Reducing Cache Traffic and Energy with Macro Data Load

Lei Jin Sangyeun Cho
Department of Computer Science
University of Pittsburgh
{jinlei,cho}@cs.pitt.edu

ABSTRACT

This paper presents a study on *macro data load*, an efficient mechanism to enhance *loaded value reuse*. A macro data load brings into the processor a maximum-width data value the cache port allows, saves it in an internal structure, and facilitates reuse by later loads. A comprehensive limit study using a generalized *memory value reuse table* (MVRT) shows the significantly increased reuse opportunities provided by macro data load. We also describe a modified load store queue design as an implementation of the proposed concept. Our quantitative study shows that over 35% of L1 cache accesses in the SPEC2k integer and MiBench programs can be eliminated, resulting in a related energy reduction of 24% and 35% on average, respectively.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Design studies; C.1.1 [**Processor Architectures**]: Single Data Stream Architectures

General Terms

Design, performance

Keywords

Memory hierarchy, LSQ design, low power

1. INTRODUCTION

Just as caches filter memory accesses so that main memory sees much less traffic, cache read traffic can be tackled within a processor by store-to-load and load-to-load forwarding techniques [8, 10]. Cache traffic reduction can lead to significantly decreased energy consumption in the cache.

In this work, we propose and study *macro data load*, an efficient mechanism to uncover additional opportunities for load-to-load forwarding by utilizing the spatial locality that exists in cache port wide *macro data*. When this mechanism is in effect, memory loads always bring a macro data from the processor cache. For example, a byte load would trigger a 64-bit macro data transfer in a 64-bit processor[1]. The processor then provides the necessary data portion to the load instruction, while saving the loaded macro data in a separate data storage. The saved macro data can be retrieved by a later load targeting the whole data or a smaller part of it. Most previous works focused on reusing the exact data described by its address and size [8, 10]. The proposed macro data load mechanism capitalizes on the largely underutilized data cache port bandwidth which is mainly due to abundant narrow data accesses in programs.

As an implementation of the proposed concept, we present a load store queue design and the necessary microarchitectural changes. The implementation cost and complexity are shown to be very modest. We perform a detailed performance study using a realistic superscalar processor model incorporating the microarchitectural changes. Our results show that the proposed approach eliminates over 30% of L1 cache traffic leading to a commensurate energy reduction, compared with a conventional processor design.

The rest of this paper is organized as follows. First, we give a limit study on how macro data loads generate more opportunities for loaded value reuse in Section 2. Then Section 3 describes a set of microarchitectural techniques to support macro data loads efficiently. Section 4 presents our quantitative evaluation using a realistic processor model. Related works are summarized and contrasted with our work in Section 5. Lastly, conclusions are drawn in Section 6.

2. VALUE REUSE IN MEMORY ACCESSES

2.1 Evaluation model

To analyze the degree of data reuse among memory instructions, we constructed a 64-bit machine model with *memory value reuse table* (MVRT), a conceptual hardware structure that tracks the address, the value, and the type of memory instructions. Allocation of a new entry and replacement of an old entry is done in a FIFO fashion. MVRT is parameterized and can have a varied number of entries.

The memory value reuse algorithm works as follows. Whenever a new memory instruction is executed, it is recorded in MVRT. If it is a store, all the MVRT entries with a previous

[1]This does not usually incur a change in cache designs. A high-performance cache provides a full port wide value on loads (*e.g.*, [2]) and the processor core selects the necessary portion using its internal data alignment logic.

ISLPED'06, October 4–6, 2006, Tegernsee, Germany.

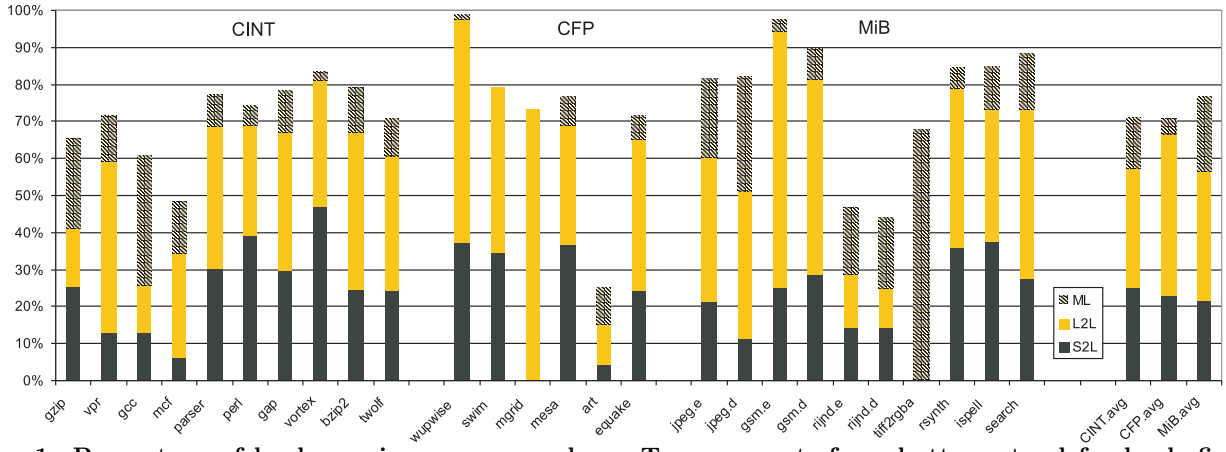

Figure 1: Percentage of loads reusing memory values. Two segments from bottom stand for loads finding their values from prior stores ("S2L" – store-to-load) and from prior loads("L2L" – load-to-load). Each top segment shows the extra opportunities offered by macro data loads ("ML").

memory instruction that overlaps in the address space with the store are invalidated. If the new instruction is a load, MVRT is searched to find a valid entry with a matching address, in which case, the load becomes redundant since the valid data can be provided from a previous memory instruction (either store or load).

For all experiments, we use a set of SPEC2k integer programs (dubbed "CINT" hereafter), SPEC2k floating-point programs ("CFP") [13], and MiBench programs ("MiB") [5]. After skipping the initialization phase, we collect analysis data from two billion instructions or until the end of execution if it comes first. Programs were compiled with gcc 2.7.2 targeting PISA [3] at the `-O3` optimization level.

2.2 Results

Although we performed a comprehensive limit study and analysis per data size, per memory region, and using different ISA, we only report a small set of results here due to space limitations. More complete results can be found in [6].

2.2.1 *Maximum memory value reuse*

In this subsection, we use a 256-entry MVRT to study how many loads find their reuse value from (1) previous stores only, (2) previous stores and loads without macro data loads, and (3) previous stores and loads with macro data loads. Figure 1 shows that on average 70% or more loads find their values within MVRT. Roughly, 20–25% of loads get a reuse value from stores and 30–40% of loads from previous loads if macro data loads are not used. Macro data loads consistently boost the number of loads that reuse a previously loaded data value. 13.6% (CINT) and 20.1% (MiB) more loads reuse memory values. In CFP programs the conventional load-to-load forwarding performs well and allows nearly 44% of loads to find their data in MVRT. Macro data loads provide a small benefit of only 4.3% additional loads. Considering only load-to-load value reuse, macro data loads provide 42.3% (CINT), 9.8% (CFP), and 57.2% (MiB) more reuse opportunities, relative to a conventional load-to-load forwarding technique.

2.2.2 *Sensitivity to MVRT size*

We changed the MVRT size from 16 to 256 and repeated our experiments. Several important observations are made from the results, although the graph is not shown here due to limited space.

First, a larger MVRT captures more value reuse opportunities. The number of covered loads increases almost linearly as we keep doubling the MVRT size, although the slope gradually dwindles after 64 entries in CINT and CFP.

Second, macro data loads expose significantly more opportunities for load-to-load forwarding in all the studied MVRT configurations, especially when MVRT is small. With a 32-entry MVRT, for example, there are 105% (CINT), 46% (CFP), and 188% (MiB) more loaded value reuse.

Third, as a result of our second observation, the area-effectiveness of MVRT, from the viewpoint of memory value reuse, is substantially improved. In the case of CINT and MiB, the total achievable degree of reuse with a 32-entry MVRT with macro data loads is comparable to that with a 256-entry MVRT without macro data loads.

3. MICROARCHITECTURE FOR LOADED VALUE REUSE

3.1 A modified LSQ design

Many recent high-performance processors implement LSQ to allow out-of-order execution of memory instructions, as well as to bypass correct store data to dependent loads [1, 4, 7, 15]. Adding the load-to-load forwarding feature in LSQ then becomes a natural extension since it already provides the necessary storage to save a loaded data and the functionality to identify it [8]. We present an example LSQ design that supports both store-to-load and load-to-load forwarding with the proposed macro data load mechanism. Figure 2 shows the LSQ, data cache, and how data from them are circulated via various buses.

A typical LSQ design is composed of two memory structures: *address-matching tag* and *data storage*. The data storage, not used for a load entry in a conventional design, can be used to hold a loaded macro data. We provide the necessary datapath to guide the data as observed at the cache port to the desired LSQ entry (Figure 2(d)).

The address tag portion is often implemented with an

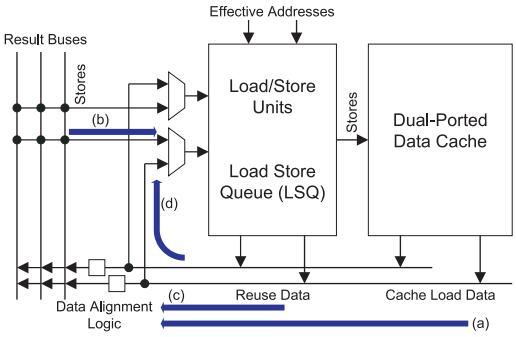

Figure 2: The LSQ structure with the macro data reuse mechanism implemented. (a) Normal load data path. (b) Store data path. (c) Reuse data path for both store-to-load and load-to-load forwarding. (d) LSQ update path. The depicted LSQ design is based on the published AMD K7 design [4].

associative memory logic such as *content addressable memory* (CAM), which helps resolve memory ordering conflicts quickly and locate a previous store for forwarding. To detect a load-to-load forwarding instance, however, LSQ should perform a *partial-match searching* since the source macro data can be larger than and inclusive of the dependent load, potentially leading to different address bits in several LSB positions. The tag has a few related bits in addition to the address, including V (valid), P (data present), and SL (store or load) bits. In a conventional design, an LSQ entry is invalidated once the corresponding instruction retires. We keep the V bit on as long as the data in the entry is up-to-date.

3.2 Pipeline design issues

To reduce L1 cache traffic, cache access should be suppressed whenever the target data can be found in LSQ. This requires that cache access be delayed and serialized with LSQ lookup. This arrangement however is likely to incur increased cache access latency for loads that do not reuse values and can potentially lead to decreased performance thereof. The performance degradation will highly depend on how many loads find their values in LSQ in this case. It is noted that the serialization delay may be avoided if we can (1) predict accurately whether or not a load will find its value in LSQ; and (2) selectively bypass the waiting time for loads that are likely to find their values in cache. This idea has been explored in similar, yet different contexts [11].

4. QUANTITATIVE EVALUATION

4.1 Experimental setup

We perform experiments using a detailed execution-driven simulator derived from sim-outorder in the SimpleScalar tool set [3]. We model a modest 4-issue processor, which has 32kB 2-way set associative L1 I/D cache, with 64B block size and 2-cycle latency. L2 cache is 2MB 4-way associative, with 128B block size and 10-cycle latency. Main memory has 120-cycle latency. The numbers of RUU entries is 128 while the LSQ size is set to 64, similar to recent high performance processors [4, 7, 15]. The data reuse latency in LSQ is set to one cycle. Section 2.1 describes the setup for our benchmarks.

We consider three configurations to reduce cache traffic: S2L, L2L and ML. They save load traffic on store-to-load forwarding, additionally on load-to-load forwarding (similar to [8]), and additionally on macro data load hits. The baseline processor, to which we compare these three configurations, accesses LSQ (for store-to-load forwarding) and cache at the same time to minimize latency and thus will have the highest number of cache accesses.

4.2 Evaluation results

4.2.1 L1 cache traffic

Figure 3 confirms the observations made in Section 2. With only store-to-load forwarding (*i.e.*, S2L), the cache traffic reduction is limited: 10% (CFP) or less (CINT and MiB). Only two programs among all the studied programs, namely wupwise and rsynth, show a traffic reduction of 15% or more. With ML, however, there is a significant reduction in cache accesses, 35% (CINT), 33% (CFP), and 38% (MiB). Four programs, namely bzip2, mgrid, jpeg.e, and gsm.d, had over 50% of cache traffic reduction. With L2L, the cache traffic reduction was 27% (CINT), 30% (CFP), and 23% (MiB), considerably lower than that of ML.

Although the results are in accordance with our limit study results presented in Figure 1, the actual traffic reduction is less than the maximum potential due to three factors: (1) speculative memory references occupy available LSQ entries and cause frequent pipeline flushing, not allowing memory value reuse between distant references; (2) speculative loads execute and generate cache traffic; and (3) memory reordering results in later loads accessing cache prior to or simultaneously with earlier loads, losing the reuse opportunities.

4.2.2 Energy consumption

We used the CACTI 100nm model [12] to consider the energy consumption related with both LSQ and cache. Since a memory reuse scheme can increase LSQ activities (*e.g.*, loads update the data array in LSQ), it is important to consider not only cache but also LSQ as we evaluate the energy consumption impact of different reuse schemes.

Experimental results show that S2L achieves a limited energy reduction of less than 10%. L2L performs better and results in additional savings totalling up to around 20%. ML performs consistently better than S2L and L2L, with a notable energy reduction of over 35% for MiB.

4.2.3 Performance impact

When the latency of memory value reuse is shorter than the cache access latency, increase in the number of loads finding their values from LSQ can lead to improved average memory latency. Our simulation configuration captures this case by setting the reuse latency to be one cycle and the cache access latency to be two cycles.

Simulation results show that the ML configuration is performance competitive with the baseline machine, in which, LSQ and cache are accessed simultaneously. For CFP and MiB, ML even performs slightly better. This is due to (1) many loads find their values from LSQ; and (2) the increased latency seen by the remaining loads is well tolerated by the out-of-order processor model.

Among the three traffic-optimized configurations, only the

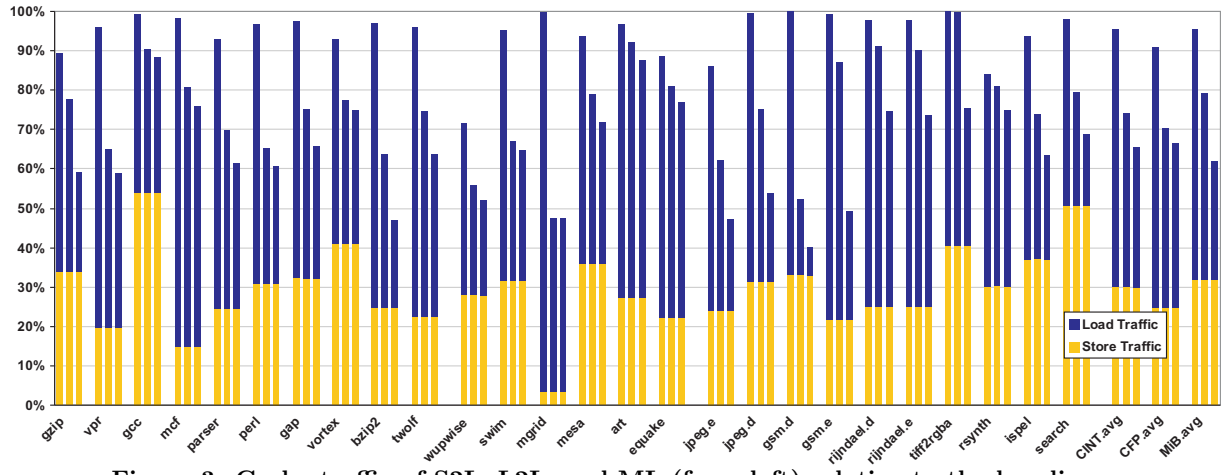

Figure 3: Cache traffic of S2L, L2L, and ML (from left) relative to the baseline.

;

ML configuration achieves a competitive performance level with the baseline design. It is noted that increase in execution time can have a detrimental effect on overall energy consumption.

5. RELATED WORK

Önder and Gupta proposed *value address association structure* (VAAS) to eliminate redundant loads and silent stores [10]. Nicolaescu *et al.* [8] proposed *cached load store queue* (CLSQ) to detect redundant loads and provide reuse data. In their design, each data entry in LSQ is allowed to cache a loaded value as well as to keep store data. Both VAAS and LSQ manage memory accesses with a FIFO policy and therefore our limit study with MVRT accurately models and predicts their performance. Our study showed that macro data loads give significantly boosted loaded value reuse compared with these techniques, given the same storage space to keep data.

More recently, Nicolaescu *et al.* [9] proposed *wide cached load store queue* (WCLSQ) to take advantage of spatial locality by having each LSQ entry keep multiple words or by increasing the LSQ data width to accomodate a large 16-byte or even 32-byte memory block. To fill WCLSQ, however, the cache should be accessed multiple times or the cache port should be widened to match the WCLSQ width. This approach potentially maximizes the loaded data reuse opportunities, but its effectiveness is offset by increased LSQ size (and thus energy consumption per access), increased initial overhead to fill WCLSQ, and much decreased area efficiency due to short stores occupying large data entries. Compared to WCLSQ, our proposal exploits only the freely available cache port bandwidth and requires no change to the cache design, promoting reuse of existing designs. It is also compatible with popular *cache subbanking* techniques [14].

6. CONCLUSIONS

This paper introduced and studied cache port wide macro data loads to enhance loaded value reuse in high-performance processors. As future microprocessors will be critically constrained by power consumption, performance-competitive, power-efficient design techniques will become even more important. Our work shows that the proposed macro data load scheme is practical and at the same time effective in reducing L1 cache traffic and energy.

7. REFERENCES

[1] D. Boggs, A. Baktha, J. Hawkins, D. T. Marr, J. A. Miller, P. Roussel, R. Singhal, B. Toll, and K. S. Venkatraman. "The Microarchitecture of the Intel Pentium 4 Processor on 90nm Technology," *Intel Technology Journal*, 8(1), Feb. 2004.

[2] D. Bradley, P. Mahoney, and B. Stackhouse. "The 16kB single-cycle read access cache on a next-generation 64b Itanium microprocessor," *Proc. Int'l Solid State Circuits Conf.*, pp. 110 – 111. Feb. 2002.

[3] D. Burger and T. M. Austin. "The SimpleScalar Tool Set, Version 2.0," *Computer Sciences Dept. TR*, No. 1342, Univ. of Wisconsin, June 1997.

[4] K. Diefendorff. "K7 Challenges Intel," *Microprocessor Report*, Vol. 12, No. 14, pp. 1 – 7, Oct. 1998.

[5] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," *Proc. Annual Workshop Workload Characterization*, Dec. 2001.

[6] L. Jin and S. Cho. "A Characterization Study on Memory Value Reuse," *Proc. Workshop Memory Performance Issues, during Int'l Symp. High-Performance Computer Architecture*, Feb. 2006.

[7] R. E. Kessler. "The Alpha 21264 Microprocessor," *IEEE Micro*, 19(2):24 – 36, March/April 1999.

[8] D. Nicolaescu, A. Veidenbaum, and A. Nicolau. "Reducing Data Cache Energy Consumption via Cached Load/Store Queue," *Proc. Int'l Symp. Low-Power Electronics and Design*, pp. 252, Aug. 2003.

[9] D. Nicolaescu, A. Veidenbaum, and A. Nicolau. "Caching Values in the Load Store Queue," *Proc. Int'l Symp. Modeling, Analysis, and Simulation of Computer and Telecomm. Systems*, pp. 580 – 587, Oct. 2004.

[10] S. Önder and R. Gupta. "Load and Store Reuse Using Register File Contents," *Proc. Int'l Conf. Supercomputing*, pp. 289 – 302, June 2001.

[11] T. Sha, M. Martin, and A. Roth. "Scalable Store-Load Forwarding via Store Queue Index Prediction," *Proc. Int'l Symp. Microarchitecture*, Nov. 2005.

[12] P. Shivakumar and N. P. Jouppi. "CACTI 3.0: An Integrated Cache Timing, Power, and Area Model," *HP WRL Research Report 2001/2*, Aug. 2001.

[13] Standard Performance Evaluation Corporation. `http://www.specbench.org`.

[14] C.-L. Su and A. M. Despain. "Cache Designs for Energy Efficiency," *Proc. Hawaii Int'l Conf. System Sciences*, pp. 306 – 315, Jan. 1995.

[15] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy. "POWER4 System Microarchitecture," *IBM J. Research & Development*, 46(1), Jan. 2002.