L1 Data Cache Power Reduction Using a Forwarding Predictor

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

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

Abstract. In most modern processor designs the L1 data cache has become a major consumer of power due to its increasing size and high frequency access rate. In order to reduce this power consumption, we propose in this paper a straightforward filtering technique. The mechanism is based on a highly accurate forwarding predictor that determines if a load instruction will take its corresponding data via forwarding from the load-store structure –thus avoiding the data cache access– or it should catch it from the data cache. Our simulation results show that 36% data cache power savings can be achieved on average, with a negligible performance penalty of 0.1%.

1 Introduction

Power dissipation in an out of order microprocessor is spread across different structures including caches, register files, the branch predictor, etc. Specifically, on-chip caches consume a significant part of the overall power by themselves. In this paper we intend to reduce the 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. Furthermore if several out-of-order cores are employed –either in an asymetric or an homogeneous multi-core design– our technique can be applied to each private DL1 cache, leading to a higher benefit.

R. van Leuken and G. Sicard (Eds.): PATMOS 2010, LNCS 6448, pp. 116–125, 2011.

The mechanism that we propose in this paper for reducing the DL1 power consumption is based on an effcient usage of the LSQ (load-store queue), a structure responsible of keeping all in flight memory instructions and detecting and enforcing memory dependences in an out of order processor. One of the main LSQ tasks is to supply the correct data to load instructions via a forwarding process –store to load forwarding– ruling out the cache data and therefore turning the cache access unnecessary. Taking advantage of Nicolaescu's CLSQ [3], in which the number of loads that receive their data from a previous store increases a lot, and using an accurate forwarding predictor, that suggests if a load instruction is likely to receive its data through forwarding, we manage to reduce significantly the amount of accesses to data cache in an x86 architecture. The small misprediction rate obtained translates into an IPC that remains largely unchanged.

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 mechanism. Section 4 details our experimental environment, while Section 5 outlines experimental results and analyses. Finally, Section 6 concludes.

2 Background

Many techniques for reducing the cache energy consumption have been explored recently. Next, we recap some of the more outstanding ones.

One alternative is to partition caches into several smaller caches [4] with the corresponding reduction in both access time and power cost per access. Another design, known as filter cache [5], trades performance for power consumption by filtering cache references through an unusually small L1 cache. An L2 cache, which is similar in size and structure to a typical L1 cache, is placed after the filter cache to minimize the performance loss. A different alternative, named selective cache ways [6], provides the ability to disable a subset of the ways in a set associative cache during periods of modest cache activity, whereas the full cache will be operational for more cache-intensive periods. Another different approach takes advantage of the special behavior in memory references: we can replace the conventional unified data cache with multiple specialized caches. Each one handles different kinds of memory references according to their particular locality characteristics [7]. These alternatives make it possible to improve in terms of performance or power efficiency. Finally, Jin *et.al* [8] obtain power savings in L1 cache by exploiting loads spatial locality. In their technique, loads always bring a macro data from the processor cache, allowing additional opportunities for load to load forwarding.

Nicolaescu *et.al* [3] propose to avoid the data cache access for those loads that receive their data through forwarding. To increase them, 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. Our work extends and improves this job.

Finally, as we are using a forwarding predictor in our design, we should mention that there are many proposals relying on memory dependence prediction, that propose techniques to know in advance which pairs of store-load instructions will depend and take appropriate actions [9] [10]. However, they all are overprovisioned for the goal of our job.

3 Filtering DL1 Accesses Using a Forwarding Predictor

3.1 Rationale

In most conventional microprocessors each load instruction consults the first level data cache (DL1) in order to move the required data into an available register. In parallel, the Store-Queue (SQ) 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 (Figure 1, Original Architecture). 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 [11], 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 [3], 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.

3.2 Overall Structure

As we have just mentioned, an obvious implementation would be to serialize the accesses (as Nicolaescu in [3]): the load first scans the SQ, and then –only when

neccessary– the cache is accessed (Figure 1, Nicolaescu's Proposal). 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 (Figure 1, Proposed Architecture) 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 [12].

There is a whole lot of research in the field of memory dependence prediction (Section 2). 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 [13] and Branch Predictor based [14].

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[1].

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

[1] As explained in [15], the SQ and LQ accesses could be avoided in this case. However, since a DL1 cache access is much more power consuming than an LQ-SQ access, in this paper we do not consider such LQ or SQ filtering capability, that would require a deeper study.

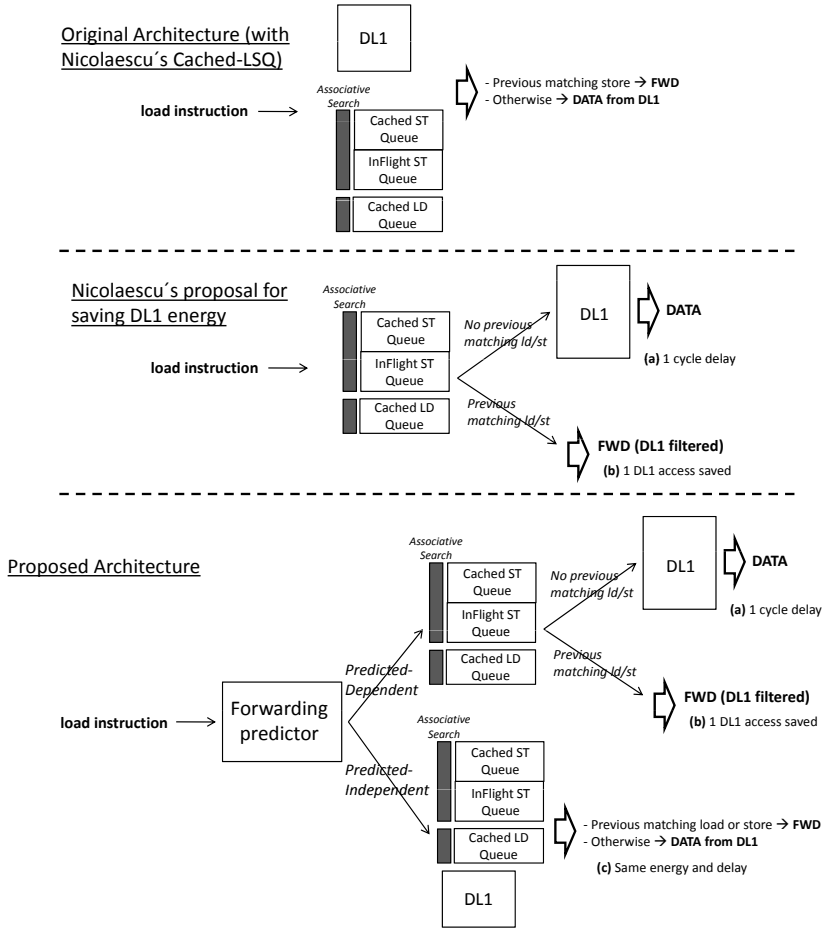

Fig. 1. Original Architecture (with the Cached-LSQ), Nicolaescu's Architecture, and our Proposed Architecture

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.

3.3 Supporting Coherence and Consistency

The LSQ from the baseline architecture receives the invalidation requests from remote processors, so coherence and consistency functionalities can easily be supported in our technnique. However, we should highlight a conflict situation

that turns up in our design when implemented in a system with a MESI coherence protocol: If a data is replaced from the DL1 but remains in the Cached-LSQ, the Shared Line will not be activated due to a remote read request, potentially putting the remote data in an erroneous Exclusive State (instead of a Shared State). A possible solution is to force the LSQ to activate the Shared line for every remote read to a load whose data was received via forwarding. As a future work we intend to improve this management since –although straightforward– it is relatively inefficient.

4 Experimental Framework

We have evaluated our proposed design using the PTLsim [16], 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 [17], an AMD K8 and an Intel Core 2 [18]. 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
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 24 benchmarks from the SPEC CPU2006 suite, compiled for the x86 instruction set. The technology parameters correspond to 45 nm, with a 1.0V V_{dd}. We simulate regions of 100M instructions after reaching a triggering point [19], that marks the beginning of code area in which the application behavior is representative of the overall execution.

To evaluate the impact of our data cache filtering over the power consumption of the DL1, we use CACTI 5.3 [20] to model the cache of Table 1. Specifically, in order to estimate the cache power consumption, we have multiplied the number of reads and writes to DL1 by the power consumption of each kind of access to this cache. Furthermore, the simulator has been modified to incorporate our predictors in the microarchitectural simulation, although their power consumption is considered negligible compared with the power savings obtained in the data cache.

In the following, we perform some quantitative analysis to further understand the effectiveness of the proposed design.

5 Evaluation

5.1 Main Results

In this section we compare the data cache power and whole system performance using either the baseline or our alternative. Figure 2 shows the power savings achieved in the data cache in our technique with respect to the Original Architecture. Figure 3 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 as we will report in next subsection. 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 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%.

Second, and more important, in our architecture average performance remains almost untouched (around 0.1% of slowdown), something that would not happen

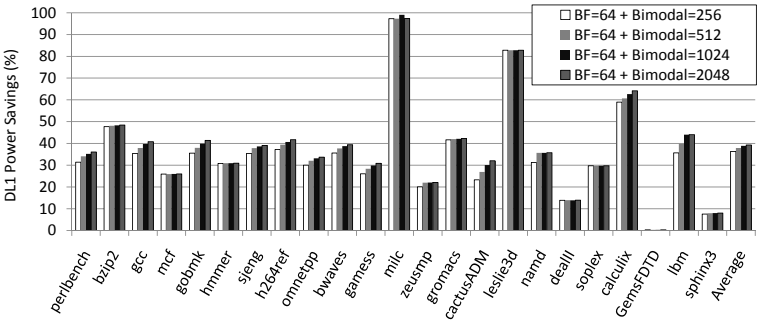

Fig. 2. DL1 Power Savings

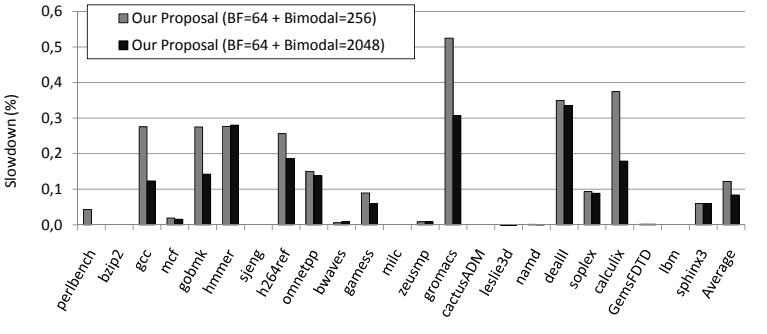

Fig. 3. Performance Impact

with Nicolaescu's Proposal. The reason is that in his case, when a load finds no previous dependent stores in the LSQ (i.e. has 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.

5.2 Forwarding Predictors

In order to compare the accuracy of the forwarding predictors evaluated –Bloom Filter, Bimodal (with 1 and 2 bits per entry), and Bimodal (2 bits) plus Bloom Filter– we follow Grunwald *et.al* and employ the following metrics used in confidence estimation for speculation control [21]:

- Predictive Value of a Positive test (PVP). It identifies the probability that the prediction of a load as dependent is correct. It is computed as the ratio between the number of correctly dependent-predicted loads and the total number of loads predicted as dependent.
- Predictive Value of a Negative test (PVN). It identifies the probability that the prediction of a load as independent is incorrect. It is computed as the ratio between the number of mispredicted independent loads and the total number of loads predicted as independent.

In our case, using predictors with a high PVP avoids degrading performance. On the other hand, if many loads are incorrectly independent-predicted (high PVN), many cache accesses are carried out unnecessarily, resulting in missed opportunities to reduce the DL1 power consumption. Therefore, in our design, only very high PVP values and very low PVN values are acceptable.

In Figure 4, we visually present the measurements of PVP and PVN for different sizes of all studied predictors. Intuitively, as we increase the size of

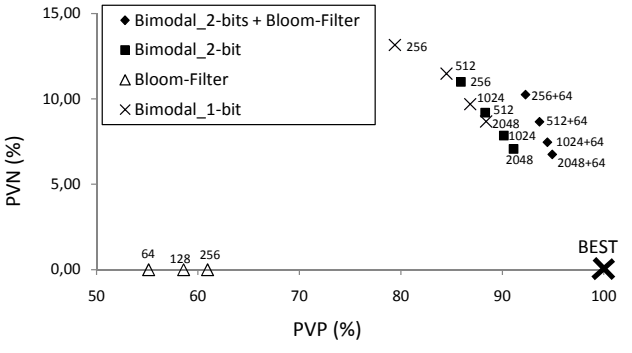

Fig. 4. PVP and PVP values for studied predictors. The results shown are the average values for all applications. For Bimodal Predictors (1 and 2 bits) the data points reflects sizes of 256, 512, 1K and 2K. For Bloom Filter we show results for 64, 128 and 256 entries. Finally, the combined predictor uses a 64-entry Bloom Filter and a Bimodal Predictor (2 bits) with 256, 512, 1K and 2K entries.

any predictor, PVP value augments and PVN decreases, leading to a better predictor behavior. Note that PVN for Bloom Filter is always zero, since no *false negatives* exist –when a load is independent-predicted, the predictor is never mistaken–. From this figure we can conclude –according to the intuition– that combining the past forwarding information (Bimodal predictor) and memory addresses (Bloom Filter) results in the most accurate predictor (around up to 95% of hits for predicted-dependent loads and only around 6% of misses for predicted-independent loads).

6 Conclusions

The main contributions of this paper are:

- We implement and evaluate Nicolaescu's CLSQ [3] in a different and more common microarchitectural model -the widespread x86-64-.
- We propose to include a forwarding predictor to know in advance whether a load will receive its data through forwarding, in which case the DL1 access can be avoided.
- We study the effectiveness of different predictors, choosing the optimal one based on a tradeoff between accuracy and HW needs.

Overall, the proposed filtering mechanism translates into DL1 power savings up to 36% on average for the studied predictor configuration (BF of 64 entries and Bimodal of 256 entries). Including this scheme leaves performance almost unvaried –less than 0.1% slowdown on average– with a minimal hardware cost of less than 100B.

References

1. Bower, F., Sorin, D., Cox, L.: The impact of dynamically heterogeneous multicore processors on thread scheduling. IEEE Micro 28(3), 17–25 (2008)
2. Hill, M.D., Marty, M.R.: Amdahl's law in the multicore era. IEEE Computer 41(7), 33–38 (2008)
3. Nicolaescu, D., Veidenbaum, A., Nicolau, A.: Reducing Data Cache Energy Consumption via Cached Load/Store Queue. In: ISLPED 2003, pp. 252–257 (2003)
4. Racunas, P., Patt, Y.N.: Partitioned First-Level Cache Design for Clustered Microarchitectures. In: ICS 2003, pp. 22–31 (2003)
5. Kin, J., Gupta, M., Mangione-Smith, W.: The Filter Cache: An Energy Efficient Memory Structure. In: MICRO 1997, pp. 184–193 (1997)
6. Albonesi, D.: Selective Cache Ways: On-Demand Cache Resource Allocation. Journal of Instruction-Level Parallelism 2 (2000)
7. Lee, H., Smelyanskiy, M., Newburn, C., Tyson, G.: Stack Value File: Custom Microarchitecture for the Stack. In: HPCA 2001, pp. 5–14 (2001)
8. Jin, L., Cho, S.: Reducing Cache Traffic and Energy with Macro Data Load. In: ISLPED 2006, pp. 147–150 (2006)
9. Subramaniam, S., Loh, G.: Store Vectors for Scalable Memory Dependence Prediction and Scheduling. In: HPCA 2006, pp. 65–76 (2006)

10. Park, I., Ooi, C., Vijaykumar, T.: Reducing Design Complexity of the Load/Store Queue. In: MICRO 2003, pp. 411–422 (2003)
11. 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), 27–36 (2006)
12. Sha, T., Martin, M., Roth, A.: Scalable Store-Load Forwarding via Store Queue Index Prediction. In: MICRO 2005, pp. 159–170 (2005)
13. Bloom, B.: Space/Time Trade-offs in Hash Coding with Allowable Errors. Communic. of the ACM 13(7), 422–426 (1970)
14. McFarling, S.: Combining Branch Predictors. Technical report tn-36, Western Research Laboratory, Digital Equipment Corporation (June 1993)
15. Sethumadhavan, S., Desikan, R., Burger, D., Moore, C., Keckler, S.: Scalable Hardware Memory Disambiguation for High ILP Procs. In: MICRO 2003, pp. 399–410 (2003)
16. Yourst, M.T.: PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. In: ISPASS 2007, pp. 23–34 (2007)
17. 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)
18. Copenhagen Univ. College of Eng.: The Microarch. of Intel and AMD CPU's: an Optimization Guide for Assembly Programmers and Compiler Makers (2009)
19. A hybrid timing-address oriented LSQ filtering for an x86 arch. Technical report
20. `http://www.hpl.hp.com/research/cacti/`
21. Grunwald, D., Klauser, A., Manne, S., Pleszkun, A.: Confidence Estimation for Speculation Control. In: ISCA 1998, pp. 122–131 (1998)