Reducing the LSQ and L1 Data Cache Power Consumption

R. Apolloni[1], P. Carazo[2], F. Castro[3], D. Chaver[3], L. Pinuel[3], and F. Tirado[3]

[1] Universidad Nacional de San Luis, Argentina
[2] Universidad Politecnica de Madrid, Spain
[3] Universidad Complutense de Madrid, Spain

Abstract. In most modern processor designs, the HW dedicated to store data and instructions (memory hierarchy) has become a major consumer of power. In order to reduce this power consumption, we propose in this paper two techniques, one to filter accesses to the LSQ (Load-Store Queue) based on both timing and address information, and the other to filter accesses to the first level data cache based on a forwarding predictor. Our simulation results show that the power consumption decreases in 30-40% in each structure, with a negligible performance penalty of less than 0.1%.

1 Introduction

Power dissipation in an out of order microprocessor is spread across different structures including Caches, Register Files, the Branch Predictor, the Load-Store Queue, etc. Specifically, the HW dedicated to store data and instructions (the LSQ, the different cache levels, and the main memory) consumes a significant part of the overall power.

In this paper we intend to reduce the LSQ and L1 data cache (DL1) power consumption in an out of order processor. It can be argued that this research problem is not a major concern now due to the trend towards multi-core architectures made by the industry, in which in some cases the pipelines employed are simpler. However homogeneous multi-manycore architectures with in-order pipelines will only provide substantial benefits for scalable applications/workloads, and some researchers have recently highlighted that future designs will benefit from asymmetric architectures that combine simple and power-efficient cores with a few complex and power-hungry cores [1]. The local inefficiencies of a complex core can translate into global performance/per-watt improvements since a complex core could accelerate the serial phases of applications when the power-efficient cores are idle. This way, a single chip will be able to provide good scalability for parallel applications as well as ensure high serial performance. In summary, as promoted in [2], researchers should still investigate methods of improving sequential performance despite we have entered into the multicore era.

We propose two separate techniques for reducing the LSQ and the DL1 power consumption: For the first, based on a set of registers and a small table, we filter

many of the required associative accesses that loads and stores would have to perform to the LSQ. For the second, based on a small forwarding predictor, we filter many of the load accesses to DL1. Our techniques lead to high power savings in those structures, which translate into important power savings on the whole processor.

The rest of the paper is organized as follows. Section 2 recaps related work. Section 3 reviews the conventional implementation and brings in our new mechanisms. Section 4 details our experimental environment, while Section 5 outlines experimental results and analyses. Finally, Section 6 concludes.

2 Background

In the last years, there has been a lot of research focused on reducing the LSQ and the Cache power consumption. Next, we summarize the proposals directly related to the mechanisms developed in our job.

Concerning the LSQ power reduction, Sethumadhavan et al. [3] propose an address-based filtering scheme named *search filtering*, which uses hashing to reduce the number of lookups to the LSQ. For this purpose two Bloom Filters [4] are employed, and based only in address information they are able to significantly reduce the associative searches needed while maintaining the program semantics. On the other hand, in our previous work from [5], we introduced two timing-based filtering mechanisms for the LSQ that avoid many of the associative searches that a conventional processor performs unconditionally. The design is based on two sets of age registers: one that filters lookups to the LQ (Load Queue), and another that carries out the same operation over the SQ (Store Queue). Upon execution, just based on straightforward age comparisons between memory instructions and the corresponding register, the mechanism is able to deliver high filtering efficiency with a negligible hardware cost.

Concerning the DL1 power reduction, Nicolaescu *et.al* [6] propose to avoid the data cache access for those loads that receive their data through forwarding. To increase forwarding, they modify the LSQ design to retain load and store instructions after their commit. Thereby, a later load increases its chances of receiving its data from a previous instruction, either an in-flight store, a commited store, or a commited load. The mechanism –named *cached load store queue*, *CLSQ*– is based on the low observed rates of LSQ occupancy for some program phases, that make it possible to earmark unoccupied entries to already commited load or store instructions.

3 Hybrid LSQ Filtering Mechanism

We present here a full LSQ filtering HW, built upon our proposal from [5] and Sethumadhavan's proposal from [3]. In both papers, the main idea is to add simple HW capable of ruling out some memory ordering violations and store to load forwardings. For that purpose, in the first scheme timing information was mainly used, while the second one employed just address information of memory

instructions. We combine both in a new hybrid design that provides more filtering capability. Besides, we test our mechanism in a different microarchitectural model -an x86 architecture- than that of the prior works. This model, besides of resulting more appealing, enables for new types of filtering which lead to extra power savings. We should highlight that it includes two new characteristics that are very important for our job [4]: (1) Each store accesses the SQ at issue time in order to perform store-store forwarding; (2) It provides a dependence predictor (LSAP), consisting on an associative table accessed by each load at issue time, that predicts whether a load will alias with a previous store.

3.1 LQ Filtering

Regarding stores searching the LQ looking for premature loads, we can take advantage of a set of registers that basically contain information about loads' age (timing information) and include implicitly some information about loads' address. This approach is referred as Multi-YLA (Multi - Youngest issued Load Age) [5]. The multi-YLA by itself provides really good results in terms of filtering capability, and according to our experiments, combining this scheme with the Bloom Filter from [3] reports no significant improvements.

3.2 SQ and Predictor Filtering

Regarding loads searching the SQ looking for previous dependent stores, and similarly to the Multi-YLA, in [5] we proposed to add a set of registers that contain information about stores' age. This approach is referred as Multi-OFS (Multi - Oldest in Flight Store) [5]. On the other hand, in [3] the authors proposed to add two Bloom Filters (BF) that contain address information (instead of timing information). Due to the very complex updating process of the OFSs set in the multi-OFS scheme its usage is inappropriate. On the other hand, using only a Bloom Filter loses the opportunity of filtering more accesses based on timing information. Hence, we propose to combine a single OFS, that holds timing information and requires a much simpler updating process than a Multi-OFS, and a BF, which provides address-based information. Besides, we include an additional register, called PAS (Pending Address Stores), to know if all in flight stores in the processor are resolved. In the next subsections we describe in detail our proposed mechanism.

Filtering of Loads accessing the SQ When a load instruction issues, it consults the OFS (which holds the age of the oldest in-flight store in the processor) and checks the PAS. If the load is older than OFS (i.e. older than the oldest in-flight store), we can assure, based just on timing information, that a store to load forwarding is not needed for that load, and both the BF access and the SQ associative search can be avoided. Otherwise, the load goes to the second stage

[4] More details of the LSQ management in this architecture can be found in [7].

of our mechanism, the Bloom Filter (a hashing table with one entry per group of addresses that holds the number of in-flight stores to those addresses). If the corresponding BF entry and the PAS register are both zero [5], or if the BF entry is zero and the LSAP does not predict a dependence with a previous store, we can guarantee, based now on address information, that the load can not receive its data via a forwarding, and again the SQ scan can be avoided. If the BF entry is zero, but PAS is bigger than zero and the LSAP predicts a dependence, an SQ search must be performed to find the closest unresolved store. However, this is a simpler and cheaper access than a common associative one, since we do not need to compare addresses. Finally, if the value recorded in the corresponding BF entry is bigger than zero, the normal SQ associative access is carried out.

Filtering of Loads accessing the Predictor The new architecture we are using allows for new filtering opportunities. One of them is the chance to filter some accesses of load instructions to the LSAP. The first opportunity to avoid such lookups happens when a load checks the OFS: If the load is older it means that no previous stores exist; hence the LSAP information is irrelevant and the predictor search turns unnecessary.

In order to increase the LSAP filtering capability, we can take advantage of the PAS register: If it is zero, we can also aviod the LSAP access, since all stores are resolved and therefore there is nothing to predict. Otherwise, the LSAP search is required.

Filtering of Stores accessing the SQ Recall that in the new architecture each store checks the SQ in order to find previous stores to the same address. Once again, we can filter some of these searches. When a store instruction issues, it compares its age with the OFS value. If they are equal, we can guarantee that no prior stores exist, and therefore both the BF access and the SQ lookup can be avoided. Otherwise, the store consults the BF, hashing its address. If the corresponding entry and PAS [6] are zero, we can assure, based now on address information, that this is the only one store to that address, and the SQ search can be avoided. If the BF entry is zero but PAS is bigger than zero, we have to perform an SQ lookup to find the closest unresolved store. Again, this is a simpler and cheaper access than a normal associative search. Finally, if the entry is bigger than zero, a normal SQ lookup is carried out.

This proposed hybrid mechanism exhibits several advantages compared to the multi-OFS [5] and the Bloom Filter [3] schemes. First, thanks to the combination

[5] PAS being zero means that we do not need to pay attention to LSAP since no unresolved stores exist.

[6] Note that we use the PAS register in combination with the BF for being able to filter some of the SQ accesses. The BF provides information about resolved stores, while the PAS register reports information about unresolved ones. As we mentioned before, without this register the BF can not be trusted in this context.

of timing and address information, the number of filterings grows significantly, as we will demonstrate in the evaluation section. Second, unlike the multi-OFS scheme, the updating process for our single-OFS is very simple and incurs no power cost: when a store instruction commits, the OFS is just updated with the age of the store accommodated in the contiguous SQ entry. Third, in our approach the Bloom Filter access is avoided when the 1^{st} stage filters the SQ search based on the OFS (note that the OFS access is much cheaper than the BF one). On the contrary, in the BF scheme, the filter is always accessed at load/store issue.

4 DL1 Filtering using a Forwarding Predictor

4.1 Rationale

In most conventional microprocessors each load instruction consults the first level data cache in order to move the required data into an available register. In parallel, the Store-Queue is searched looking for a previous matching in-flight store. If it is found, the store forwards the corresponding data. Otherwise, the data is provided by the cache. The technique that we propose in this paper is based on the observation that if a load gets its data directly from an earlier store, the data cache access becomes completely unnecessary, and hence we could avoid it saving some power. Obviously, this is only useful if the percentage of loads that get the data from the SQ is high enough.

In a RISC processor, the amount of architectural registers is commonly set to 32 and a register-register architecture is generally implemented. With such configuration, the number of store to load forwardings is relatively small (for example, in [8], less than 15% on average), and maybe the benefits of trying to avoid the DL1 access in such reduced occasions could turn meaningless. However, in a register-memory architecture with only 16 architectural registers –as in the case of x86-64, the architecture employed in this job– the number of store to load forwardings is higher as a result of the extra operations due to register spilling.

In a complementary way, we can use Nicolaescu's CLSQ from [6], which significantly increases the number of loads that receive their data via forwarding, both due to store-load forwarding from the Cached-SQ and to load-load forwarding from the Cached-LQ.

In summary, on a x86-64 architecture using Nicolaescu's Cached-LSQ, the number of forwardings can be relatively high – up to 40% of the loads –, which makes our initial intuition appealling. However, in order to be able to filter out these accesses, we need to either serialize the LSQ and DL1 cache searches, or know in advance –i.e. make a prediction– whether the load will receive the data via forwarding or not. This is a key issue that has to be addressed.

4.2 Overall structure

As we have just mentioned, an obvious implementation would be to serialize the accesses (as Nicolaescu in [6]): the load first scans the SQ, and then –only when

neccessary– the cache is accessed. However, this design is not effcient: when a previous matching store is not found the delay incurred in accessing to the data cache will result in a significant slowdown. In this paper we will turn up with a much more convenient approach.

The design that we propose is based on a forwarding predictor: for each load, we predict whether it will receive its data through forwarding. For convenience of discussion, we loosely refer to these loads as *predicted-dependent* loads and the remainder *predicted-independent* loads. For predicted-dependent loads, only the SQ and the cached-LQ are accessed, omitting the DL1 access (of course, at the risk of being wrong, in which case the cache access is launched with a delay of 1 cycle). For the remaining, both the SQ, the cached-LQ and the DL1 are accessed in parallel (note that in this case, if the predictor is wrong, the data cache access is unnecessary). A predictor with high accuracy provides significant power savings at the cost of a tiny performance degradation. This idea has been explored in similar, yet different contexts [9].

There is a whole lot of research in the field of memory dependence prediction. However, they all employ sophisticated predictor structures, which are excessive for our goal of predicting in advance if a load will receive its data through forwarding. For this reason, we have not considered them in our job. Instead, we have evaluated two kinds of simple predictors: Bloom Filter based [4] and Branch Predictor based [10].

Bloom Filter based predictor In this first kind of predictors, we implement a low-overhead hash table of counters: At issue time, every load and store hash their memory addresses to a single entry and increment the corresponding counter. Then, at commit, the entry is decremented. Besides, at issue time, loads read the counter before it was incremented to perform the prediction. If it is bigger than zero, there is a likely (but not certain) address match with another memory instruction, and the load is predicted to receive its data via a forwarding. On the other hand, if the counter is zero, the load is predicted-independent.

Branch Predictor based The second kind of predictors is based on the well-known bimodal branch predictor. Similarly to branch instructions, the majority of loads are usually strongly biased, so such a predictor works well. An advantage of this Bimodal Predictor versus the Bloom Filter based is that the prediction can be performed as soon as the load instruction is decoded, based on its PC. On the contrary, a Bloom Filter is consulted with the memory address of the load, that needs to be calculated first, so the prediction is delayed to issue phase in this case.

Combined Predictor Finally, we should mention that we have also considered in our evaluation a combined predictor, merging a Bloom Filter with a Bimodal predictor. For extracting the final decision, we predict that a load will receive its data through forwarding only when both structures predict the load to be

dependent. Such a structure benefits from both the past forwarding information of loads and memory address information, giving the best results as we will show in the Evaluation Section.

5 Experimental Framework

We have evaluated our proposed design using the PTLsim [11], a performance-oriented simulation tool. The microarchitecture models the default PTLsim configuration that results from the merging of different features of an Intel Pentium 4 [12], an AMD K8 and an Intel Core 2 [13]. Some of the main simulation parameters are listed in Table 1.

Table 1. Simulation parameters for default PTLSim configuration

Branch predictor	Combined (Bim-2bits + Gshare), 2K BTAC
Instruction Fetch queue size	32
ROB size	128
LSQ size	80 (LQ: 48, SQ: 32)
LSAP size	16
Physical Registers	256
Fuctional Units (INT)	8: 4 ALU (2 INT, 2 FP), 2 Load, 2 Store
Fetch/Decode/Issue/Commit width	4/4/4/4
L1 Instruction Cache	32KB (4 way, 64B line)
L1 Data Cache	16KB (4 way, 64B line, 2 cycles latency)
L2 Data Cache	256KB (16 way, 64B line, 6 cycles latency)
L3 Data Cache	4MB (32 way, 64B line, 16 cycles latency)
Main memory latency	140 cycles

The evaluation of our proposal has been performed using 23 benchmarks from the SPEC CPU2006 suite, compiled for the x86 instruction set. We simulate regions of 100M instructions after reaching a triggering point [14], that marks the beginning of code area in which the application behavior is representative of the overall execution. To evaluate the impact of our designs over the power consumption of the LSQ or the DL1, we use a power model developed in [14] and CACTI 5.3 [15].

6 Evaluation

6.1 LSQ Power Savings

In this section we report results for the power savings obtained in the LSQ, according with the Power Model explained in [14]. We show 3 different schemes: our proposed scheme (that includes a Multi-YLA for LQ filtering and our Hybrid scheme for SQ and LSAP filtering), the proposal from [3] (that includes two Bloom Filters, one for LQ filtering and the other for SQ filtering), and the proposal from [5] (that includes a Multi-YLA for LQ filtering and a Multi-OFS for SQ filtering). Note that none of these schemes affects performance, since they just care about filtering unnecessary accesses.

Figure 1 compares the three approaches for a similar extra HW amount. It shows the average over the studied 23 SPEC-06 applications. Clearly, our scheme reports the highest dynamic power savings. For example, with 1024 bits, the 2 Bloom filters save around 25%, the Multi-YLA + Multi-OFS, 22%, and our design, 38% of the dynamic power consumption of a conventional LSQ. Note that we are including in the LSQ power consumption the LSAP power cost –of course, apart from LQ/SQ power waste–.

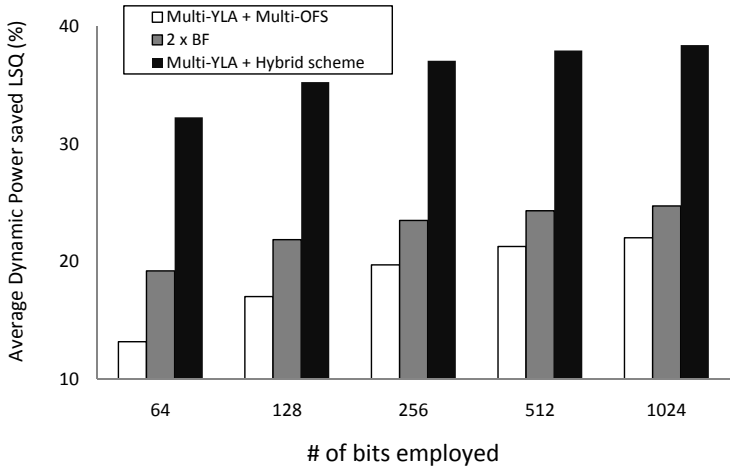
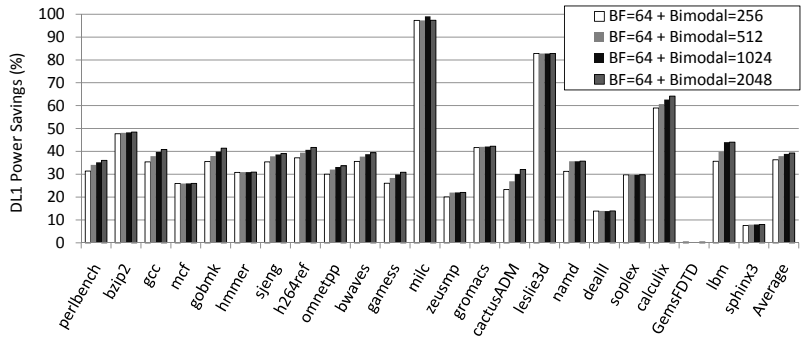


Fig. 1. Average dynamic power saved over the conventional LSQ.

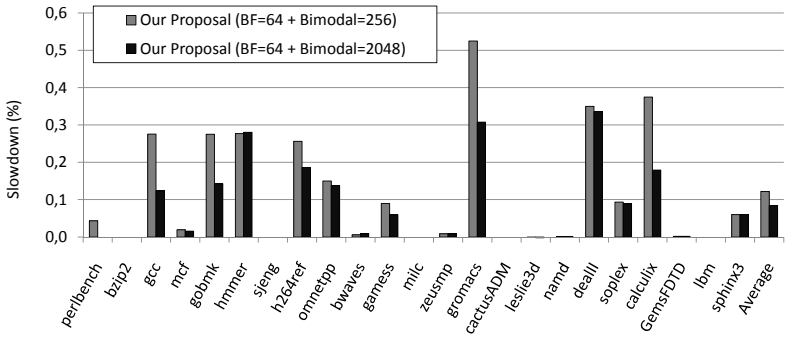
6.2 DL1 Power Savings

In this section we show the data cache power reduction and the whole system performance using either the baseline or our alternative. Figure 2(a) shows the power savings achieved in the data cache in our technique with respect to the original architecture. Figure 2(b) illustrates the performance impact of our proposal with respect to the original architecture. In these experiments we always employ the combined predictor, since it reports the highest accuracy values (in [16] we show a comparison of the different forwarding predictors). We can extract the following conclusions.

First, by including our proposed scheme, a significant fraction of loads are correctly predicted-dependent, and therefore the corresponding data cache accesses avoided. This leads to a significant fraction of the DL1 dynamic power consumption eliminated, as Figure 2(a) shows. On average, for a Bloom Filter with 64 entries and a Bimodal Predictor of 256, the DL1 power savings of our approach are around 36%.



(a)



(b)

Fig. 2. (a) DL1 Power Savings. **(b)** Performance Impact.

Second, and more important, in our architecture average performance remains almost untouched (around 0.1% of slowdown), something that would not happen with Nicolaescu's Proposal. The reason is that in his case, when a load finds no previous dependent stores in the LSQ (i.e. experiments no forwarding) incurs a delay of 1 cycle accessing the DL1, while in our case the forwarding predictor avoids this to happen by predicting most of these loads as independent.

7 Conclusions

The main contributions of this paper are:

– We implement a hybrid design, based on two previous proposals [5] and [3], that filters most of the irrelevant searches to the LSQ. The extra HW involved is almost negligible and the technique carries no performance degradation at all. On average, our technique saves up to 39% of the LSQ dynamic power consumption.

– We implement a mechanism that filters many accesses to the first level data cache based on a forwarding predictor and Nicolaescu's CLSQ [6]. Both the extra HW and the performance degradation are negligible. On average, our mechanism saves up to 36% of the DL1 dynamic power, with a HW cost less than 100B and a slowdown less than 0.1%.
– All these schemes were tested in a different and more common microarchitectural model -the widespread x86-64- than the one used in previous works.

References

1. Bower, F., Sorin, D., Cox, L.: The impact of dynamically heterogeneous multicore processors on thread scheduling. Micro, IEEE **28**(3) (May-June 2008) 17–25
2. Hill, M.D., Marty, M.R.: Amdahl's law in the multicore era. IEEE Computer **41**(7) (2008) 33–38
3. Sethumadhavan, S., Desikan, R., Burger, D., Moore, C., Keckler, S.: Scalable Hardware Memory Disambiguation for High ILP Procs. In: MICRO'03. 399–410
4. Bloom, B.: Space/Time Trade-offs in Hash Coding with Allowable Errors. Communic. of the ACM **13**(7) (1970) 422–426
5. Castro, F., Chaver, D., Pinuel, L., Prieto, M., Tirado, F.: Using Age Registers for a simple Load Store Queue Filtering. Journal of Systems Architecture **55**(2) (February 2009) 79–89
6. Nicolaescu, D., Veidenbaum, A., Nicolau, A.: Reducing Data Cache Energy Consumption via Cached Load/Store Queue. In: ISLPED'03. 252–257
7. Yourst, M.: PTLsim Users Guide and Reference: The Anatomy of an x86-64 Out of Order Superscalar Microprocessor, http://www.ptlsim.org/documentation.php. (2007)
8. Castro, F., Chaver, D., Pinuel, L., Prieto, M., Huang, M., Tirado, F.: A Load-Store Queue Design based on Predictive State Filtering. Journal of Low Power Electronics **2**(1) (April 2006) 27–36
9. Sha, T., Martin, M., Roth, A.: Scalable Store-Load Forwarding via Store Queue Index Prediction. In: MICRO'05. 159–170
10. McFarling, S.: Combining Branch Predictors. Technical report tn-36, Western Research Laboratory, Digital Equipment Corporation (June 1993)
11. Yourst, M.T.: PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. In: ISPASS'07. 23–34
12. Hinton, G., Sager, D., Upton, M., Boggs, D., Carmean, D., Kyker, A., Roussel, P.: The Microarchitecture of the Pentium 4 Proc. Intel Technology Journal (Q1 2001)
13. Copenhagen Univ. College of Eng.: The Microarch. of Intel and AMD CPU's: an Optimization Guide for Assembly Programmers and Compiler Makers. (2009)
14. Apolloni, R., Chaver, D., Castro, F., Pinuel, L., Prieto, M., Tirado, F.: A hybrid timing-address oriented LSQ filtering for an x86 architecture. Accepted for publication on journal IET-Computers and Digital Techniques (06-08-2010).
15. http://www.hpl.hp.com/research/cacti/
16. Carazo, P., Apolloni, R., Castro, F., Chaver, D., Pinuel, L., Tirado, F.: L1 Data Cache Power Reduction using a Forwarding Predictor. Accepted for publication on international conference PATMOS-2010.