Load Squared: Adding Logic Close to Memory to Reduce the Latency of Indirect Loads with High Miss Ratios

Sami Yehia† Jean-Francois Collard‡
Olivier Temam†
† INRIA Futurs and LRI, University of Paris-Sud
‡ Hewlett-Packard Laboratories, 1501 Page Mill Road, Palo Alto CA 94304

Abstract

Indirect memory accesses, where a load is fed by another load, are ubiquitous because of rich data structures and sophisticated software conventions, such as the use of linkage tables and position independent code. Unfortunately, they can be costly: if both loads miss, two round trips to memory are required even though the role of the first load is often limited to fetching the address of the second load. To reduce the total latency of such indirect accesses, a new instruction called load squared is introduced. A load squared does two fetches, the first fetch reading the target address of the second. (An offset is optionally added to the result of the first fetch.) The load squared operation is performed by memory-side logic (typically, the memory controller if it isn't located on the main processor chip). In this study, load squared is not an architecturally visible instruction: the micro-architecture transparently decides which loads should be replaced by loads squared. We show that performance is sometimes improved significantly, and never degraded.

1. Introduction

Indirect memory accesses occur in a wide range of data structures (lists, graphs, sparse matrices,...) and programming constructs (e.g., linkage tables a.k.a global offset tables) and can induce severe performance degradations. However, to the best of our knowledge, only prefetching schemes have been proposed to tackle them: for instance, Roth et al. [14] proposed a hardware prefetching scheme for efficiently traversing pointer-based data structures (graphs, lists), and more recently Cooksey et al. [6] proposed a tagged-prefetching-like scheme for indirect accesses that showed significant potential for commercial applications. However, the scope of such schemes may be limited by the difficulty of properly identifying indirect memory accesses.

In this article, we propose a hardware scheme for reducing the latency of many indirect memory accesses that relies on two key features: (1) a way to identify indirect memory accesses using simple hardware to track chains of dependent loads, rather than "guessing" the occurrence of indirect accesses as in prefetching schemes, and (2) logic (and some tables, see below) to perform indirect loads as close as possible to main memory (in the memory controller, as evaluated in our experimental set-up, or possibly in memory) in order to reduce the total indirect load memory latency. Moreover, our scheme applies not only to complex data structures traversals, but also to very irregular and hard to predict indirect accesses such as accesses to linkage tables and sparse matrices accesses. Note that access through the linkage table is extremely common due to two factors. One is preemption: A symbol can be preempted if some time after linkage, the object it refers to, or the address of that object, may change. In Linux, all symbols but the internal ones (local data of procedures, static procedures and variables) can be preempted. The second is position-independent code (PIC). For instance, ELF linkers support PIC code with a linkage table in each shared library that contains pointers to static data referenced in the main program. The dynamic linker resolves and relocates all of the pointers in the linkage table. The sad thing is, only one read was required by the algorithm; the indirection here is only due to software convention.

The general code pattern tackled by our scheme is:

```
load b = [a]
add c = b+d
load v = [c]
```

which corresponds to most indirect accesses. (We use the Alpha ISA, whose `ldq` instruction uses a base register and an offset. Thus, code patterns we target may not have an intervening `add`.) In a nutshell, our method operates as follows: the dataflow dependence between

two nearby load instructions is identified and recorded, the behavior of both loads (hit/miss) is recorded and if both are found to miss frequently, then the second load is dynamically replaced by a new load instruction, called *load squared*, which takes two arguments, the address `a` of the first load and displacement `d`. The load squared instruction returns the value stored at address `[a]+d`, where `[a]` is the content of the pointer-sized location pointed to by `a`. Note that two loads use address `a`: the first load instruction, and the inserted load squared instruction.

We applied this scheme to all SpecInt and SpecFP benchmarks [7]. Providing results for all SPEC benchmarks is important for two reasons: first, it prevents us from showing only favorable results. Second, it shows that, even though the performance speedup we get are not always remarkable, we slow none of them down. We also show results on 9 of the 10 Olden benchmarks [13]. (We could not make `voronoi` execute correctly.)

In Section 2, we present the principles and implementation of our scheme, in Section 3, we present the experimental framework and the performance evaluation in Section 4.

2. Principles

The general principle of our approach is to reduce the overall latency of two dependent loads, and more precisely, the time necessary to perform two round-trips to memory, in case both loads miss. As noted in the introduction, the value of the first load is brought back to the processor for the sole purpose of computing the address of the dependent load, and in many cases, this computation simply consists in adding an integer offset to the data fetched by the first load. If memory-side logic is added to perform such computations, it is no longer necessary to go back to the processor to compute the address of the second load, so that the overall latency of the two dependent loads can be significantly reduced. Observe again that this logic may be implemented in memory, even though we assume in our simulations that it is located in the memory controller.

Figure 1 describes both the behavior of two dependent loads (a) on a simplified standard memory hierarchy, and (b) on a memory hierarchy augmented with the *load squared* technique. (Part (c) is described in Section 3.) In the two cases, we assume two dependent load instructions, where the first one has issued a virtual address `a` corresponding to data `b`, which is used to compute a new virtual address `c` for the second load instruction. On a standard memory hierarchy, the two requests are processed as follows. In Step 1, the processor sends virtual address `a` to the first level cache and the TLB. Assuming the first load misses, physical address `a'` is sent to the memory-side logic in Step 2. The memory-side logic accesses main memory M in Step 3 and value b (along with its corresponding cache line) is sent to the cache and the processor in Step 4. The processor computes virtual address `c=b+d` in Step 5 and in Step 6 makes a second request to the memory hierarchy through Steps 7 to 9.

Figure 1(b) shows the *load squared* operation. In this example, we assume both loads miss, and we will see in the next section that the mechanism includes a predictor for deciding when both loads are most likely to miss and for applying the *load squared* technique only in this case. In Step 1, virtual address `a` is sent to the cache (miss) then to the memory; the mechanism identifies the pairs of dependent loads, and when the second load is ready to issue, it is replaced by a *load squared* which sends the address `a` of the *first* load along with the offset `d` to memory. In Step 3, the memory-side logic sends address `a` to memory, gets data `b` in Step 4, computes address `b+d` in Step 5, then immediately sends the new virtual address `c` to memory after translating it in its own TLB in Step 6. Following the idea of the hardware page walker on Itanium 2 [8], address translation is done by the memory-side logic entirely in hardware. This assumes the page table follows a predefined format, which the OS must follow. In Step 6, a fetch of `v` at address `c'` is sent to memory; that request returns to the processor in Steps 7 and 8.

Area Cost. The main area cost of the mechanism lays in the additional TLB and the adder located in the memory-side logic, and the prediction tables used to detect dependent loads which both miss (see next section). Several modifications are also necessary in the cache controller and the miss address file.

Coherence. The value of `v` is speculative since the content of memory may not be up-to-date at this step: if some of the cache levels are write-back, they may contain a dirty line that contains the most up-to-date content of address `c'`. (Even if all cache levels are write-through, the request from the memory-side logic may arrive to memory just before the copy back is complete.) To ensure memory coherence, the memory-side logic sends a read request for address `c'` to all cache levels, at Step 6b, simultaneously with the memory fetch in Step 6. If the cache read is a hit, the value in the cache is at least as recent as the value read from memory, and this value is sent by the cache to the processor (not to the memory-side logic) as the final result of the load squared. Eventually, value `v` read in Step 7 from memory is sent by the memory-side logic to the main processor. This value is propagated by the

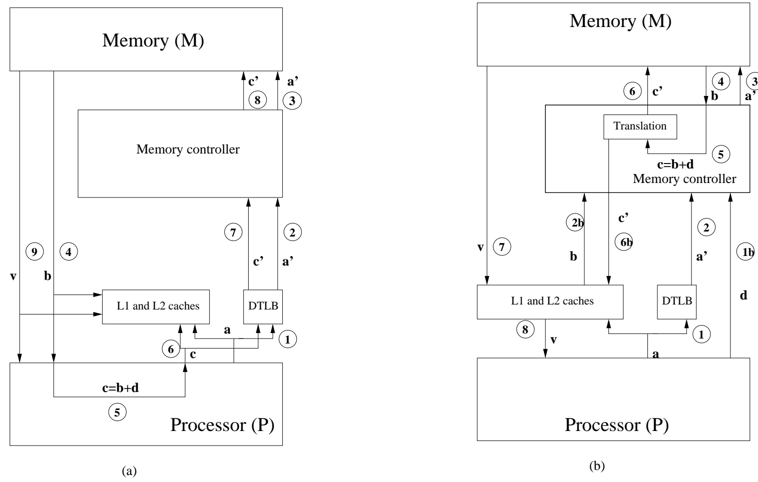


Figure 1. (a) Normal Operation (b) Load Squared Operation.

cache hierarchy and is simply ignored by the miss address file. Alternatively if the read is a miss, no further action is taken by the cache at this point (except perhaps for initiating a fill request) and the memory read initiated by the memory-side logic at Step 7 provides `v`.

Note also that a coherence issue also arises when a load squared is followed by a store. In this case, the processor cannot disambiguate the load squared from the store, even dynamically (it can only disambiguate the first load and the store), and must conservatively assume a WAR dependence. We solve this issue by enforcing the following restriction: a store following a load squared cannot be presented to the memory hierarchy before the load squared completes. Since stores are seldom on the critical path, delaying them that way is not a significant performance issue. Other solutions to this problem, which we will consider in future work, include ways to let the compiler assert the store and preceding loads squared do not alias.

Detecting a dual miss. Replacing the second load by a load squared is only worthwhile if both loads miss in all cache levels, otherwise the time spent accessing the DRAM twice may be higher than a full memory access plus a cache access in case of one hit and one miss (let alone two cache accesses in case of two hits). Therefore, we complement the *load squared* approach with a predictor for detecting when two dependent loads are likely to miss; the second load is replaced by a *load squared* only in this case.

Optimization when the first load is a miss. A load squared request is sent to memory via the existing memory hierarchy. Therefore, it only makes sense to check on the fly whether address `a` misses or hits the different caches. If the address hits any cache level, value `b` is read and sent to the memory-side logic, which directly computes `c` and fetches the corresponding value. This can be seen as an on-the-fly change of opcode, from that of a load squared instruction with arguments `a` and `d` to that of a "load squared with hit" instruction with arguments `b` and `d`.

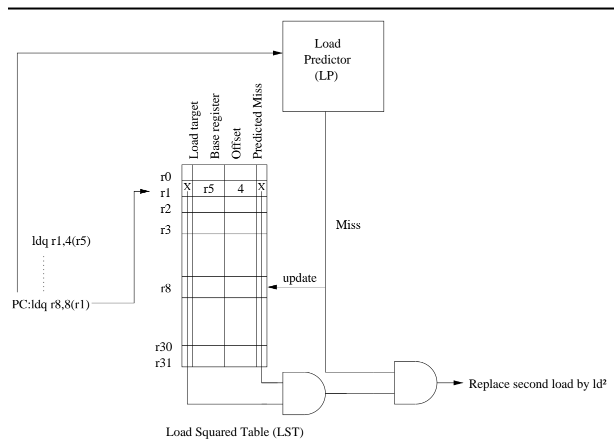
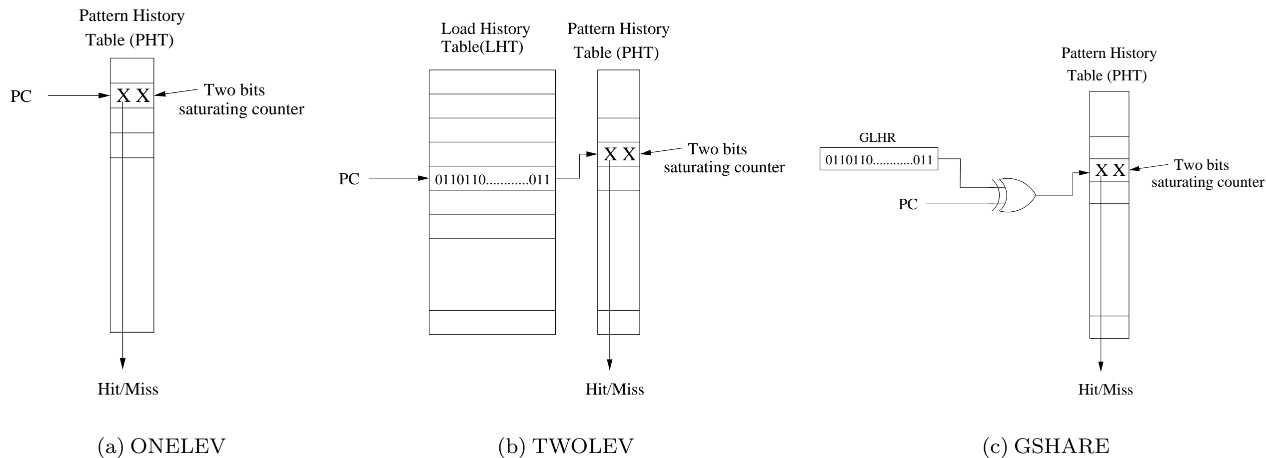


Figure 2. Predicting and Issuing Load Squared.

2.1. Detecting and Issuing Load Squared

In addition to the memory-side logic, the mechanism relies on a table, the Load Squared Table (LST), to dynamically identify chains of dependent loads, and on a predictor to guess which Load/Load chains probably result in a double miss. Note that because the Alpha instruction set provides a load instruction with offset, we consider only pairs of directly dependent loads.



(a) ONELEV (b) TWOLEV (c) GSHARE

Figure 3. Load Predictors.

The LST is shown in Figure 2, and its operation is illustrated with instructions `ldq r1,0(r5)` and `ldq r8,8(r1)`. The LST keeps track of "producer" loads: when a register is the target of a load (`r1` after the first load), it is marked as such in the LST. The base register of the load, and the value of its offset, are also recorded in the second and third column of the LST, respectively. When a candidate "consumer" load is decoded, the entry in the LST corresponding to its base register is read. If the base register is marked as the target of a load, then a Load/Load chain is detected. Of course, any intervening instruction (other than a load) that modifies a load target register causes the load target bit of the LST to be cleared. Note also that this mechanism spots Load/Load chains in distinct, and possibly distant, basic blocks.

Detecting Load/Load chains is the first condition for replacement by a load squared; the second is to predict a double miss with high probability. Each time a load instruction is decoded, it is predicted as a hit or a miss (see predictor below), and the predicted miss bit of the LST entry is updated accordingly. This way, when the second load is decoded (`ldq r8,8(r1)` in the example), the hit/miss prediction for the producing load is available. Separately, the consumer load is itself predicted as a hit or a miss; if it is predicted missing, and it is fed by a load, and that load was also predicted missing, then the second load is replaced by a load squared. The base address (`a` in Figure 1) of the load squared is computed from the second and third columns of the LST. The offset of the load squared is that of the second load. For instance, `ldq r8,8(r1)` is replaced by a load squared whose target register is `r8` and input operands are immediate 8 and the target address of the first load, i.e., `4(r5)`.

The Load Predictor. We have found that it is possible to relatively accurately predict whether pairs of dependent loads will miss in all cache levels before the second load is issued, and we have come up with a hardware mechanism to implement a prediction strategy. The strategy is similar to history-based branch predictors [16, 19, 12]: prediction is based on both the hit/miss behavior of previous loads and on the target load address. We studied three types of load predictors: the one-level load predictor (ONELEV), the two-level load predictor (TWOLEV) and the gshare (GSHARE) load predictor, see Fig. 3. A predictor is read when the load is decoded and updated when it is committed.

The one-level load predictor (ONELEV) shown in 3(a) consists in one table of 2-bit saturating counters, the Pattern History Table (PHT). The PHT is indexed by the PC of the load. When the load misses, the counter is incremented, and decremented otherwise. Therefore, the predictor predicts the load will miss if it missed at least on the last two occurrences. The one-level load predictor behaves particularly well with loops that access chained data structures not yet present in cache (cold misses).

The two-level load predictor (TWOLEV) further exploits the historical behavior of each load. It includes a Load History Table (LHT) that records the hit/miss behavior history of each load, see Figure 3(b). The LHT is indexed by the PC of each load, the corresponding history of the load indexes the PHT.

The gshare load predictor (GSHARE) exploits the correlation between several previous loads by maintain-

Fetch width	8
Issue / Decode / Commit width	8
RUU size (Inst. window- ROB)	128
LSQ size	32
ExeUnits	4 IALU, 1 IMULT, 4 FPALU, 1 FPMULT
Branch	Combined, 4K entries bimodal, and 2 level Gap predictor, 8K 2nd level entries, 14 history wide, 1K meta-table size 7 cycle BR resolution
L1 DCache	16kB, 4 ways, 1 cycle
L1 ICache	16kB, 1 way, 1 cycle
L2 Unified Cache	256kB, 4 ways, 6 cycles

Table 1. Baseline configuration of the processor.

ing a global behavior history in the Global Load History Register (GLHR). The GLHR is XORed with the PC to index the Pattern History Table as shown in 3(c). We further study the performance of each predictor in Section 4.

3. Experimental Framework

We simulated an 8-way superscalar processor using the SimpleScalar 3.0 Alpha toolset [3]; Table 1 shows our baseline simulation environment.

We modified the system bus to operate at 1/5 the processor frequency. Also, to properly measure the effect of load squared, we subdivided the processor-to-memory round-trip latency into: 60 cycles for the processor-to-memory-side logic latency, 10 cycles for the memory-side logic latency, 20 cycles for logic-to-memory latency, 80 cycles for a DRAM access, 20 cycles from the memory back to the memory-side logic, and 60 cycles from the logic back to the processor. The TLBs in the memory-side logic and in the CPU have 32 entries. The cost of a TLB miss in the memory-side logic is 80 cycles, and 30 cycles for the CPU's TLB (because a CPU TLB miss can be serviced by the cache). We also assume a 60-cycle latency for direct transfer from the memory to the processor, so that a traditional read to memory (without TLB miss) has a total latency of 240 cycles.

The latency of a load squared depends on whether address `a'` was a cache hit (Steps 2 or 2b of Fig. 1), and on whether address `c'` is a hit (at Step 6b). Assuming the memory-side TLB hits, the latency can therefore be one of the following: 60+10+60 = 130 cycles (two hits), 60+10+20+80+60 = 230 (`a'` hits), 60+10+20+80+20+60 = 250 (`c'` hits) and 60+10+20+80+20+20+80+60 = 350 (no hits). As said earlier, any miss in the memory-side TLB adds 80 cycles.

Figure 4. Potential Loads Squared and Miss/Miss Occurrences.

Predictor configuration. We used a Pattern History Table (PHT) of 4K entries for the ONELEV load predictor, a 4K-entry Load History Table (LHT) for the TWOLEV load predictor. The PHT for both the TWOLEV and the GSHARE load predictors are indexed using a 14-bit history length (XORed with the PC for the GSHARE predictor)

Benchmarks. We simulated all SpecInt and SpecFP benchmarks [7] as well as 9 of the Olden [13] benchmark suite. We simulated 100 Million instructions for each benchmark, and we fast-forwarded 2 billion instructions for the SpecInt and the SpecFP benchmarks.

4. Performance Evaluation

4.1. Load Squared Potential

Figure 4 shows the percentage of loads that are directly dependent on other loads (Load/Load). On average, 12.5% and up to 47% (`ammp`) of all executed loads are directly dependent on another load. The Miss/Miss columns of Figure 4 show the percentage loads that are fed by another load, and such that both miss at all cache levels. Those Miss/Miss loads are those that should be replaced by loads squared. While the percentage of Miss/Miss occurrences is relatively low, their effect on the performance is significant because of the high latency of memory accesses.

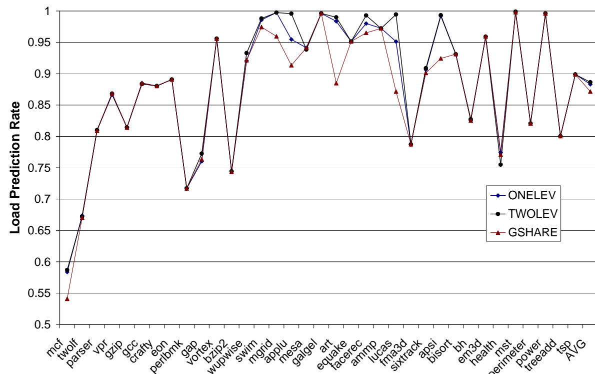


Figure 5. Load Prediction Rate.

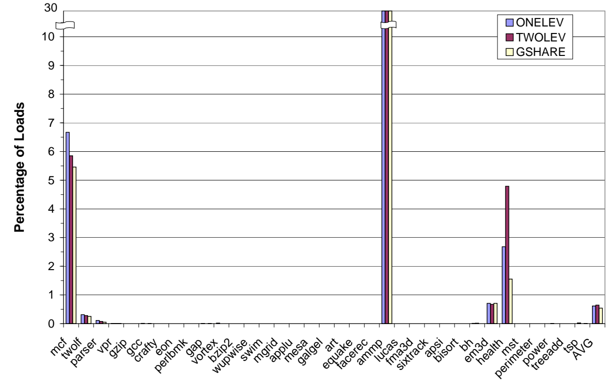


Figure 7. Percentage of Load Squared.

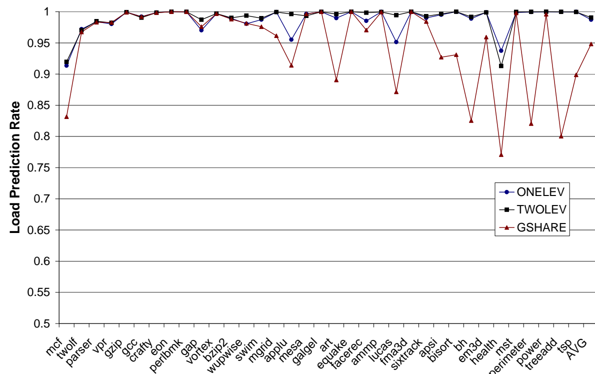


Figure 6. Non-speculative Load Prediction Rate.

Note also that we did not observe significant differences in TLB miss ratios with and without loads squared.

4.2. Efficiency of Load Predictors

Because the *load squared* mechanism targets pairs of loads that both miss in the cache hierarchy, replacing a load by a load squared when one of the two loads (or both) hits may negatively affect performance.

On the other hand, failing to catch appropriate Miss/Miss pairs naturally means achieving only a fraction of the potential performance improvement. Therefore, the load squared mechanism is fairly sensitive to the efficiency of the load miss prediction. Figure 5 shows the load prediction rates of the three types of predictors discussed in Section 2.1, i.e., how often a dynamic instance of a load is correctly predicted as hitting or missing (the average load prediction rate is 87%). We notice that the TWOLEV and the ONELEV load predictors exhibit a better prediction behavior than GSHARE for many benchmarks. The relative low prediction rate of the GSHARE load predictor suggests that loads behavior have little correlation.

Interestingly, control speculation has a strong impact on the load prediction rates. While Figure 5 gives the rate for all loads, Figure 6 only considers nonspeculated loads, or loads that were on a correctly predicted path. (Both figures are on the same scale.) Clearly, the accuracy of our Miss/Miss predictors is then much better, which indicates a possible correlation between branch prediction and Miss/Miss prediction. How to leverage this observation to improve our Miss/Miss predictors is left for future work.

Figure 7 shows the percentage of loads replaced by loads squared, for each predictor. This figure should be compared with Figure 4. Miss/Miss sequences are numerous in `ammp`, `mcf` and `health`; a few occurrences also occur in `twolf` and `em3d`.

4.3. Performance Results

Figure 8 shows the speedups achieved with the load squared mechanism. Our approach significantly improves the performance of several benchmarks (5% for `twolf`, more than 7% for `em3d` and about 50% for `ammp`). Importantly, performance is never degraded.

We also notice that the speed-up for `mcf` is smaller than we could have hoped for from looking at Figure 7. Comparing Fig. 4 with Fig. 7, a possible explanation is that our predictors may be too aggressive on this benchmark. However, a deeper investigation shows that the root cause may be more complex. Consider Table 2, where each row provides data for a specific static load in the main loop of procedure `refresh_potential`, which accounts of 35% of the total execution time. The loads considered here are the most frequent loads that are fed at least once by an-

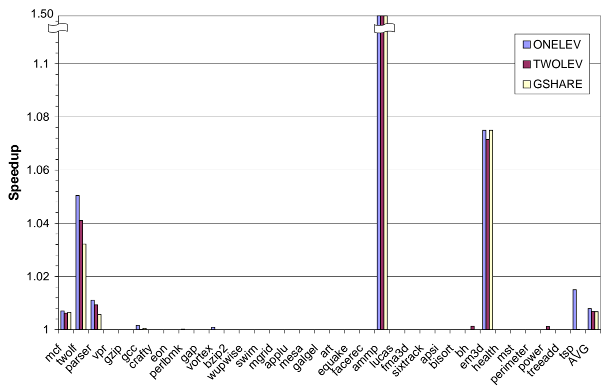


Figure 8. Speedup Obtained with the Load Squared Mechanism.

other load (control flow may imply that not all dynamic instances of a load are fed by the same instruction). Note that there are four back-to-back loads at addresses 7980 to 798c. Column 1 gives the PC of each load, and Column 2 their dynamic counts. Column 3 says how many instances were fed by a load. Columns 4 and 5 indicate how many instances missed all cache levels, and how many both missed and were fed by a load that missed, respectively. Clearly, Column 5 shows our sweet spot. Column 6 gives the average latency, in cycles, elapsed between the issuing of the producer load and the write-back of the current load, when load squared is turned off. When it is turned on, Column 7 indicates how many instances were converted to loads squared using the ONELEV predictor, and Column 8 gives the resulting average latency, in cycles, of the load squared.

Table 2 first confirms that caches do their work well on these loads: most Load/Load's are not Miss/Miss. The ONELEV predictor sees this correctly and, where double misses are rare, it seldom converts the 2nd load into a load squared. This is not true, however, for the load at address ending in 7988: most of its instances miss and are fed by a load that also misses. Also, most of them are converted into loads squared – again, a bit too many. Nevertheless, the result on the latency of this specific load is dramatic.

5. Related Work

Linked Data Structures Traversal. Several software-based prefetching techniques [10, 11, 9] proposed compiler optimizations for inserting instructions to early-prefetch linked data structures. Roth et al. [14] proposed a hardware mechanism called *dependence-based prefetching* that dynamically identifies loads that access linked data structures, collects them and executes them speculatively in a prefetch engine to slip ahead of the execution.

Beside the speculative nature of dependence-based prefetching, these mechanisms assume that there are enough non-load instructions for the prefetching engine to run ahead of the execution; moreover, dependence-based prefetching has the same caveats as all prefetching schemes, it does not reduce the overall latency of indirect accesses but only hides part of it. Also, the same authors later proposed [15] to add *jump pointers* that are used to prefetch future nodes in a linked list; the approach relies on making pointers explicit in data structures.

Cooksey et al. proposed [6] a content-directed data prefetching mechanism that searches for virtual addresses in data fetched from memory. By then prefetching these addresses, the mechanism implements a form of pointer chasing. However, because the mechanism relies on guessing which addresses are pointer addresses, the number of useless prefetches or the number of missed prefetching opportunities can be significant. For example, we observed in `ammp` that, when the first `next` address is read in a linked list of relatively "large" nodes (larger then the prefetched line size), the prefetcher will only partially prefetch the node, and since the `next` pointer is defined at the end of the node, the prefetcher fails to get the `next` address. A similar approach [5] proposes to store pointer addresses in a cache rather than guessing them on-the-fly.

Bekerman et al. [2] proposed to enhance a stride address prefetcher with a correlated load-address predictor to predict linked data structures addresses. Again, this mechanism does not efficiently handle long load dependencies, if the amount of work to overlap is not sufficient. Besides, correlated load address predictor needs prior traversal of a data structure in order to properly learn the correlation. Hence, the first traversal of a data structure, which is more likely to miss in the cache hierarchy, may not benefit from the approach.

Solihin et al. [17] proposed a User-Level Memory Thread (ULMT) that can be located either in a memory-side logic or integrated in the DRAM. The ULMT is a correlated prefetcher that sends prefetched data to the L2 cache of the main processor. Also, Yang et al. [18] proposed to attach a prefetcher to each level of the memory hierarchy to push data to the processor rather than pulling data from the memory.

Memory-side logic. FlexRAM [1] is a distributed architecture where compute nodes are attached to local memories. It is more powerful and more complex than what we suggest. It also introduces a new programming model, which we don't.

PC	Occur. count	Fed by a LD	Miss count	Both loads missed	Avg Lat (w/o LD2)	LD2 Count	Avg Lat (w/ LD2)
0x120007974	2626041	740306	314269	62	553.163	0	549.284
0x120007980	2614229	734581	2231030	4830	555.701	51	554.132
0x120007984	2610730	733853	1418065	769	551.326	16	548.146
0x120007988	2598325	2595990	2597224	2154484	517.025	2467031	371.110
0x12000798c	2597455	2589596	48916	392	283.307	0	262.348
0x1200079c0	2621919	743407	567721	1350	551.763	24	550.839

Table 2. Stats for 181.mcf.

Impulse [4] is a memory controller equipped with address translation hardware. Its architecture is therefore close to our work, but its goal vastly different: Impulse allows the controller to access shadow addresses, i.e., legitimate addresses that are not backed by DRAM. This feature in turn allows applications and/or the compiler to deploy optimizations like mapping noncontiguous addresses to contiguous shadow addresses, thereby improving spatial locality.

6. Conclusions and Future Work

We have shown that it is possible to tackle the long latency of even very irregular indirect memory accesses using a simple memory-side logic, and processor-based hardware add-ons. We now want to improve the detection of missing dependent loads by taking advantage of clustered performance events, since our experiments and recent litterature suggest significant correlation between branch and cache events.

References

[1] Flexram: Toward an advanced intelligent memory system. In *Proc. IEEE Int'l Conf. on Comp. Design*, page 192, 1999.

[2] M. Bekerman et al. Correlated load-address predictors. In *Proc. 26th Int'l Symp. on Comp. Arch.*, pages 54–63, 1999.

[3] D. Burger, T. M. Austin, and S. Bennett. Evaluating future microprocessors: The SimpleScalar tool set. Technical Report CS-TR-1996-1308, 1996.

[4] J. B. Carter et al. Impulse: Building a smarter memory controller. In *HPCA*, pages 70–79, 1999.

[5] J. Collins, S. Sair, B. Calder, and D. M. Tullsen. Pointer cache assisted prefetching. In *Proc. 35th ACM/IEEE Int'l Symp. on Microarchitecture*, pages 62–73, 2002.

[6] R. Cooksey, S. Jourdan, and D. Grunwald. A stateless, content-directed data prefetching mechanism. In *Proc. 10th Int'l Conf on Arch. Support for Prog. Lang. and Op. Sys.*, pages 279–290, 2002.

[7] J. Henning. Spec cpu2000: measuring cpu performance in the new millennium. *IEEE Computer*, 33(7):28–35, 2000.

[8] Intel Corp. Intel Itanium 2 Processor Reference Manual.

[9] M. Karlsson, F. Dahlgren, and P. Stenstrom. A prefetching technique for irregular accesses to linked data structures. In *Proc. 6th Int'l Symp. on High-Perf. Comp. Arch. (HPCA'6)*, pages 206–217, 2000.

[10] M. H. Lipasti et al. Spaid: software prefetching in pointer- and call-intensive environments. In *Proceedings of the 28th annual international symposium on Microarchitecture*, pages 231–236, 1995.

[11] C.-K. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. In *Proc. 7th Int'l Conf. on Arch. Support for Prog. Lang. and Op. Sys.*, pages 222–233, 1996.

[12] S. McFarling. Combining branch predictors. Technical Note TN-36, Digital WRL, june 1993.

[13] A. Rogers, M. C. Carlisle, J. H. Reppy, and L. J. Hendren. Supporting dynamic data structures on distributed-memory machines. *ACM Trans. on Prog. Lang. and Sys. (TOPLAS)*, 17(2):233–263, 1995.

[14] A. Roth, A. Moshovos, and G. S. Sohi. Dependence based prefetching for linked data structures. In *Proc. 8th Int'l Conf on Arch. Support for Prog. Lang. and Op. Sys.*, pages 115–126, 1998.

[15] A. Roth and G. S. Sohi. Effective jump-pointer prefetching for linked data structures. In *Proc. 26th Int'l Symp. on Comp. Arch.*, pages 111–121, 1999.

[16] J. E. Smith. A study of branch prediction strategies. In *Proc. 8th Symp. on Comp. Arch.*, pages 135–148, 1981.

[17] Y. Solihin, J. Lee, and J. Torrellas. Using a user-level memory thread for correlation prefetching. In *Proc. 29th Int'l Symp. on Comp. Arch.*, pages 171–182, 2002.

[18] C.-L. Yang and A. R. Lebeck. Push vs. pull: data movement for linked data structures. In *Proc. 14th Int'l Conf. on Supercomputing*, pages 176–186, 2000.

[19] T.-Y. Yeh and Y. N. Patt. Two-level adaptive training branch prediction. In *Proc. 24th Int'l Symp. on Microarchitecture*, pages 51–61, 1991.