstract

The load/store queue (LSQ) is one of the most complex parts of contemporary processors. Its latency is critical for the processor performance and it is usually one of the processor hotspots.

*This paper presents a highly banked, set-associative, multiple-instruction entry LSQ (*SAMIE-LSQ*) that achieves high performance with small energy requirements. The* SAMIE-LSQ *classifies the memory instructions (loads and stores) based on the address to be accessed, and groups those instructions accessing the same cache line in the same entry. Our approach relies on the fact that many in-flight memory instructions access the same cache lines. Each* SAMIE-LSQ *entry has space for several memory instructions accessing the same cache line. This arrangement has a number of advantages. First, it significantly reduces the address comparison activity needed for memory disambiguation since there are less addresses to be compared. It also reduces the activity in the data TLB, the cache tag and cache data arrays. This is achieved by caching the cache line location and address translation in the corresponding* SAMIE-LSQ *entry once the access of one of the instructions in an entry is performed, so instructions that share an entry can reuse the translation, avoid the tag check and get the data directly from the concrete cache way without checking the others. Besides, the delay of the proposed scheme is lower than that required by a conventional LSQ.*

We show that the SAMIE-LSQ *saves 82% dynamic energy for the load/store queue, 42% for the L1 data cache and 73% for the data TLB, with a negligible impact on performance (0.6%).*

1. Introduction

As technology evolves, power dissipation increases and cooling systems become more complex and expensive. Reduction of power dissipation in the hottest spots of the processor can be very beneficial not only in terms of energy reduction, but also for reducing cooling costs or increasing performance for a given thermal solution.

The load/store queue (LSQ) of superscalar processors is one of their main hotspots. Besides, the LSQ has a significant delay due to its complexity and it is difficult to pipeline. Overall, the design of a low latency and low power LSQ is an important challenge for continuing scaling up the performance of superscalar processors.

The LSQ is typically implemented using fully-associative schemes to check dependences between load and store instructions. When the address of a load instruction is known, it must be compared with the address of the older in-flight store instructions to catch the right data in case of a match. Similarly, when the address of a store instruction is known, it must be compared with the address of the younger in-flight load instructions to forward them the right data in case of a match. Even though this approach may help to improve instruction-level parallelism (ILP), its latency may be high and offset its potential benefits. Additionally, its complexity grows drastically if we increase the number of ports or the LSQ size. Large instruction windows are required for augmenting the opportunities to extract more ILP, which in turn requires wider pipelines. Hence, large and highly ported LSQs are desirable in high-performance processors, provided that their latency and power dissipation is reasonable.

This paper presents a set-associative, multiple-instruction entry LSQ (*SAMIE-LSQ*), which is a new LSQ scheme for low power and low-complexity. The *SAMIE-LSQ* is much more suitable for high-performance superscalar processors with large instruction windows than conventional LSQs and scales much better than a fully-associative LSQ. The *SAMIE-LSQ* design is based on two observations: first, many in-flight memory instructions access the same cache line so they can be placed in entries where the cache line address is shared by several load/store instructions; second, in-flight load/store instructions access very few cache lines with the same low-order bits, and thus, we can use a set-associative structure since it produces few conflicts for most of the programs.

The *SAMIE-LSQ* achieves significant energy savings with respect to a conventional LSQ. Moreover, since the *SAMIE-LSQ* entries can hold multiple memory instructions, it enables caching some information like the location of the cache line in the L1 data cache and the address translation provided by the data TLB. As a consequence, a significant number of load

1-4244-0054-6/06/$20.00

and store instructions do not need to check L1 data cache tags nor access all ways of a set-associative L1 data cache, and the number of data TLB accesses is reduced. This results in significant energy savings in the L1 data cache and the data TLB with negligible performance overhead.

The rest of the paper is organized as follows. Section 2 reviews some related work. Section 3 presents the proposed scheme and section 4 evaluates its performance. Section 5 summarizes the main conclusions of this work.

2. Related Work

Dynamic memory disambiguation has been extensively studied. This section reviews some techniques to increase the performance and/or save energy of the logic devoted to disambiguate loads and stores.

Some techniques [3][6][8][14] focus on predicting dependences between loads and stores. If the address of a load is known but there are older stores whose addresses are still unknown, they predict whether the load depends on those stores or not. On a missprediction, a significant overhead may be incurred because the pipeline must be flushed like in a branch misspredict ion.

Other approaches [5] simplify the logic devoted to memory disambiguation by executing loads without comparing their addresses against stores addresses. Loads are later validated by re-executing them right before commit. If there is a mismatch between the data loaded at the execution stage and the data loaded at the re-execution stage, then the pipeline is flushed. Different mechanisms [2][10][11] have been proposed to filter the number of instructions that need re-execution since they require memory ports, which are a scarce resource.

Based on the observation that it is usual to have some loads in-flight that fetch the same data, Nicolaescu *et al.* [7] propose forwarding the data to among loads instead of accessing several times the same data in cache. This technique reduces the L1 data cache energy consumption but requires that loads can obtain their data forwarded not only from stores, but also from loads.

Sethumadhavan *et al.* [12] propose using hash encoding of the memory addresses to check dependences between loads and stores. When the filter (Bloom filter) predicts that a given load or store has no dependences with other memory instructions, the instruction can be executed safely. On the other hand, if the filter predicts that the memory instruction may have a conflict, the associative search in the LSQ must be done to check whether the dependence exists or not. This mechanism saves a significant number of power-hungry associative searches in the LSQ by checking only the low power filter, but does not reduce the intrinsic complexity of the LSQ and introduces indeterminism in the latency to check address dependences.

Park *et al*. [9] propose a segmented LSQ to reduce its latency although checking for the dependences of a load/store may take several cycles since LSQ segments are checked sequentially.

Franklin and Sohi [4] propose distributing the LSQ into N banks and classifying the instructions in the banks according to the addresses they access. Each bank has M different addresses, and each address has space for P instructions, being P the maximum number of in-flight loads/stores allowed. There is space for $N \cdot M \cdot P$ instructions but only P instructions are allowed in total. This scheme relies on the idea that as we increase N, M can be decreased. As shown in the next section, even if N is large, many programs require M to be also large. Thus, $N \cdot M$ must be large not to lose significant performance, which implies that a lot of space is wasted, and small benefits are achieved with respect to a conventional LSQ.

We propose the *SAMIE-LSQ*, which is an extreme distribution of the LSQ into multiple queues that is based on the loads/stores addresses and requires very few entries per queue. We add a small queue for instructions that do not have room in their corresponding queue. Our approach is based on the observation that it is very common having several in-flight instructions that access the same cache line. The proposed LSQ can hold several instructions that access the same cache line in a single entry, which results in several benefits:

- Loads and stores that are placed in the LSQ must compare their address only with other very few cache line addresses, which saves significant energy.
- Once an instruction has accessed the L1 data cache, the LSQ entry records where the cache line is located. Then, further accesses to the same cache line can access the data cache as if it was a direct-mapped cache (just a single bank) even if the cache is set-associative in practice, and it is not necessary to compare the tag. Hence, many cache accesses require little energy and lower latency.
- Once an instruction has accessed the data TLB to translate its address, the translation can be cached in the LSQ entry. The other instructions of the same LSQ entry do not access the TLB, which results in significant energy savings in the TLB and may reduce the latency of memory instructions.

Additionally, *SAMIE-LSQ* can be easily combined with any technique that filters the number of accesses to the LSQ, and with those techniques that execute loads speculatively.

3. SAMIE-LSQ

The load/store queue is one of the most complex components of today's microprocessors. Its energy requirements and complexity motivates the research on alternative designs. As outlined in the previous section, the ARB [4] reduces the complexity and energy consumption of the LSQ by distributing it into several banks where instructions are allocated depending on the memory address to be accessed. To achieve significant dynamic power savings the ARB requires a high degree of banking of their LSQ. Figure 1 shows the performance of ARB with respect to an unbounded size LSQ for different configurations of number of banks and addresses per bank. For instance, the

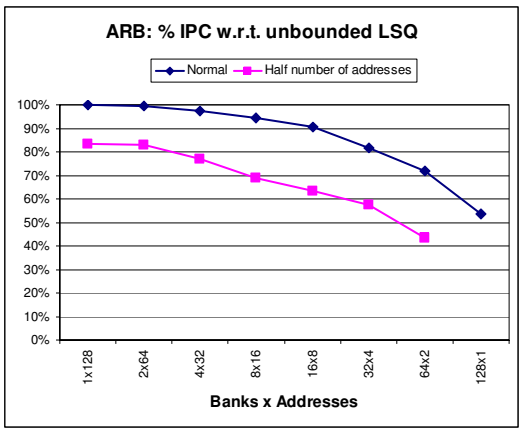

Figure 1. IPC of ARB with respect to an ideal unbounded LSQ. Configurations with different number of banks and addresses per bank are shown.

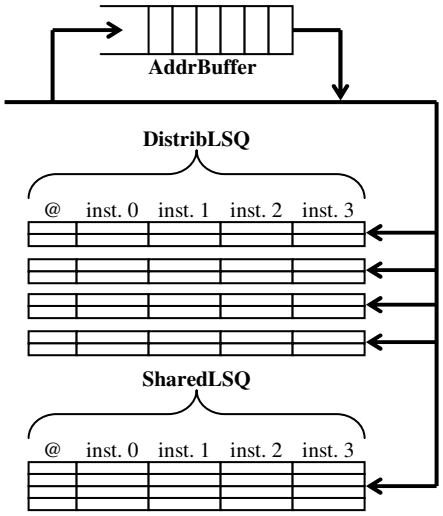

Figure 2. *SAMIE-LSQ* organization.

configuration 2x64 corresponds to having 2 banks with 64 different addresses each. The processor configuration is detailed in the evaluation section. The most relevant processor parameters are its width (8) and the window size (256 instructions).

Looking at Figure 1 we can see that when the number of banks is very low, the dynamic power savings are very low since the number of addresses to be compared in a given bank is significant. As we increase the degree of banking and reduce the number of entries per bank, the power savings potential increases but the performance decreases dramatically. The configuration with 64 banks and 2 addresses each loses as much as 28% IPC. Additionally, the ARB requires that each entry has space for an address and in the worst case as many memory instructions as total number of in-flight instructions. Thus, the leakage of ARB may be very high. For instance, assuming a maximum of 128 loads and stores in-flight, the ARB storage has to be as large as 16384 (128·128) loads and stores. On the other hand, reducing the number of banks, addresses per bank or allowed in-flight memory instructions significantly harms performance. For instance, we can observe in the figure the performance when the number of in-flight memory instructions allowed is reduced to the half. The performance loss is 16% for the fully associative configuration (1 bank with 64 addresses).

Our objective is distributing the LSQ in many small banks to save dynamic power with moderate total LSQ storage requirements. The distribution is based on the memory address to be accessed.

3.1 Structures

Figure 2 shows a block diagram of the *SAMIE-LSQ*. We observe three main structures: *DistribLSQ*, *SharedLSQ* and *AddrBuffer*.

DistribLSQ is a banked LSQ (4 banks in the figure). Each bank can hold instructions accessing to different data cache lines (2 different cache lines per bank in the example). For each LSQ entry we have an associated cache line and several instructions. We refer to these parts of an entry as slots. The basic information required for each instruction is its offset within the cache line, its relative age identifier used for data forwarding, and the data loaded or to be stored if available. Additionally, each instruction needs a bit to know if its data is available, another bit to know if older stores addresses are known, and some bits with other instruction information like the number of bytes to be loaded/stored, the type of instruction (load/store), and the slot of the store that forwards its data (if any) in case this instruction is a load.

Those instructions that do not find an available entry/slot in its corresponding bank of the *DistribLSQ* are placed in the *SharedLSQ* whose entries have the same fields as the *DistribLSQ*. We assume 4 entries in the figure.

Finally, instructions that can be placed neither in the *DistribLSQ* nor in the *SharedLSQ*, are placed in a waiting buffer called *AddrBuffer*. Memory instructions in the *AddrBuffer* cannot access cache; they have to go first to the *DistribLSQ* or *SharedLSQ* for disambiguation. Each entry of this buffer holds the complete address to be accessed (cache line address and offset), the age identifier of the instruction, and those bits indicating whether it is a load or a store and how many bytes must be accessed.

To maintain coherence and do not allow loads to be issued before knowing the addresses of older stores, the reorder buffer is extended with some information. For each entry, there is a bit (*readyBit*) used for memory disambiguation. If the instruction is a store and its address is known, its *readyBit* is set. Any load has its bit set only if there are no older stores whose addresses are still unknown. A load cannot access memory until this bit is set. The reorder buffer entries have also a field telling where the instruction is placed (*whereLSQ*). The *whereLSQ* field is only relevant for loads. When a load is placed in the *DistribLSQ* or the *SharedLSQ*, this field is set with the corresponding location.

Every time that a store address is computed, its *readyBit* is set. In case there are no older stores whose addresses are still

unknown, it also sets the *readyBit* of all the following instructions in the reorder buffer until a store with unknown address is reached. All loads whose *readyBits* are set during this process are notified by using the *whereLSQ* field.

3.2 Operation

When the address of a memory instruction is computed, it is forwarded to the LSQ. The *DistribLSQ* is a set-associative structure where the banks are assigned in a direct-mapped manner based on the effective address and the entries of each bank are accessed in a fully-associative manner. If the instruction finds its cache line address in any of the entries of the corresponding bank and there is a free slot, the instruction fills this slot. If no entry has the same cache address or it is present but without free slots, a free entry is allocated and one of its slots is used.

If an instruction fails to be placed in the *DistribLSQ* due to lack of space, it is placed in the *SharedLSQ*, which is a small fully-associative structure. The process is the same: a free slot in an entry with the same cache line address is chosen if available; otherwise, an empty entry is allocated. Both structures are accessed in parallel, so the address of the load/store is compared with any other address in the corresponding bank of the *DistribLSQ* and all the addresses in the *SharedLSQ*.

Finally, if neither the *DistribLSQ* nor the *SharedLSQ* have room for the instruction, it is placed in the *AddrBuffer*. The instructions in the *AddrBuffer* have priority over the ones coming from the functional units when choosing which ones are to be placed in the *DistribLSQ* or the *SharedLSQ*.

3.3 Deadlock Avoidance

It may happen that the oldest in-flight instruction is in the *AddrBuffer* and it does not find free space in the LSQ (*DistribLSQ* and *SharedLSQ*) because younger instructions have filled the entries where this instruction can be placed. This is easily detected by checking whether the head instruction of the reorder buffer is not placed in the LSQ. Our evaluations show that sizing properly the different structures makes this happen very rarely (less than once every million instructions). Thus, in case of detecting this scenario we take the easy solution to avoid deadlocks: the pipeline is flushed. Since the oldest instruction will be the first to re-enter the pipeline, it will get an entry in the LSQ, which guarantees forward progress.

There is another situation where the *SAMIE-LSQ* might require the pipeline to be flushed: this is when an address computation finishes and it cannot be placed in any of the structures (*DistribLSQ*, *SharedLSQ* and *AddrBuffer*). Sizing the structures properly prevents this to happen. For instance, if the *AddrBuffer* has as many entries as in-flight memory instructions allowed, this situation will never happen. Note that the *AddrBuffer* is a simple FIFO structure so its complexity is rather low (e.g., no associative searches are performed in it). In our simulations, even assuming a smaller *AddrBuffer*, it never happens. An alternative solution would be not allowing address computations to be executed if they are not guaranteed to have at least one free slot in the *AddrBuffer*.

3.4 SAMIE-LSQ Extensions

SAMIE-LSQ puts several instructions that access the same cache line in the same entry of the *DistribLSQ* or the *SharedLSQ*. We take advantage of this to save energy in the L1 data cache and the data TLB.

We can save L1 data cache (Dcache for short) energy by caching the physical location of the cache line (set and way) in the corresponding LSQ entry once it is accessed, and adding a bit per cache line (*presentBit*) in the Dcache indicating whether its physical location has been cached in the LSQ or not. When the first instruction in a given entry accesses the Dcache, the physical location (set and way) of the cache line is stored in the LSQ entry and both the cache line and LSQ entry *presentBit* are set. Any other access to this cache line from this LSQ entry (note that all instructions in the same entry access the same cache line) needs neither to check the tags nor to read all the ways. These low power accesses read the data from the cache line of the concrete way without checking the tag. The storage to hold the physical location of the cache line requires just few bits. For instance, a 32KB cache with 32 bytes per line has 1024 lines, and thus, 10 bits are enough to record the physical location of the cache line. The *DistribLSQ* may require fewer bits to encode the physical cache line since, for a given *DistribLSQ* bank, only one or few sets can be accessed. This simple mechanism saves significant Dcache energy, and has two positive side effects: these accesses have lower latency, and we know that they will hit in advance. The benefits of these two effects are not considered in the performance study presented in this paper.

When a cache line is replaced, some LSQ entries in the *SharedLSQ* and the *DistribLSQ* may have to reset their *presentBit* flag. To avoid the comparison of the cache line address being replaced and the addresses in the LSQ, we use a very simple alternative, which consists of resetting the *presentBit* flag of all entries that can be potentially affected.

Data TLB (DTLB for short) energy is also saved by keeping the translated address in the LSQ entries. When the first instruction in the entry access the data cache, the DTLB is looked up and the address translation is cached in the corresponding entry of the *DistribLSQ* or *SharedLSQ*. The other instructions read this information from their LSQ entry. Similarly to the technique applied to save Dcache energy, there are two additional positive side effects whose benefit is not considered in the quantitative evaluation later in this paper: the translation has much lower latency and the translation hit rate may increase since the DTLB is not accessed for many instructions.

3.5 Sizing SAMIE-LSQ Structures

We initially experimented with a configuration without the *SharedLSQ* that places all instructions in the *DistribLSQ*. We

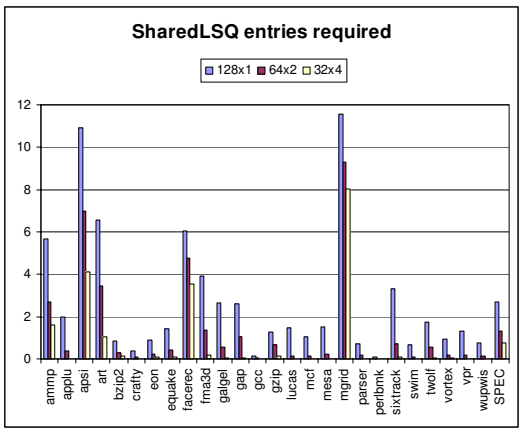

Figure 3. Average number of entries occupied in an unbounded *SharedLSQ* for different configurations of the *DistribLSQ*.

have found that different programs (we use Spec2000 benchmarks [15]) show extremely different address patterns. For instance, integer programs often require few entries in the *DistribLSQ* and these entries are distributed across different banks even if the number of banks is low. Hence, they hardly use the *SharedLSQ*. Most of the FP (floating-point) programs require a lot of entries in the *DistribLSQ*, but they exhibit different patterns. Some FP programs use evenly the different banks, which is beneficial to save energy in the *DistribLSQ*, but other FP programs concentrate most of their entries in few banks even if the number of banks is high. Increasing the number of entries of all the *DistribLSQ* banks is a waste of space since only a few banks will use them at any given point in time. Thus, a most cost-effective solution is using the *SharedLSQ* to hold the instructions that cannot be placed in the *DistribLSQ*.

Another important design parameter is the number of slots per entry. A large number benefits energy savings for address comparisons, Dcache and DTLB, for those programs where the number of in-flight memory instructions accessing the same L1 data cache line is high. On the other hand, there are some programs that do not take advantage of a large number of slots per entry.

Summing up, we need a highly banked *DistribLSQ* with enough entries to place most memory instructions, and some entries in the *SharedLSQ* for conflicting addresses. Figure 3 shows the average occupancy of the *SharedLSQ* for different configurations of the *DistribLSQ* varying the degree of banking (banks *x* entries per bank). The *SharedLSQ* is assumed to be unbounded, and there are 8 slots per entry in both the *DistribLSQ* and the *SharedLSQ*. Other configuration details are reported in the evaluation section. We observe that a configuration with 128 banks of 1 entry each (128x1) requires a significant number of entries in the *SharedLSQ* for many programs. That means that the *SharedLSQ* must be quite large and many comparisons will have to be done since each address is compared with the addresses of the corresponding bank of the *DistribLSQ* and all the addresses in the *SharedLSQ*. Thus, this configuration is too much banked.

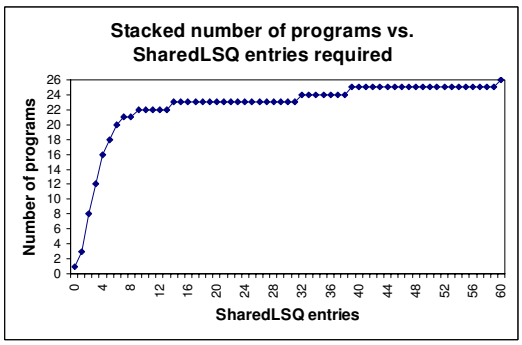

Figure 4. Number of programs that do not use the *AddrBuffer* during the 99% of their execution for a varying number of *SharedLSQ* entries.

On the other hand, we observe that the *SharedLSQ* space requirements of the 64x2 *DistribLSQ* are only a bit higher than those of the 32x4 *DistribLSQ*. Thus, we select the 64x2 configuration of the *DistribLSQ* because its banks are small and its *SharedLSQ* space requirements low.

Figure 4 shows the number of programs that need a given number of *SharedLSQ* entries in order not to require the *AddrBuffer* during 99% of the time. It can be seen that 4 entries are enough for 16 over 26 programs, so 10 programs may lose some performance, whereas 8 entries are enough for 21 programs. If we consider a *SharedLSQ* with 12 entries only one more program has enough entries during the 99% of the time of its execution. Hence, an 8-entry *SharedLSQ* seems a good tradeoff and is what we assume in our experiments.

The number of slots per entry is set to 8. More slots per entry would help to reduce the energy consumption since more instructions may benefit from power reductions when accessing the Dcache and the TLB. The drawback of increasing the number of slots per entry is that leakage and delay are increased.

Using a lower number of slots per entry would help to save leakage and reduce the delay, but it is counterproductive for some programs whose memory references tend to concentrate in few cache lines, because the associated LSQ bank or the *SharedLSQ* would require more entries. This may offset the benefits of reducing the number of slots per entry.

As shown in Figure 3 some programs require a large number of entries in the *SharedLSQ*. When they fail to place an instruction in the *SharedLSQ*, the instruction has to wait in the *AddrBuffer*. We have observed that an *AddrBuffer* of 64 entries is always enough for all programs. A few programs such as `ammp` and `facerec` need more than 32 entries more than 5% of their execution time. Since the *AddrBuffer* is a cheap structure in terms of energy and delay, we set its size to 64 entries.

3.6 Delay

The delay of the different components has been evaluated using CACTI 3.0 [13] with 0.10µm technology. The largest delay for *SAMIE-LSQ* corresponds to *DistribLSQ* (64 banks, 2 entries/bank, 8 slots/entry). We also assume an extra

Size	Assoc.	Ports	Conventional delay (ns)	Physical line known delay (ns)	Improvement over conventional
8KB	2 way	2	0.865	0.700	19.4%
8KB	2 way	4	1.014	0.875	13.7%
8KB	4 way	2	1.008	0.878	12.9%
8KB	4 way	4	1.307	1.266	3.1%
32KB	2 way	2	1.195	1.092	8.7%
32KB	2 way	4	1.551	1.490	4.0%
32KB	4 way	2	1.194	1.165	2.5%
32KB	4 way	4	1.693	1.693	0.0%

Table 1. Access time of conventional cache accesses and access time when the physical cache line is known for different cache configurations. The number of bytes per line is 32 in all configurations.

latency to send the addresses to the banks with respect to a conventional LSQ because it is a larger structure. We have assumed this additional delay to be equal to the delay of the buses (bitlines and wordlines) of a 128-entry structure with the same total capacity. The maximum delay of *DistribLSQ* is the delay to send an address to a bank (0.124ns) plus the delay of comparing the cache line addresses in such a bank (0.590ns). Thus, the total *DistribLSQ* delay is 0.714ns. The delays for *SharedLSQ* (8 entries, 8 slots/entry) and *AddrBuffer* (64 slots) are 0.617ns and 0.319ns respectively.

The assumed baseline LSQ (128 entries) has a delay of 0.881ns, which is 23% higher than the delay of *SAMIE-LSQ*. We also have found that a conventional LSQ with 16 entries has a delay similar (4% larger) to our *SAMIE-LSQ* configuration.

In terms of delay, we have found that those accesses to the Dcache where the physical cache line to access is known, may be done with lower delay than conventional Dcache accesses. Table 1 shows the access time of both types of accesses for different cache configurations. It can be observed that most of the configurations have lower access time when the physical cache line is known beforehand. Although in this work we do not take advantage of this lower access time for Dcache accesses, we consider that this feature of the *SAMIE-LSQ* can provide additional benefits and will be the target of future work.

4. Evaluation

This section presents performance and energy statistics for the *SAMIE-LSQ* and a baseline with a conventional fully-associative LSQ. First, the experimental framework for performance and energy modeling is presented. Then, the impact of the *SAMIE-LSQ* in performance, dynamic energy and leakage is discussed.

4.1 Experimental Framework

The *SAMIE-LSQ* performance has been evaluated with an enhanced version of `sim-outorder`, which is a microarchitecture-level performance simulator included in the Simplescalar toolset [1]. The main enhancements are the separation of the reorder buffer and the issue queue, and the modeling of ports for different structures. Energy results are derived from CACTI 3.0 [13], which is a timing, power and

Fetch, decode, commit width: *8 instructions* Issue width: *8 INT + 8FP instructions*
Branch predictor: *Hybrid 2K Gshare, 2K bimodal, 1K selector* BTB: *2048 entries, 4-way*
L1 Icache: *64KB, 2-way, 32 byte line (1 cycle)* L1 Dcache: *8KB, 4-way, 32 byte line, 4 R/W ports (2 cycles)* L2 unified cache: *512KB, 4-way, 64 byte line (10 cycles hit, 100 cycles miss, 2 cycles interchunk)* ITLB: *128 entries fully-associative (1 cycle)* DTLB: *128 entries fully-associative (1 cycle)*
Fetch queue: *64 entries* Issue queue: *128 INT + 128 FP entries* Reorder buffer: *256 entries* Load/store queue: *128 entries for the baseline*
Register file: *160 INT + 160 FP registers*
INT functional units: *6 ALU (1 cycle)* *3 mult/div (3 cycles mult, 20 cycles non-pipelined div)* FP functional units: *4 ALU (2 cycles)* *2 mult/div (4 cycles mult, 12 cycles non-pipelined div)*
Technology: *0.10 μm*

Table 2. Processor configuration.

DistribLSQ:	*64 banks* *2 entries per bank* *8 slots per entry*
SharedLSQ:	*8 entries* *8 slots per entry*
AddrBuffer:	*64 slots*

Table 3. *SAMIE-LSQ* configuration.

area model for memory-like structures. Table 2 shows the processor configuration and Table 3 shows the *SAMIE-LSQ* configuration.

For this study we have used the whole Spec2000 benchmark suite [15] with the *ref* input data set. We have simulated 100 million instructions for each benchmark after skipping the initialization part and warming up the cache for 100 million instructions. The benchmarks have been compiled with the HP/Alpha compiler with `-O4 -non_shared` flags.

4.2 Energy Model for the LSQ

The energy and area parameters used are derived from CACTI 3.0 [13]. The baseline LSQ is a conventional fully-associative structure of 128 entries. For the sake of a fair comparison, we assume for the baseline that a load address is only compared with the addresses of the older stores whose address is known. On the other hand, a store address is only compared with the addresses of the younger loads whose address is known. If there is any match, the matching loads data are forwarded from a store when it is available and the load does not access the Dcache. Table 4 details the energy consumption for the different types of accesses.

Our proposed *SAMIE-LSQ* requires comparing each address with all the addresses (entries) in-use of the corresponding bank of the *DistribLSQ* and all the addresses in-use of the *SharedLSQ*. Additionally, the age identifier (it is implemented as the reorder buffer position plus an extra bit)

LSQ	Energy
Address comparison	452 pJ + 3.53 pJ per address compared
Read/Write an address	57.1 pJ
Read/Write a datum	93.2 pJ

Table 4. Energy consumption of the different types of accesses to a 128-entry conventional LSQ.

DistribLSQ	**Energy**
Address comparison	4.33 pJ + 2.17 pJ per address compared
Read/Write an address	4.07 pJ
Age id comparison in one entry	19.4 pJ + 1.21 pJ per age id compared
Read/Write an age id	1.64 pJ
Read/Write a datum	10.9 pJ
Read/Write a TLB @ translation	6.02 pJ
Read/Write a cache line id	0.236 pJ
Bus to *DistribLSQ*	
Send an address	54.4 pJ
SharedLSQ	
Address comparison	22.7 pJ + 2.83 pJ per address compared
Read/Write an address	6.16 pJ
Age id comparison in one entry	19.4 pJ + 2.43 pJ per age id compared
Read/Write an age id	1.64 pJ
Read/Write a datum	10.9 pJ
Read/Write a TLB @ translation	8.73 pJ
Read/Write a cache line id	0.342 pJ
AddrBuffer	
Read/Write a datum	31.6 pJ
Read/Write an age id	15.7 pJ

Table 5. Energy consumption for the different activities of the *SAMIE-LSQ*.

of the instruction whose address has just been computed is compared with all the age identifiers of the slots in-use in the corresponding bank of the *DistribLSQ* and all the age identifiers of the *SharedLSQ*. This way, if it is a load, it will record the slot where there is the store that forwards its data. If it is a store, it updates the forwarding information of the loads. The energy consumption for the different activities is shown in Table 5.

The energy consumption of a Dcache access is 1009 pJ, whereas the energy consumption is 276 pJ when only one of the ways is accessed and no address is compared for a 8KB 4-way cache. For the DTLB, the energy of an access is 273 pJ.

Since CACTI does not estimate leakage, we keep track of the active area for the baseline LSQ and the *SAMIE-LSQ*, which is closely related to the leakage energy. Both mechanisms are intended to be energy efficient, so we assume that the conventional LSQ has active all in-use entries plus four extra entries for new instructions. This limitation hardly impacts the performance (less than 0.1% IPC loss) and significantly reduces leakage. On the other hand, the *SAMIE-LSQ* has active all in-use entries plus one extra entry in each bank of the *DistribLSQ* and one extra entry in the *SharedLSQ*. In each entry, the slots in-use plus an extra slot are considered to be active. The *AddrBuffer* has all in-use slots plus four extra slots active. As in the conventional LSQ, the performance degradation of these limitations is negligible. The type and area of the different cells is detailed in Table 6.

Conventional LSQ	**Type**	**Area**
Address	CAM	28 μm^2
Datum	RAM	20 μm^2
DistribLSQ		
Address	CAM	10 μm^2
Age id	CAM	10 μm^2
Datum	RAM	6 μm^2
TLB address translation	RAM	6 μm^2
Cache line id	RAM	6 μm^2
SharedLSQ		
Address	CAM	10 μm^2
Age id	CAM	10 μm^2
Datum	RAM	6 μm^2
TLB address translation	RAM	6 μm^2
Cache line id	RAM	6 μm^2
AddrBuffer		
Datum	RAM	20 μm^2
Age id	RAM	20 μm^2

Table 6. Area of the different components of the conventional LSQ and *SAMIE-LSQ*.

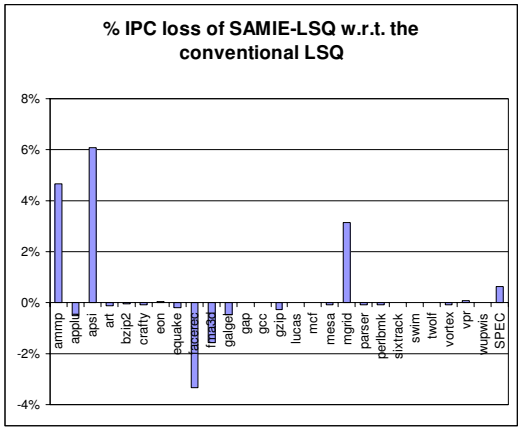

Figure 5. % IPC loss of *SAMIE-LSQ* with respect to the 128-entry conventional LSQ.

4.3 Performance

Figure 5 presents the IPC of the *SAMIE-LSQ* with respect to the conventional LSQ. We observe that *SAMIE-LSQ* loses some performance for `ammp`, `apsi` and `mgrid`. As shown in Figure 3, these programs would require a large number of *SharedLSQ* entries. Thus, the *SharedLSQ* often becomes full and some instructions have to wait in the *AddrBuffer*, which implies than some instructions that are ready to execute have to wait for an available entry/slot in the proper bank of the *DistribLSQ* or the *SharedLSQ*. Furthermore, since some instructions have to wait in the *AddrBuffer*, it may happen that the oldest memory instruction cannot be placed neither in the *DistribLSQ* or the *SharedLSQ*, firing the deadlock avoidance scheme (i.e. pipeline flush) described above. Figure 6 shows the number of deadlocks per million of cycles. It can be seen that `ammp` is the only program with a significant number of deadlocks.

In Figure 5 we also observe that some programs such as `facerec` and `fma3d` perform better with the *SAMIE-LSQ* than with the conventional LSQ. This is so because these

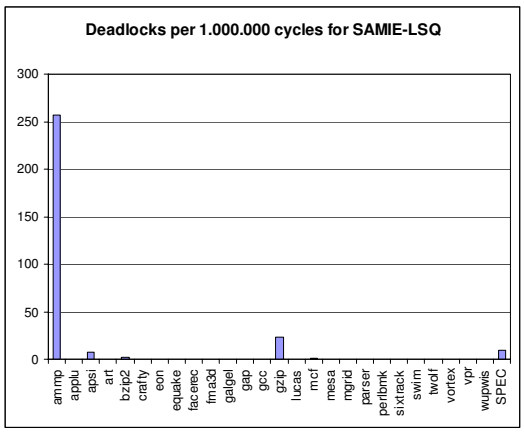

Figure 6. Number of deadlock-avoidance pipeline flushes per million of cycles for *SAMIE-LSQ*.

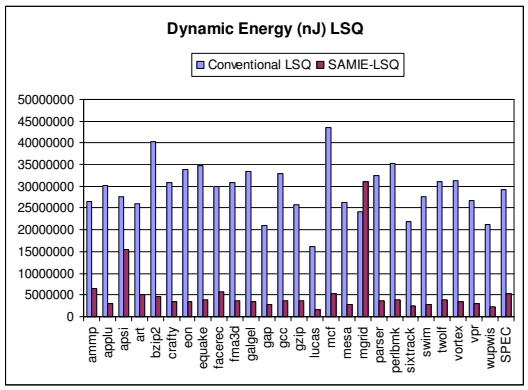

Figure 7. Dynamic energy consumption for the LSQ.

programs have high LSQ pressure and the conventional LSQ can hold up to 128 memory instructions, whereas the *SAMIE-LSQ* can hold many more if they are well distributed among the different banks.

On average, the *SAMIE-LSQ* loses 0.6% IPC with respect to the conventional LSQ. This does not take into account the potential benefits from the fact that the delay of the *SAMIE-LSQ* is lower than that of the conventional LSQ, as shown in section 3.

4.4 Dynamic Energy

Figure 7 shows the dynamic energy consumption of the conventional LSQ and the *SAMIE-LSQ*. We observe that the *SAMIE-LSQ* is much more energy-efficient than the conventional LSQ for all but one program. In fact, the programs that have high energy consumption with the *SAMIE-LSQ* are those with high *SharedLSQ* requirements. This trend can be seen in Figure 8 where the energy of the *SAMIE-LSQ* is broken down. Most of the programs spend the energy in the *DistribLSQ* and the buses, but `ammp`, `apsi`, `facerec` and `mgrid` have significant number of conflicts and require large space in the *SharedLSQ* and the *AddrBuffer*.

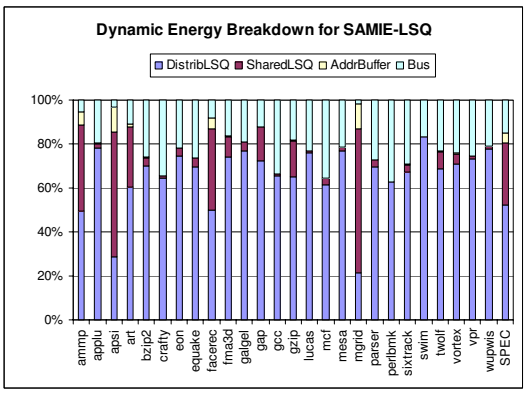

Figure 8. Dynamic energy consumption breakdown for the *SAMIE-LSQ*.

On average, the *SAMIE-LSQ* saves 82% of the dynamic energy of the conventional LSQ with negligible performance degradation.

As stated in section 3, the *SAMIE-LSQ* enables significant energy savings in the L1 data cache and the data TLB by caching the location of the data in cache and the address translation respectively. Figure 9 shows the energy consumption of the Dcache for both the conventional LSQ and the *SAMIE-LSQ*. It can be seen that the energy savings for the *SAMIE-LSQ* are consistent across all benchmarks. On average, 42% of the L1 data cache energy can be saved, `ammp` and `swim` being the programs with highest savings (58%), and `sixtrack` being the program with lowest energy savings (21%).

Figure 10 shows the data TLB energy consumption. In general, those Dcache accesses that do not compare the address and only access one way, do not access the DTLB because the address translation has also been cached. Thus, the fraction of energy savings for the DTLB is higher than that for the Dcache. On average, 73% DTLB energy is saved if we compare the *SAMIE-LSQ* with a conventional LSQ. The highest savings correspond to `ammp` (84%) and the lowest to `mcf` (55%).

4.5 Leakage

The *SAMIE-LSQ* is larger than the conventional LSQ because it has practically the same number of addresses but space for 8 instructions per address, whereas the conventional LSQ only has space for one instruction per address. Nevertheless, the *SAMIE-LSQ* can work with an active area similar to that of the conventional LSQ as shown in Figure 11. We accumulate the area every cycle instead of using the average area to take into account the longer or shorter execution time of the different programs. The accumulated active area for both the conventional LSQ and the *SAMIE-LSQ* are very similar, and slightly favorable to the *SAMIE-LSQ* (5%). The best scheme in terms of active area depends on the program, some integer programs (`bzip2`, `crafty`, `gcc`, `parser`, `perlbmk`) being the worst programs for

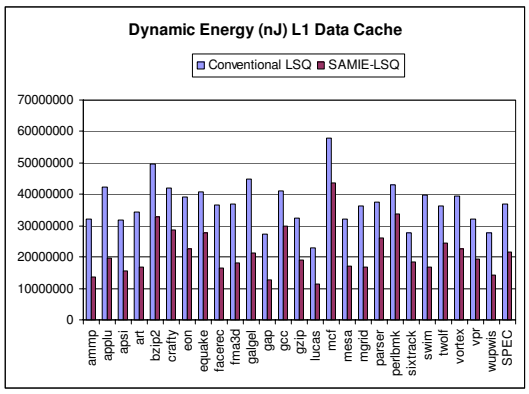

Figure 9. Dynamic energy consumption for the L1 data cache.

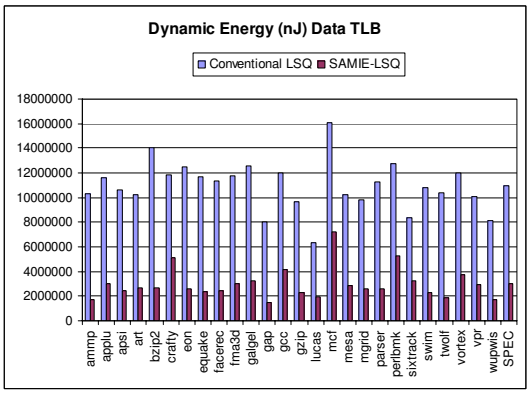

Figure 10. Dynamic energy consumption for the data TLB.

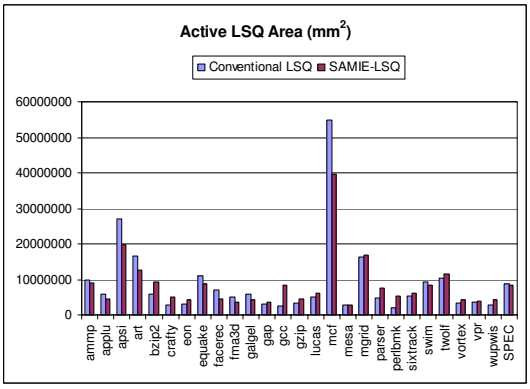

Figure 11. Accumulated active area in mm^2 for the LSQ.

SAMIE-LSQ because they have very low LSQ space requirements and the *SAMIE-LSQ* keeps larger empty area active than the conventional LSQ.

Figure 12 shows the area breakdown for *SAMIE-LSQ*. The *DistribLSQ* is the structure with the largest active area, and the *SharedLSQ* active area is noticeable only in those

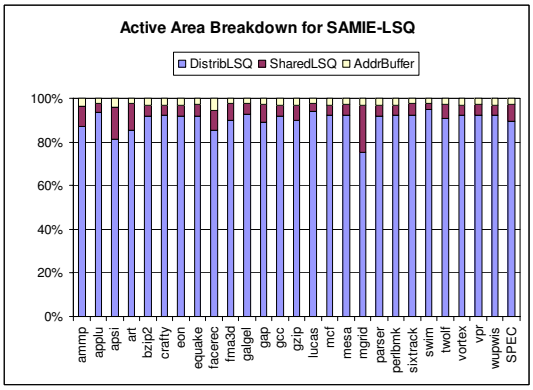

Figure 12. Active area breakdown for the *SAMIE-LSQ*.

programs with high *SharedLSQ* space requirements (`ammp`, `apsi`, `art`, `facerec`, `mgrid`).

5. Conclusions and Future Work

We have presented the *SAMIE-LSQ*, which is a new power-aware load/store queue design. The *SAMIE-LSQ* exploits the fact that many in-flight loads and stores access the same cache line and places these instructions in the same entry. This reduces the required number of address comparisons and other activity in the data cache and the TLB. This number of comparisons is further reduced by using a set-associative organization instead of a fully-associative one.

The *SAMIE-LSQ* saves 82% dynamic energy for the load/store queue, 42% for the L1 data cache and 73% for the data TLB, with a negligible impact on performance (0.6%).

Additionally, the delay of the *SAMIE-LSQ* is lower than that of a conventional load/store queue, and the access time for many L1 data cache references is also reduced. This enables further opportunities for optimizations to improve the performance and/or energy requirements, which have not been exploited in this work and will be the target of our future research. Another interesting future research direction is the coupling of the *SAMIE-LSQ* with the L1 data cache by integrating the *DistribLSQ* entries and their corresponding cache set(s) in the same physical structure to further reduce the cache access time.

Acknowledgements

This work has been partially supported by the Ministry of Education and Science under grants AP2002-3677, TIN2004-07739-C02-01 and TIN2004-03072, the CICYT project TIC2001-0995-C02-01, Feder funds, and Intel Corporation. We would like to thank the anonymous reviewers by their comments.

References

[1] D. Burger, T. Austin. The SimpleScalar Tool Set, Version 3.0. Technical report, Computer Sciences Department, University of Wisconsin-Madison, 1999.

[2] H. Cain, M. Lipasti. Memory Ordering: A Value Based Definition. In proceedings of the 31st International Symposium on Computer Architecture (ISCA'04), München (Germany), June 2004.

[3] G. Chrysos, J. Emer. Memory Dependence Prediction using Store Sets. In proceedings of the 25th International Symposium on Computer Architecture (ISCA'98), Barcelona (Spain), June 1998.

[4] M. Franklin, G.S. Sohi. ARB: A Hardware Mechanism for Dynamic Reordering of Memory References. In IEEE Transactions on Computers, volume 45, issue 5, May 1996.

[5] K. Gharachorloo, A. Gupta, J. Hennessy. Two Techniques to Enhance the Performance of Memory Consistency Models. In proceedings of the International Conference on Parallel Processing (ICPP'91), Austin (Texas), August 1991.

[6] A. Moshovos, S. Breach, T.N. Vijaykumar, G.S. Sohi. Dynamic Speculation and Synchronization of Data Dependences. In proceedings of the 24th International Symposium on Computer Architecture (ISCA'97), Denver (Colorado), June 1997.

[7] D. Nicolaescu, A. Veidenbaum, A. Nicolau. Reducing Data Cache Energy Consumption via Cached Load/Store Queue. In proceedings of the 9th International Symposium on Low Power Electronics and Design (ISLPED'03), Seoul (Corea), August 2003.

[8] S. Onder, R. Gupta. Dynamic Memory Disambiguation in the Presence of Out-of-order Store Issuing. In proceedings of the 32nd International Symposium on Microarchitecture (MICRO'99), Haifa (Israel), November 1999.

[9] I. Park, C.L. Ooi, T.N. Vijaykumar. Reducing Design Complexity of the Load/Store Queue. In proceedings of the 36th International Symposium on Microarchitecture (MICRO'03), San Diego (California), December 2003.

[10] A. Roth. A High Bandwidth Low Latency Load/Store Unit for Single and Multi- Threaded Processors. Technical Report MS-CIS-04-09, University of Pennsylvania, August 2004.

[11] A. Roth. Store Vulnerability Window (SVW): Re-Execution Filtering for Enhanced Load/Store Optimization. Technical Report MS-CIS-04-29, University of Pennsylvania, December 2004.

[12] S. Sethumadhavan, R. Desikan, D. Burger, C.R. Moore, S.W. Keckler. Scalable Hardware Memory Disambiguation for High ILP Processors. In proceedings of the 36th International Symposium on Microarchitecture (MICRO'03), San Diego (California), December 2003.

[13] P. Shivakumar and N.P. Jouppi. CACTI 3.0: An Integrated Cache Timing, Power and Area Model. Research report 2001/2, WRL, Palo Alto, CA (USA), 2001.

[14] A. Yoaz, M. Erez, R. Ronen, S. Jourdan. Speculation Techniques for Improving Load-Related Instruction Scheduling. In proceedings of the 26th International Symposium on Computer Architecture (ISCA'99), Atlanta (Georgia), May 1999.

[15] SPEC 2000 http://www.specbench.org/osg/cpu2000/