A Characterization Study on Memory Value Reuse

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

Abstract— **This paper presents a comprehensive characterization study on the exploitable memory value reuse present in programs. We compare three reuse schemes: *store value reuse*, *loaded value reuse*, and *macro data reuse* [12], [13]. Macro data reuse, enabled by macro data loads, capitalizes on under-utilized cache port bandwidth and makes use of the spatial locality found in port-wide macro data. Using a generalized *memory value reuse table* (MVRT) model, we present the results of (1) per program reuse analysis, (2) per data size analysis, (3) per region analysis, (4) per MVRT size analysis, and (4) estimating the impact of ISA and machine widths. The macro data load mechanism is shown to open up significantly more loaded value reuse instances compared with previous loaded value reuse proposals: over 75% (SPEC2k integer), 23% (SPEC2k floating-point), and 139% (MiBench) more load-to-load forwarding opportunities using a 64-entry MVRT. We also perform a quantitative study using a realistic processor model and show that over 35% of L1 cache accesses in the SPEC2k integer and MiBench programs can be eliminated, resulting in a related energy reduction of 27% and 31% on average, respectively.**

I. Introduction

Microprocessor performance is critically dependent on the timely delivery of data from memory. To fill the widening speed gap between a microprocessor and slow main memory, increasingly deep cache memory hierarchy has been used in virtually all high-performance microprocessors [10]. A well-designed memory hierarchy based on caches provides a view of fast and large main memory to a processor.

Just as caches filter memory accesses so that main memory sees much less traffic, memory references can be filtered inside a processor, reducing L1 cache traffic. Cache read traffic can be tackled with various store-to-load and load-to-load forwarding techniques [18], [16] and write traffic can be suppressed by squashing redundant stores [15], [18].

In this work, we present a comprehensive characterization study on the memory value reuse exploitable by three memory value reuse schemes: *store value reuse*, *loaded value reuse*, and *macro data reuse*. Macro data reuse is based on *macro data load*, which brings in full port-wide data whenever cache is accessed and keeps the macro data within processor. This makes it possible that future loads that reference the nearby addresses could reuse the *macro data* by exploiting the spatial locality [12], [13]. 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,

[1]This does not necessarily incur a change in cache design. A typical high-performance cache provides a full port wide value on loads (*e.g.*, [3]) and the processor selects the necessary portion using its internal data alignment logic.

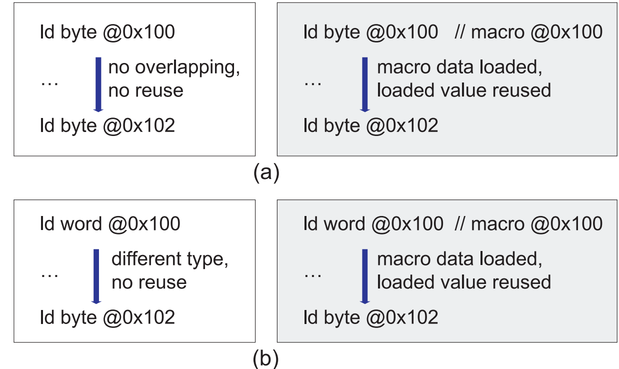

Fig. 1. Two examples where the macro data load mechanism provides additional memory value reuse opportunities. (a) Two byte loads target nearby data but are not related (left). With macro data loads, however, the first load brings in a macro data inclusive of the second data (right). (b) A word load targets a data inclusive of the data needed by the second load, but this case is not considered previously (left). With macro data loads, this case is subsumed into the previous case (right).

while saving the 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 portion of it. Two motivating examples showing the additional opportunities provided by macro data loads are illustrated in Fig. 1. Most previous works focused on reusing the exact data described by its address and size [18], [16].

To quantify the maximum available memory value reuse, we perform a comprehensive limit study using a generic *memory value reuse table* (MVRT) model. Our study with an aggressive 256-entry MVRT shows that macro data loads can expose 45% (SPEC2k integer), 10% (SPEC2k floating point), and 57% (MiBench) more load-to-load forwarding opportunities, compared to a conventional loaded value reuse technique. With a more realizable 64-entry MVRT, macro data loads provide over 75% and 139% more load-to-load opportunities for SPEC2k integer and MiBench programs, respectively, and over 23% more for SPEC2k floating-point programs. We also perform a quantitative study using a realistic processor model and corroborate our characterization study results.

The rest of this paper is organized as follows. In Section II, we give a comprehensive limit study on how many reuse opportunities are detected by different schemes. Section III then discusses some implementation issues and presents a quantitative evaluation of the memory reuse schemes. Related works are summarized and contrasted with our work in Section IV. Lastly, conclusions will be drawn in Section V.

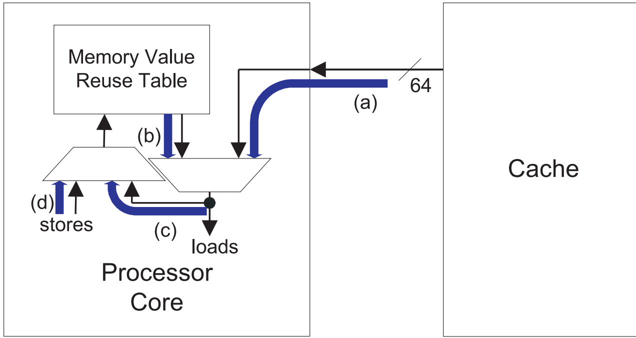

Fig. 2. A processor model with MVRT. Data paths related with loaded value reuse are highlighted. (a) Load data path from cache. (b) Load data path from MVRT. (c) MVRT update data path from a prior load. (d) MVRT update data path from a store.

II. Value Reuse in Memory Accesses

A. 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 modified Load/Store Queue 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. Fig. 2 depicts the processor model.

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 memory instruction that overlaps in the address space with the store address 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). Once MVRT provides a value for reuse, it can be saved back into the newest MVRT entry corresponding to the current load, in the hope that the value can be further reused. We call this operation *data promotion*. If there is no matching entry found in MVRT, then cache is accessed to fetch port wide data block to a newly allocated MVRT entry. MVRT supports memory value reuse analysis both with and without the macro data load mechanism.

We note that the MVRT size roughly accounts for the size and complexity of a hardware mechanism to implement a memory value reuse technique. The impact of the MVRT size on memory value reuse will be discussed in Section II-B.4.

For all experiments, we use a set of SPEC2k integer programs (dubbed "CINT" hereafter), SPEC2k floating-point programs ("CFP") [23], and MiBench programs ("MiB") [9]. After skipping the initialization phase [21], we profile and 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 [4] at the `-O3` optimization level.

B. Results

1) Maximum memory value reuse: In this subsection, we look at the maximum reuse offered by different memory value reuse techniques using a 256-entry MVRT[2]. More specifically, we 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. Fig. 3 shows the result.

The result 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 without macro data loads. 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 the 256-entry 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, compared to a conventional load-to-load forwarding technique.

There are several programs showing interesting value reuse behaviors. In wupwise, almost all loads reuse previous memory values. This is due to data promotion between MVRT entries. For the studied program phase in wupwise, stores generate values used by later loads and the values are further passed to even later loads continuously. This behavior lasts for the entire span of the examined program phase. In fact, with only 16 entries in MVRT, over 75% of all loads find their values in MVRT, suggesting that the reuse distance in wupwise is very short. gsm.e showed a similar behavior.

Many loads in mgrid find their values from previous loads, but not from previous stores. This suggests that newly generated store values are seldom used by the following loads at least within the next 256 memory instructions.

In tiff2rgba memory reuse was exposed only when macro data loads came into play. It shows a long scanning access pattern over short data items. Since these items are loaded once, a conventional load-to-load forwarding scheme does not find any reuse opportunities, nor does a store-to-load forwarding technique. With macro data loads, however, each loaded macro data can potentially provide a portion of the macro data block to short loads accesses multiple times.

2) Per data size reuse analysis: Before looking at how loads of different sizes exploit memory value reuse, it is worthwhile to consider the dynamic mix of load sizes, presented in Fig. 4.

It is shown that programs show vastly different mixes. In CINT programs, the dominating size is word (32 bits). In FP programs, word and double word accesses are pronounced, covering nearly 98% of all loads. In MiB programs, however,

[2]We chose a large MVRT size to study a "maximum" degree of reuse that different schemes can offer without first confining our discussions to a specific configuration.

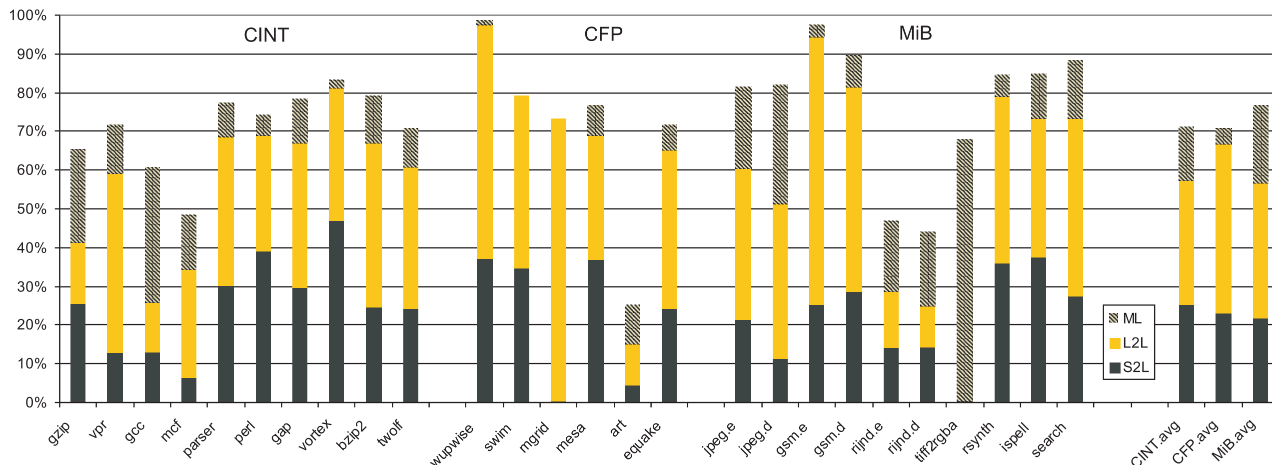

Fig. 3. Percentage of loads reusing memory values. Two segments from bottom stand for loads finding their values from prior stores ("S2L" – store-to-load) and additionally from prior loads without macro data loads ("L2L" – load-to-load). Each top segment shows the extra opportunities offered by macro data loads ("ML"). The MVRT size is 256.

there are significantly more byte and half-word loads than the CINT and CFP programs. In general, the mix of different size loads is largely dependent on the algorithm, how data structures are designed accordingly, the programming practices, the compiler, and the instruction set architecture (ISA).

Given this, Fig. 5 presents the decomposition of memory value sources per data size, showing that small data types, such as byte and half, benefit more from macro data loads than other types across all the examined benchmarks. This is due to the exploitable spatial locality provided by earlier macro data loads.

Word loads find their values from a previous store more frequently than byte and half word loads. This is because stack references fall into this category in the studied ISA, see Section II-B.3. Although double word loads can benefit from our scheme by reusing the values loaded by previous smaller loads, we seldom find such occasions in the studied programs. Interestingly, double word loads in the INT and MiB programs find their values very often from an earlier store.

3) Per region reuse analysis: In this subsection we review the memory reuse behavior of loads targeting data in different regions, see Fig. 6. First of all, loads targeting the stack region frequently find their values from previous stores - as often as 70% or more in CINT. Stack pushes and pops often form a producer-consumer relationship that is well detected and exploited by a store-to-load forwarding scheme [6].

CINT and MiB programs have more loads going to the heap region than other regions and CFP programs have more loads going to the data region than other regions; This is responded in the all-load average bar. Except in the stack region, loaded value reuse is more pronounced than store value reuse.

4) Sensitivity to MVRT size: We changed the MVRT size from 16 to 256 and repeated our experiments. Results are presented in Fig. 7 and several observations are made.

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. This trend is projected to be sustained even further if we increase the MVRT size beyond 256 entries [5], as MVRT with a FIFO replacement policy begins to simulate a regular data cache with an LRU policy [10]. This suggests that there is a large amount of exploitable memory value reuse opportunities (already 70% with 256 entries but possibly more beyond this point), although we will have to employ a large expensive hardware to bookkeep many memory values to exploit all the opportunities, if not impossible.

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 with macro data loads. This suggests that the reuse distance becomes significantly smaller in the presence of macro data loads. Extra data we bring and keep in MVRT using freely available cache port bandwidth are well subscribed to.

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. This suggests that the macro data load mechanism can provide critical implementation advantages over previous techniques, especially when hardware budget is limited and thus any optimization strategy should be carefully justified.

5) Impact of ISA: In this subsection we consider several interesting aspects of memory data reuse associated with ISA. The PISA used in this work is a 64-bit ISA supporting 32-bit address space. Stack references are done in word size.

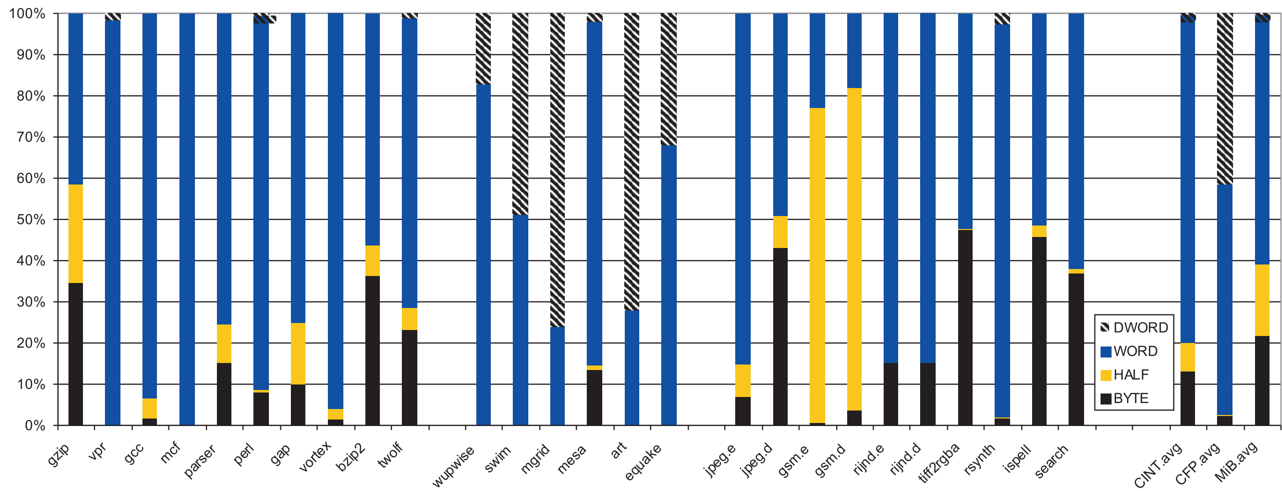

Fig. 4. Mix of loads targeting different-size data.

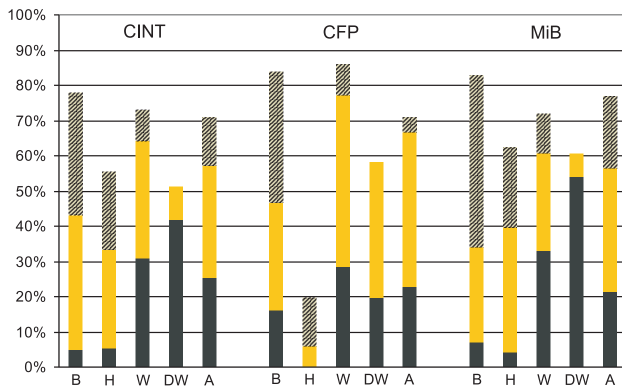

Fig. 5. Percentage of loads reusing memory values per type: byte ("B"), half word ("H" – 16 bits), word ("W" – 32 bits), and double word ("DW" – 64 bits). The rightmost bars ("A") are for all loads. The MVRT size is 256.

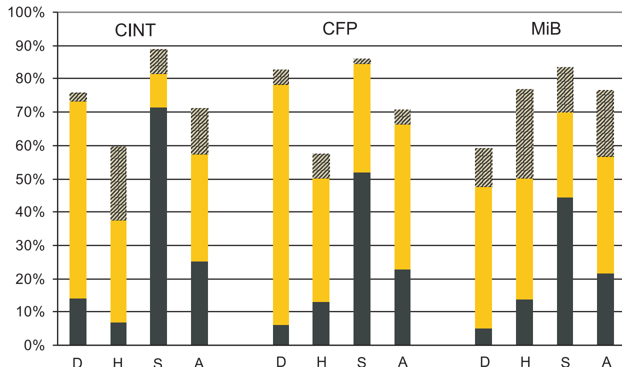

Fig. 6. Percentage of loads reusing memory values per region: data ("D"), heap ("H"), and stack ("S"). "A" is for all loads. The MVRT size is 256.

Double word type loads are generated when programs have `long long` or `double` type variables. On the other hand, binaries compiled for the Alpha architecture, for instance, will generate more 64-bit memory accesses to handle pointers, due to its 64-bit address space. Alpha binaries have considerably more double word loads than PISA. Our first-order experience with the Alpha ISA shows however that macro data loads still uncover 23% and 38% more opportunities for load-to-load forwarding in a set of SPEC2k integer and floating-point programs, respectively, when MVRT has 128 entries [12].

It is worthwhile to consider the following two questions: (1) "how much gain can we get from macro data loads on a 32-bit machine?" and (2) "how much gain can we get when we run 32-bit applications on a 64-bit machine?" To answer these questions, we projected load type mixes (byte, half, and word) of 32-bit binaries using the result in Fig. 4. We further assumed that one double word access becomes two word accesses. Then we apply the results in Fig. 5 based on a simple assumption that the degree of reuse per type does not change and fetching a double word using two word accesses leads to the second access covered by a previous macro data load on a 64-bit machine.

Fig. 8 provides our projections to address the above two questions. It is shown that on a 32-bit machine, macro data loads uncover very limited additional load-to-load forwarding opportunities for the CFP programs as they have few byte and half loads. On the other hand, the CINT and MiB programs still benefit considerably as they have many loads accessing small size data.

When we run 32-bit binaries on a 64-bit machine, macro data loads offer significantly more opportunities for load-to-load forwarding. Even CFP programs crop over 38% more load-to-load opportunities. The CINT and MiB programs have a small number of double word loads (0.3%) and thus their behavior is virtually same as what we observe in the

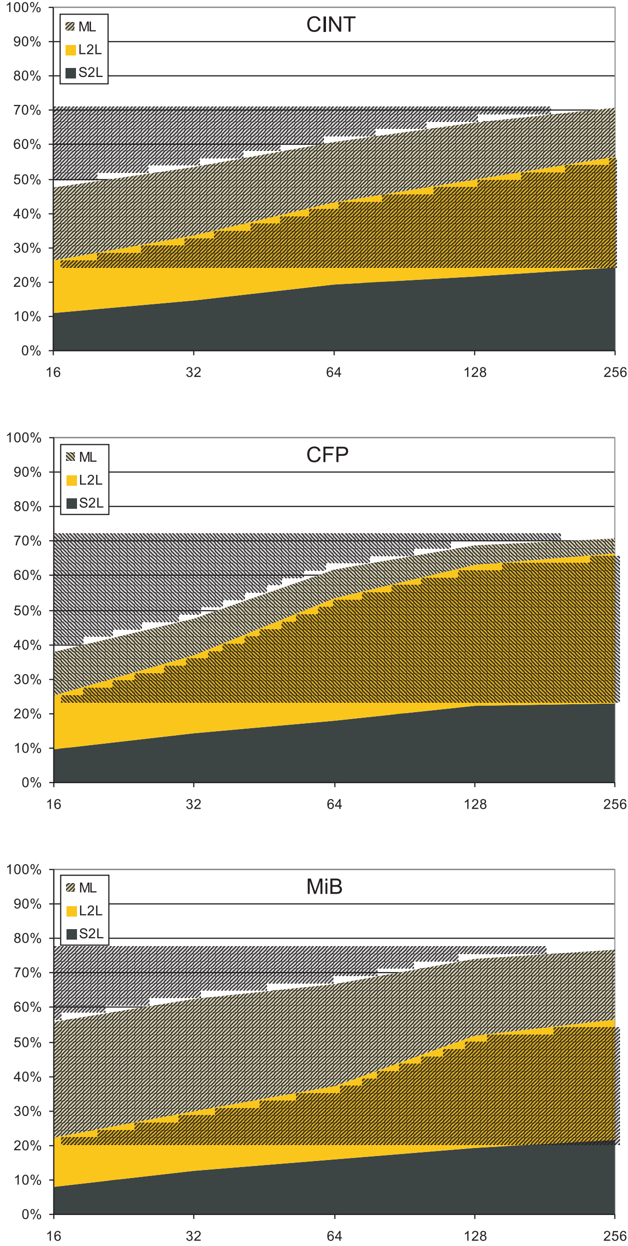

Fig. 7. Memory value reuse with varying MVRT sizes. 16 entries (128 bytes) to 256 entries (2k bytes) are considered.

PISA results. The findings in this subsection are particularly encouraging for us since many 32-bit legacy applications are running on 64-bit platforms today [11], [26].

6) Impact of false sharing: Macro data loads can give rise to previously nonexistent dependencies between loads and stores. Consider the following memory access sequence:

```
ld.w r1, @0x10001000
st.w @0x10001004, r2
ld.w r3, @0x10001000
```

In a conventional load store processing mechanism, the store does not interfere with the other two loads, allowing the second

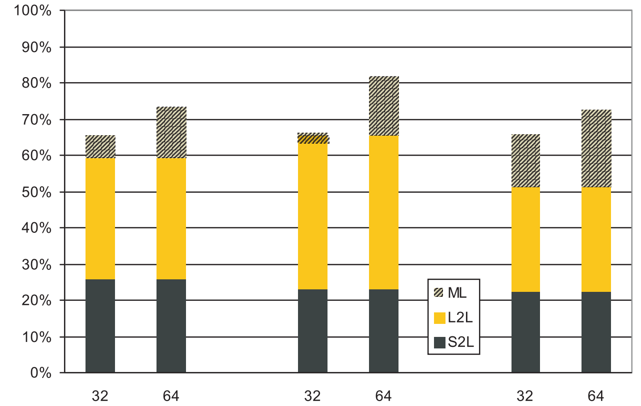

Fig. 8. Projected maximum reuse when running 32-bit binaries on a 32-bit machine ("32") or a 64-bit machine ("64").

load to use the value loaded by the first load.When the macro data load mechanism comes into play, however, the data loaded by the first load becomes a double word and will be killed by the store since they share data (unless there is a partial data invalidation mechanism, which we do not consider in this paper). We examined our benchmarks and found that there are three programs – bzip2, gsm.e, and gsm.d – that lose some load-to-load forwarding opportunities due to this false sharing when MVRT has 256 entries. Interestingly, the lost opportunities were mostly for word data and these programs still benefited from macro data loads thanks to other loads targeting smaller data. Moreover, the impact of false data sharing disappears quickly as we reduce the size of MVRT to 128 and fewer.

Another interesting data sharing pattern is that large store data can be targeted by smaller loads. In gsm.d, for example, there are byte loads bringing data written by previous word stores.Programming practices leading to such occasions are not prevalent, however, and we do not find many such instances in the studied programs. It is noted that recent processors support the store-to-load forwarding of this *misaligned* data [24], [2].

III. Quantitative Evaluation

A. Microarchitectural change

In this subsection, we briefly discuss two major aspects of a microarchitecture that supports memory value reuse: *data storage* to keep memory values and *address matching* to detect reuse opportunities. For more details, readers are referred to our previous work [13]. We use the *load store queue* (LSQ) commonly found in modern superscalar processors [25], [8], [14], [24], [2] to implement memory reuse schemes, targeting both store and loaded values [16], [12]. To further support macro data loads, we provide a datapath from cache ports to LSQ so that data as seen at a cache port is stored intact in a LSQ entry.

The data storage in LSQ, typically used only for storing store values, is used to hold a loaded macro data. The address tag portion is often implemented with an associative memory logic such as *content addressable memory* (CAM), and is

Issue width	4
Branch pred.	Combined, 2k-entry BTB
I-cache	32kB, 2-way, 64B line, 2-cycle latency
D-cache	32kB, 2-way, 64B line, 2-cycle latency, 2 ports
L2 cache	2MB, 4-way, 128B line, 10-cycle latency
Mem. latency	120 cycles
RUU/LSQ	128 and 64 entries

Fig. 9. Summary of the simulated machine model.

extended to detect a previous load as well as a previous store for forwarding opportunities. 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 non-identical addresses in several LSB positions.

When a macro data item in LSQ is accessed, it should go through the *data alignment logic* so that only the necessary portion of it is extracted and aligned before being launched on result buses. A conventional processor implements this logic at the cache port boundary so that thereafter the actual value as viewed by the running program will be circulated. In our LSQ design, this logic is placed after the LSQ read ports or duplicated for LSQ to handle data from LSQ as well as cache. Effectively, the macro data from LSQ and cache are treated in the same manner. This arrangement does not affect the timing of data arrival from cache, but may do so for data from LSQ. Since the necessary selection bits (derived from the tag bits) are available ahead of the data, fortunately, fast data alignment logic can be constructed. Recent processors support similar data selection function to deal with misaligned data in LSQ efficiently [24], [2].

In summary, we can turn a conventional LSQ design into a small, level-0 cache within the processor core with relatively small overheads. It is fully-associative, has a port-wide line size, and supports a FIFO allocation and replacement policy. Implementing this highly effective L0 cache requires only a small change in a typical LSQ and memory pipeline design.

B. Studied memory pipeline configurations

We study four different configurations: BASE-P, BASE-T, OPT-T, and OPT-P. The BASE configurations resemble conventional processors and perform store-to-load forwarding. The OPT configurations allow both store-to-load and load-to-load forwarding with macro data loads. The "T" configurations optimize for *cache traffic* and do not access cache until it is known that LSQ does not have a reuse value for forwarding. This is implemented by inserting a pipeline stage to look up LSQ for possible matching before cache access can commence. The "P" configurations focus on *processor performance*, initiating cache access and LSQ reuse look up in the same clock cycle. Most commercial processors implement a form of BASE-P [8], [14], [24], [2].

C. Experimental setup

We perform experiments using a detailed execution-driven simulator derived from sim-outorder in the SimpleScalar tool set [4]. We model a modest 4-issue processor whose important parameters are summarized in Fig. 9. The LSQ size is set to 64, similar to recent high performance processors [8], [14], [24]. The data reuse latency in LSQ is set to one cycle. Section II-A describes the setup for our benchmarks.

D. Evaluation results

1) L1 cache traffic: Fig. 10 shows the result, which confirms the observations made in Section II. With only store-to-load forwarding (*i.e.*, BASE-T), the cache traffic reduction is limited: 10% (CFP) or less (CINT and MiB). Only two programs among all the studied programs, namely parser and wupwise, show a traffic reduction of 20% or more. With both store-to-load and load-to-load forwarding enhanced with macro data loads (*i.e.*, OPT-T), however, there is a significant reduction in cache accesses, 35% (CINT), 33% (CFP), and 38% (MiB). If we consider load traffic only, the reductions correspond to 49% (CINT), 44% (CFP), and 55% (MiB). Four programs, namely bzip2, mgrid, jpeg.e, and gsm.d, had over 50% of cache traffic reduction. With store-to-load and load-to-load forwarding *without* macro data loads (not shown), the cache traffic reduction was 27% (CINT), 30% (CFP), and 23% (MiB), considerably lower than that of OPT-T; We do not further detail this configuration in the following discussions.

Although the results are in accordance with our limit study results presented in Fig. 3 and Fig. 7, the actual traffic reduction is less than the maximum potential due to three factors: (1) speculative memory references frequently occupy available LSQ entries and cause pipeline flushing, thereby not allowing memory value reuse between distant references; (2) speculative loads execute and generate cache traffic; and (3) memory reordering can result in later loads accessing cache prior to early loads, losing the reuse opportunities.

2) Energy consumption: We used the CACTI 100nm model [22] to consider the energy consumption related with both LSQ and cache. As the proposed reuse scheme increases LSQ activities (*e.g.*, loads update the data array in LSQ), it is important to consider not only cache but also LSQ when evaluating the related energy consumption of different memory reuse schemes. The modified LSQ we modeled is effectively a fully associative cache. We expect the CACTI could give an approximate evaluation of LSQ's power consumption. Fig. 11 shows the result.

As can be expected, the "T" configurations achieve less energy consumption compared with the corresponding "P" configurations. OPT-P consumes more energy than BASE-P due to an increase in LSQ energy. Comparing OPT-T to BASE-P, up to 31% of energy reduction (MiB) is observed. Roughly 27% of energy reduction is achived for CINT and CFP. Energy reduction due to BASE-T is limited (less than 10%) because its cache traffic reduction is limited.

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 performance. Our simulation configuration captures this case by setting the reuse latency to be one cycle and the cache access latency to be two cycles.

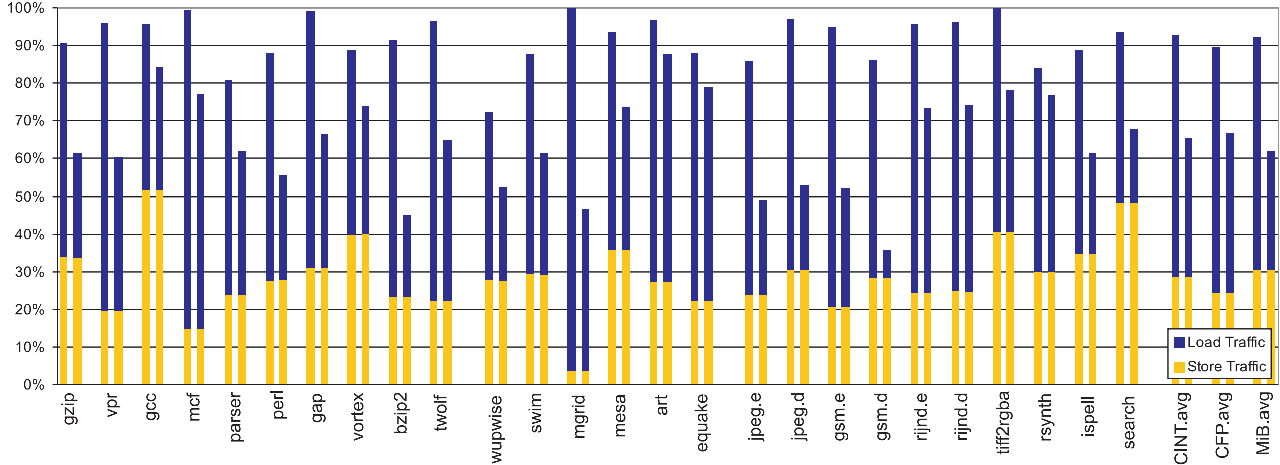

Fig. 10. Cache traffic of BASE-T (left) and OPT-T (right) relative to BASE-P (not shown). For readability, results for OPT-P were omitted as they are close to those of BASE-P.

The BASE-T and OPT-T configurations show lower performance than the BASE-P and OPT-P configurations, respectively. Since the OPT configurations achieve far more memory value reuse, their performance (OPT-T and OPT-P) is better than that of the BASE-T and BASE-P configurations. OPT-P achieves a small performance improvement over BASE-P; There were three programs with a noticeable performance improvement of 3% or more: gzip (4.4%), gap (4.4%), and bzip2 (3.0%). It is further shown that the traffic-optimized OPT-T configuration is performance-competitive with the performance-optimized BASE-P configuration since (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.

Overall, the positive performance impact (*i.e.*, OPT-P vs. BASE-P) was not as pronounced as the traffic and energy reduction (*i.e.*, OPT-T vs. BASE-P). An interesting aspect here is that extensive memory value reuse enabled by macro data loads makes the processor performance less sensitive to the cache latency. Increasing the cache latency to 3 cycles resulted in 4.2% (CINT), 0.9% (CFP), and 1.1% (MiB) performance degradation for BASE-P, but only 2.2%, 0.6%, and 0.4% for OPT-P.

IV. Related Work

Bodik *et al.* provides a limit study on dynamic load reuse and develops a compiler-based load reuse analysis technique [1]. Using a set of SPEC95 benchmarks, they show that 55% of all loads exhibit reuse, among which their compiler analysis can expose about 80%. Önder and Gupta proposed *value address association structure* (VAAS) to eliminate redundant loads and silent stores [18]. By associating with each physical register the address (and some additional information) of a memory instruction and inserting a new pipeline stage to perform an associative address search, they detect and eliminate nearly 60% of loads using a 128-entry

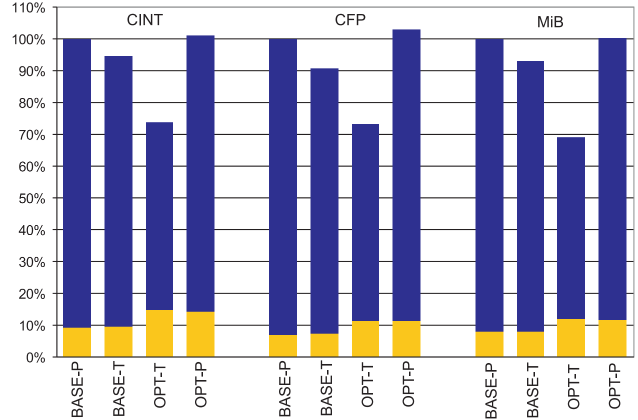

Fig. 11. Energy consumption of LSQ and cache, relative to BASE-P. Each bar is stacked and divided into two parts, cache access energy (upper bars) and energy spent in LSQ (lower bars).

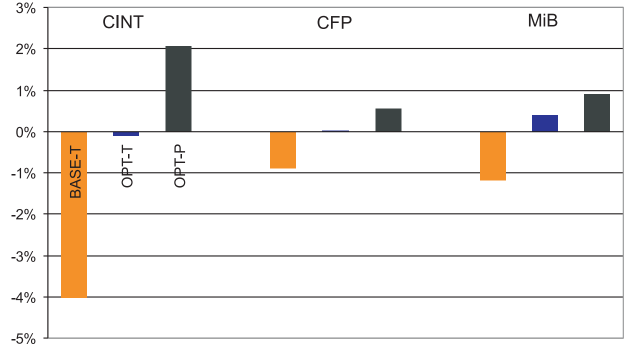

Fig. 12. Performance difference of BASE-T, OPT-T, and OPT-P relative to BASE-P.

VAAS. Bodik *et al.* and Önder and Gupta suggest that the available memory reuse can be well exploited by a FIFO-style hardware mechanism.

Nicolaescu *et al.* [16] proposed *cached load store queue* (CLSQ) to detect redundant loads and provide reuse data. In their design, each data entry in CLSQ is allowed to cache a loaded value as well as to keep store data. Using LSQ to capture loaded values is especially attractive since many modern high-performance processors already employ an LSQ and implement necessary logic to detect dependent memory references for store-to-load optimizations [25], [8], [14], [24], [2]. Both VAAS and LSQ manage memory accesses with a FIFO policy and therefore our limit study with MVRT accurately models and predicts their performance.

More recently, Nicolaescu *et al.* [17] 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 accommodate a large 16-byte or even 32-byte memory block. To fill WCLSQ, the cache should be accessed multiple times or the cache port should be widened to match the WCLSQ width. This approach can potentially exploit more spatial locality (*i.e.*, more hits in one LSQ entry) than our macro data approach, at the expense of increased LSQ size and decreased area efficiency due to short stores occupying large data entries. Compared to this work, our proposal exploits only the freely available cache port bandwidth and requires little change to the cache design.

V. Conclusions

This paper presented a detailed characterization study on the degree of memory value reuse exploitable by three different schemes: *store value reuse*, *loaded value reuse*, and *macro data reuse*. The reported results include per program reuse analysis, per data size analysis, per region analysis, per MVRT size analysis, and the impact due to different ISA/machine width combinations. Our work shows that the macro data load mechanism provides significantly more loaded value reuse opportunities compared with a conventional loaded value reuse scheme: 75% (CINT), 23% (CFP), and 139% (MiB) more load-to-load forwarding instances when MVRT has 64 entries.

Further, we model different memory value reuse schemes in an execution-driven simulator and perform a detailed simulation study using a realistic processor configuration. We report a cache traffic reduction of 35% (CINT), 33% (CFP), and 38% (MiB) on average with macro data loads, compared with a conventional processor configuration. Savings in LSQ and cache energy with macro data loads were 27% (CINT), 27% (CFP), and 31% (MiB).

Just as store-to-load forwarding techniques have become conventional in high-performance processors, we expect that an area-efficient and complexity-effective load-to-load forwarding technique like the macro data load proposal will be seriously considered in future processors. Our evaluation results will be a valuable reference for a real design.

References

[1] R. Bodik, R. Gupta, and M. L. Soffa. "Load-Reuse Analysis: Design and Evaluation," *Proc. Int'l Conf. Programming Language Design and Implementation*, pp. 64 – 76, May 1999.

[2] 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*, Vol. 8, No. 1, Feb. 2004.

[3] 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.

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

[5] J. F. Cantin and M. D. Hill. "Cache Performance for Selected SPEC CPU2000 Benchmarks," *Computer Architecture News*, Sept. 2001.

[6] S. Cho, P.-C. Yew, and G. Lee. "Decoupling Local Variable Accesses in a Wide-Issue Superscalar Processor," *Proc. Int'l Symp. Computer Architecture*, pp. 100 – 110, May 1999.

[7] K. Cooper and L. Xu. "An Efficient Static Analysis Algorithm to Detect Redundant Memory Operations," *Proc. Workshop. Memory System Performance*, pp. 97 – 107, Aug. 2002.

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

[9] 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.

[10] J. L. Hennessy and D. A. Patterson. *Computer Architecture A Quantitative Approach*, 2nd Ed., Morgan Kaufmann Publishers, 1996.

[11] Intel Corp. "Intel Itanium Processor Family Reference Guide: IA-32 Execution Layer," *Quick Reference Guide*, 2004.

[12] L. Jin and S. Cho. "Macro Data Load and Loaded Value Reuse," *Submitted for publication*, Also as *CS Tech. Report*, TR-05-125, Univ. of Pittsburgh, Sept. 2005.

[13] L. Jin and S. Cho. "Enhancing Loaded Value Reuse with Macro Data Load," *Submitted for publication*, Also as *CS Tech. Report*, TR-05-130, Univ. of Pittsburgh, Nov. 2005.

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

[15] K. M. Lepak and M. H. Lipasti. "Silent Stores for Free," *Proc. Int'l Symp. Microarchitecture*, pp. 20 – 31, Dec. 2000.

[16] 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 – 257, Aug. 2003.

[17] 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.

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

[19] A. Roth. "A High-Bandwidth Load/Store Unit for Single- and Multi-Threaded Processors," *CIS Tech. Report* MS-CIS-04-09, Univ. of Pennsylvania, June 2004.

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

[21] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. "Automatically Characterizing Large Scale Program Behavior," *Proc. Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 45 – 57, Oct. 2002.

[22] 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.

[23] `http://www.specbench.org`.

[24] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy. "POWER4 System Microarchitecture," *IBM J. Research & Development*, Vol. 46, No. 1, Jan. 2002.

[25] K. C. Yeager. "The MIPS R10000 Superscalar Microprocessor," *IEEE Micro*, Vol. 16, No. 2, pp. 28 – 40, April 1996.

[26] C. Zdebel and S. Solotko. "The AMD64 Computing Platform: Your Link to the Future of Computing," *White Paper*, May 2003.