MLP Exploitation with Seamless Preload

Zhen Yang, Xudong Shi, Feiqi Su and Jih-Kwon Peir

Computer and Information Science and Engineering
University of Florida
Gainesville, FL 32611, USA
{zhyang, xushi, fsu, peir}@cise.ufl.edu

***Abstract*—Modern out-of-order processors with non-blocking caches exploit Memory-Level Parallelism (MLP) by overlapping cache misses in a wide instruction window. Exploitation of MLP, however, can be limited due to long-latency operations in producing the base address of a cache miss load. When the parent instruction is also a cache miss load, a serialization of the two loads must be enforced to satisfy the load-load data dependence. In this paper, we describe a special Preload that is issued in place of the dependent load without waiting for the parent load, thus effectively overlapping the two loads. The Preload provides necessary information for the memory controller to calculate the correct memory address upon the availability of the parent's data to eliminate any interconnect delay between the two loads. Performance evaluations based on SPEC2000 and Olden applications show that significant speedups ranging 2-50% with an average of 16% are achievable by using the aggressive MLP exploitation method.**

***Index Terms*—Instruction/Issue Window, Memory-Level Parallelism, Pointer-Chasing Loads, Data Prefetching.**

1 Introduction

Memory latency presents a classical performance bottleneck. Studies show that for SPEC2000 benchmarks running on modern microprocessors, over half of the time is spent on stalling loads that miss the second-level (L2) cache [15, 26]. This problem will be exacerbated in future processors with widening gaps and increasing demands between processor and memory [24]. It is essential to exploit *Memory-Level Parallelism (MLP)* [7] by overlapping multiple cache misses in a wide instruction window. Exploitation of MLP, however, can be limited due to a load that depends on another load to produce the base address (referred as *load-load dependences*). A cache miss of the parent load forces sequential executions of the two loads. One typical example is the pointer-chasing problem in many applications with linked data structures, where accessing the successor node cannot start until the pointer is available, possibly from memory. Similarly, indirect accesses to large array structures may face the same problem when both address and data accesses encounter cache misses.

There have been several prefetching techniques to reduce penalties on consecutive cache misses involved loads with tight load-load dependences [17, 16, 19, 27, 25, 4, 9, 8, 28]. For example, the dependence-based prefetching scheme [19] dynamically identifies nearby pointer loads with tight dependences and packs them together for fast traversal and prefetching. With software help, the push-pull scheme [27, 28] places these tightly dependent pointer loads in a prefetcher closer to the memory to reduce the interconnect delay. A similar approach has been presented in [11]. The content-aware data prefetcher [9] identifies potential pointers by examining word-based content of a missed data block. The identified pointers are used to initiate prefetching of the successor nodes. Using the same mechanism to identify pointer loads, the pointer-cache approach [8] builds a correlation history between heap pointers and the addresses of the heap objects they point to. A prefetch is issued when a pointer load misses the data cache, but hits the pointer cache.

In this paper, we introduce a new approach to exploit MLP among nearby loads even when load-load dependences exist among them. After dispatch, if the base register of a load is not ready due to an early cache miss load, a special *preload* (referred as *P-load*) is issued in place of the dependent load. The P-load instructs the memory controller to calculate the needed address once the parent load's data is available from the DRAM array. The inherent interconnect delay in memory access can thus be overlapped regardless the location of the memory controller [2]. When executing pointer-chasing loads, a sequence of P-loads can be initiated according to the dispatching speed of these loads.

The proposed P-load makes three unique contributions. First, in contrast to the existing methods, it does not require any special predictors and/or any software-inserted prefetching hints. Instead, the P-load scheme issues the dependent load early following the instruction stream. Second, the P-load exploits more MLP from a larger instruction window without the need to enlarge the critical issuing window [1]. Third, an enhanced memory controller with proper processing power is introduced that can share certain computations with the processor. Performance evaluations based on SPEC2000 [23] and Olden [18] applications on modified SimpleScalar simulation tools [3] show that significant speedups of 2-50% are achievable with an average about 16%.

This paper is organized as follows. Section 2 demonstrates performance loss due to missing MLP opportunities. Section 3 provides a detailed description of the proposed P-load scheme. Section 4 describes the evaluation methodology. This is followed by performance evaluations and comparisons in

Section 5. Related works are summarized in Section 6. A brief conclusion is given in Section 7.

2 Missing MLP Opportunities

Overlapping cache misses can reduce the performance loss from long-latency memory operations. A data dependence that exists between a load and an early instruction may stall the load from issuing. In this section, we will show the performance loss due to existence of such data dependences in real applications.

An idealized scheme to exploit MLP is simulated. This ideal MLP exploitation assumes no delay of issuing any cache miss load after it is dispatched regardless of whether the base register is ready or which instruction, if exists, the base register depends on. Nine workloads, *Mcf*, *Twolf*, *Vpr*, *Gcc-200*, *Parser*, and *Gcc-scilab* from SPEC2000 integer applications, and *Health*, *Mst*, and *Em3d* from Olden applications [18] are selected for the experiment because of their high L2 miss rates. An Alpha 21264-like processor with 1MB L2 cache is simulated. Detailed descriptions of the simulation model will be given in Section 4.

Fig. 1 shows the measured MLP of the base and the ideal schemes where the MLP is defined as the average number of memory requests during the period when there is at least one outstanding memory request [7]. The base MLP is obtained when data dependences are correctly enforced within the instruction window. As observed, there are huge gaps between the base and the ideal MLPs, especially for *Mcf*, *Gcc-200*, *Parser*, *Gcc-scilab*, *Health*, and *Mst*. Thus, significant MLP improvement can be expected, when the delay of issuing cache misses can be reduced.

3 Overlapping Cache Misses with P-loads

In this section, we describe the P-load scheme using an example function *Refresh_potential* from *Mcf* (Fig. 2). *Refresh_potential* is invoked frequently to refresh a huge tree structure that exceeds 4MB. The tree is initialized with a regular stride pattern among adjacent nodes on the traversal path. However, the tree structure is slightly modified between two adjacent visits. After a period of time, the address pattern on the traversal path becomes irregular and is difficult to predict accurately.

The function traverses a structure with three travesal links: *child*, *pred*, and *sibling* (highlighted in *italics*), and accesses basic records with a data link, *basic_arc*. In the first inner *while* loop, the execution traverses down the path through the link: *node→child*. With accurate branch predictions, several iterations of the *while* loop can be initiated in a wide instruction window. The recurrent instruction, *node = node→child* that advances the pointer to the next node, becomes a potential bottleneck since accesses of the records in the next node must wait until the pointer (base address) of the node is available. As shown in Fig. 3 (a), four consecutive *node = node→child* must be executed sequentially. In the case of a cache miss, each of them encounters delays in sending the request, accessing the

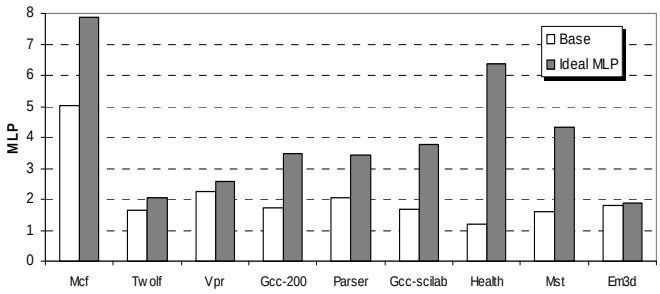


Fig. 1. Gaps between base and ideal MLP exploitations

```
Long refresh_potential (network_t *net)
{
 .....
  tmp = node = root->child;
  while (node != root) {
    while (node) {
      if (node->orientation == UP)
        node->potential = node->basic_arc->cost
                        + node->pred->potential;
      else {
        node->potential = node->pred->potential
                        - node->basic_arc->cost;
        checksum++; }
      tmp = node;
      node = node->child;
    }
    node = tmp;
    while (node->pred) {
      tmp = node->sibling;
      if (tmp) {
        node = tmp;
        break; }
      else node = node->pred;
    }
  }
  return checksum;
}
```

Fig. 2. Example tree-traversal function from *Mcf*

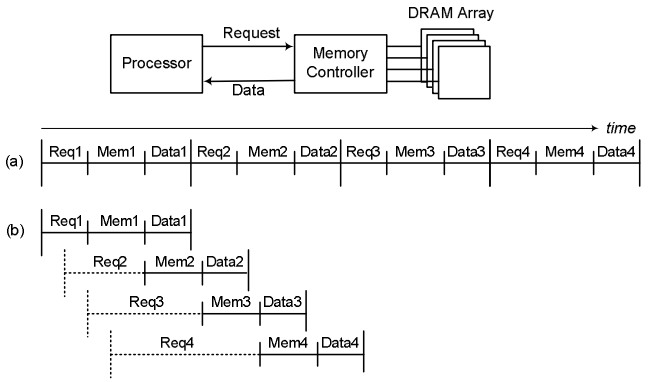


Fig. 3. Pointer Chasing: (a) Sequential accesses; (b) Pipeline using P-loads

DRAM array, and receiving the data. These non-overlapped long latency memory accesses can congest the instruction issue window and stall the processor.

Instruction Window

ID	Instr.	Request		
		Type	ID	Disp
101	lw $v0,28($a0)	Load [28($a0)]	—	—
102	bne $v0,$a3,L1			
103	lw $v0,32($a0)	(partial hit)		
104	lw $v1,8($a0)	(partial hit)		
105	lw $v0,16($v0)	P-load [addr(103)]	105	16
106	lw $v1,44($v1)	P-load [addr(104)]	106	44
107	addu $v0,$v0,$v1			
108	J L2			
109	sw $v0,44($a0)			
110	addu $v0,$0,$a0			
111	lw $a0,12($a0)	(partial hit)		
112	bne $a0,$0,L0			
113	lw $v0,28($a0)	P-load [addr(111)]	113	28
114	lw $v0,32($a0)	P-load [addr(111)]	114	32
115	lw $v1,8($a0)	P-load [addr(111)]	115	8
116	lw $v0,16($v0)	P-load [p-id(114)]	116	16
117	lw $v1,44($v1)	P-load [p-id(115)]	117	44
118	lw $a0,12($a0)	P-load [addr(111)]	118	12

Memory Request Window

ID	Link	Offset	Address	Disp
New	—	—	[28($a0)]	—
105	New	32($a0)		16
106	New	8($a0)		44
113	New	12($a0)		28
114	New	12($a0)		32
115	New	12($a0)		8
116	114	32($a0)		16
117	115	8($a0)		44
118	—	—	[12($a0)]*	—

* Assume New removed, 118 uses address [12($a0)] to fetch DRAM or P-load Buffer

Note, thick lines divide iterations

Fig. 4. Example of issuing P-loads seamlessly without load address

On the other hand, the proposed P-load can effectively overlap the interconnect delay in sending/receiving data as shown in Fig. 3 (b). In the following subsections, detailed descriptions of identifying and issuing the P-loads are given first, and then the design of the memory controller. Several issues and enhancement related to the P-loads will also be discussed.

3.1 Issuing P-Loads

We will describe P-loads issuing and execution within the *instruction window* and the *memory request window* (Fig. 4) by walking through the first inner while loop of *refresh-potential* from *Mcf* (Fig. 2). Assume the first load, *lw $v0,28($a0),* is a miss and is issued normally. The second and third loads encounter a partial hit to the same line as the first load, thus no memory request is issued. After the fourth load, *lw $v0,16($v0),* is dispatched, a search through the current instruction window finds a dependence on the second load, *lw $v0,32($a0)*. Normally, the fourth load must stall. In the proposed method, however, a special P-load will be inserted into a small *P-load issue window.* When the hit/miss of the parent load is known, an associative search for dependent loads in the *P-load issue window* is performed. All dependent P-loads are either ready to be issued (if the parent load is a miss), or canceled (if the parent is a hit). The P-load consists of the address of the parent load, the displacement, and a unique instruction ID to instruct the memory controller for calculating the address and fetching the correct block. Details of the memory controller will be given in Section 3.2. The fifth load is similar to the fourth. The sixth load, *lw $a0,12($a0),* advances the pointer and is also a partial hit to the first load.

The instruction window moves to the second iteration with a correct branch prediction. The first three loads in the second iteration all depend on *lw $a0,12($a0)* in the previous iteration. Three P-loads are issued accordingly using the address of the parent load. The fourth and fifth loads, however, depend on early loads that are themselves also P-loads without addresses. In this case, special P-loads with an invalid address are issued. The parent load IDs (*p-id*), 114 and 115 for the fourth and fifth loads respectively, are encoded in the address fields to instruct the memory controller for obtaining correct base addresses. This process continues across the entire instruction window to issue a sequence of P-loads seamlessly.

The P-load does not occupy a separate location in the recorder buffer, nor does it keep a record in the MSHR. Similar to other memory-side prefetchers [22], the returned data block must come with the address. The processor searches and satisfies any existing memory requests located in the MSHR upon receiving a P-load block from memory. The block is then put into cache if it is not already there. Searching the MSHR is needed since the P-load cannot stop other requests that target the same block. The regular load, from which a P-load was initiated, will be issued normally when the base register is ready from its parent load.

3.2 Memory Controller Design

Fig. 5 illustrates the basic design of the memory controller. DRAM accesses are processed and issued in the *memory*

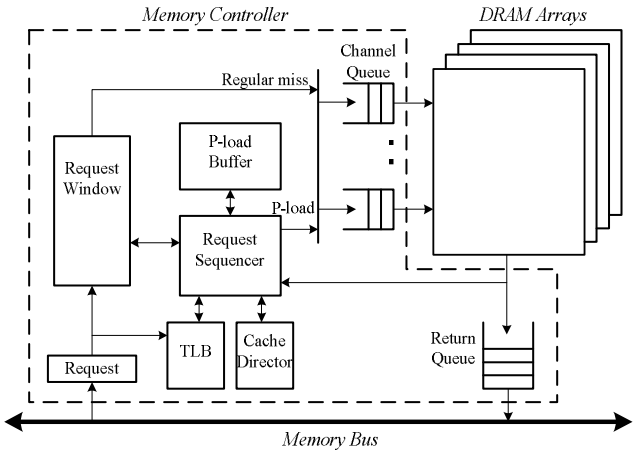


Fig. 5. The basic design of the memory controller

request window similar to the out-of-order execution in processor's instruction window. The memory *address*, the *offset* for the base address, the *displacement* value for computing the target block address, and the dependence *link*, are recorded for each request in their arriving order. For a normal cache miss, the address and a unique ID assigned by the *request sequencer* are recorded. Such miss requests will access the DRAM array without delay as soon as the target DRAM channel is open. The regular miss may be merged with an early active P-load that targets the same block to achieve reduced penalties.

Two different procedures are applied when a P-load arrives. First, when the P-load comes with a valid address, the block address is used to search for any existing memory requests. Upon a match, a dependence link is established; the *offset* within the block is used to access the correct word from the parent's data block without the need to access the DRAM. In the case of no match, the address that comes with the P-load is used to access the DRAM as illustrated by request 118 assuming that the first request has been removed from the memory window. Second, when the P-load comes without a valid address, the dependence link encoded in the address field is extracted and saved in the corresponding entry as shown by requests 116 and 117 (Fig. 4). In this case, the correct base addresses can be obtained from 116's and 117's parent requests, 114 and 115, respectively. The P-load is dropped if the parent P-load is no longer in the request window.

Once a memory block is fetched, all dependent P-loads will be woken up. For example, the availability of the *New* block will trigger P-loads 105, 106, 113, 114, and 115 as shown in Fig. 4. The target word in the block can be retrieved and forwarded to the dependent loads. The memory address of the dependent P-load is then calculated by adding the target word (base address) with the displacement value. The P-load block is fetched if the block has not been in the request window. The fetched P-load block in turn triggers its dependent P-loads. A memory request will be removed from the window after the data block is sent.

3.3 Issues and Enhancement

There are many essential issues that need to be resolved for implementing the P-load efficiently.

Maintaining Base Register Identity: The base register of a qualified P-load may experience renaming or constant increment/decrement after the parent load. These indirect dependences can be easily identified and established by proper adjustment to the displacement value of the P-load.

Address Translation at Memory Controller: The memory controller must perform virtual to physical address translations for the P-load in order to access the physical memory. A shadowed TLB can be maintained in the memory controller for this purpose (Fig. 5). The processor issues a TLB update to the memory controller whenever a TLB miss occurs and the new address translation is available. The TLB consistency can be handled similarly to that in a multiprocessor environment. A P-load is simply dropped upon a TLB miss.

Reducing Excessive Memory Requests: Since P-load is issued without the memory address, it may generate unnecessary memory traffic if the target block is already in cache, or when multiple requests address to the same data block. Three approaches are considered in this paper. First, when a regular miss request arrives, all outstanding P-loads are searched. In the case of a match, the P-load is changed to a regular miss for saving variable delays. Second, a small P-load buffer (Fig. 5) buffers the data blocks for recent P-loads. A fast access to the buffer occurs when the requested block is located in the buffer. Third, a topologically equivalent cache directory of the lowest cache level is maintained to predict cache hit/miss for filtering the returned blocks. By capturing cache misses, P-loads, and dirty-block writebacks, the memory-side cache directory can predict cache hits accurately.

Inconsistent Data Blocks between Caches and Memory: Similar to other memory-side prefetching techniques, P-loads fetch the blocks without knowing whether the blocks are already located in cache. It is possible to fetch a stale copy if the block is located in caches in the modified state. In general, the staled copy is likely to be dropped either by a cache-hit prediction or by searching through the directory before updating the cache. However, in a rather rare case when a modified block is writeback to the memory, this modified block must be detected against outstanding P-loads to avoid fetching the stale data.

4 Evaluation Methodology

We modified the SimpleScalar simulator to model an 8-wide superscalar, out-of-order processor with Alpha 21264-like pipeline stages [14]. Important simulation parameters are summarized in Table 1.

Nine workloads, *Mcf, Twolf, Vpr, Gcc-200, Parser,* and *Gcc-scilab* from SPEC2000 integer applications, and *Health, Mst,* and *Em3d* from Olden applications are selected because of high L2 miss rates as ordered according to their appearances. Pre-compiled little-endian Alpha ISA binaries are downloaded

Table 1. Simulation parameters

Processor
Fetch/Decode/Issue/Commit Width: 8
Instruction Fetch Queue: 8
Branch Predictor: 64K-entry G-share, 4K-entry BTB
Mis-Prediction Penalty: 10 cycles
RUU/LSQ size: 512/512
Inst./P-load Issue Window: 32/16
Processor TLB: 2K-entry, 8-way
Integer ALU: 6 ALU (1 cycle); 2 Mult/Div: Mult (3 cycles), Div (20 cycles)
FP ALU: 4 ALU (2 cycles); 2 Mult/Div/Sqrt: Mult (4 cycles), Div (12 cycles), Sqrt (24 cycles)
Memory System
L1 Inst./Data Cache: 64KB, 4-way, 64B Line, 2 cycles
L1 Data Cache Port: 4 read/write port
L2 Cache: 1MB, 8-way, 64B Line, 15 cycles
L1/L2 MSHRs: 16/16
Req./DRAM/Data Latency: 80/160/80
Memory Channel: 4 with line-based interleaved
Memory Request Window: 32
Channel/Return Queue: 8/8
P-load Buffer: 16-entry, fully associative
Memory TLB: 2K-entry, 8-way
Cache-Hit Prediction: 8-way, 16K-entry (same as L2)

from [21]. We follow the studies done in [20] to skip certain instructions, warm up caches and other system components with 100 million instructions, and then collect statistics from the next 500 million instructions.

A processor-side *stride* prefetcher is included in all simulated models [10]. To demonstrate the performance advantage of the P-load, the historyless *content-aware* data prefetcher [9] is also simulated. We search exhaustively to determine the *width* (number of adjacent blocks) and the *depth* (level of prefetching) of the prefetcher for best performance improvement. Two configurations are selected. In the limited option (*Content-limit; width=1, depth=1*), a single block is prefetched for each identified pointer from searching a missed data block, i.e. both width and depth are equal to 1. In the best-performance option (*Content-best; width=3, depth=4*), three adjacent blocks starting from the target block of each identified pointer are fetched. The prefetched block initiates content-aware prefetching up to the fourth level. Other prefetchers are excluded due to the need of huge history information and/or software prefetching help.

5 Performance Results

5.1 Speedup Comparison

Fig. 6 summarizes the IPCs and the average memory access times for the *base* model, the content-aware prefetching (*Content-limit* and *Content-best*) and the P-load schemes

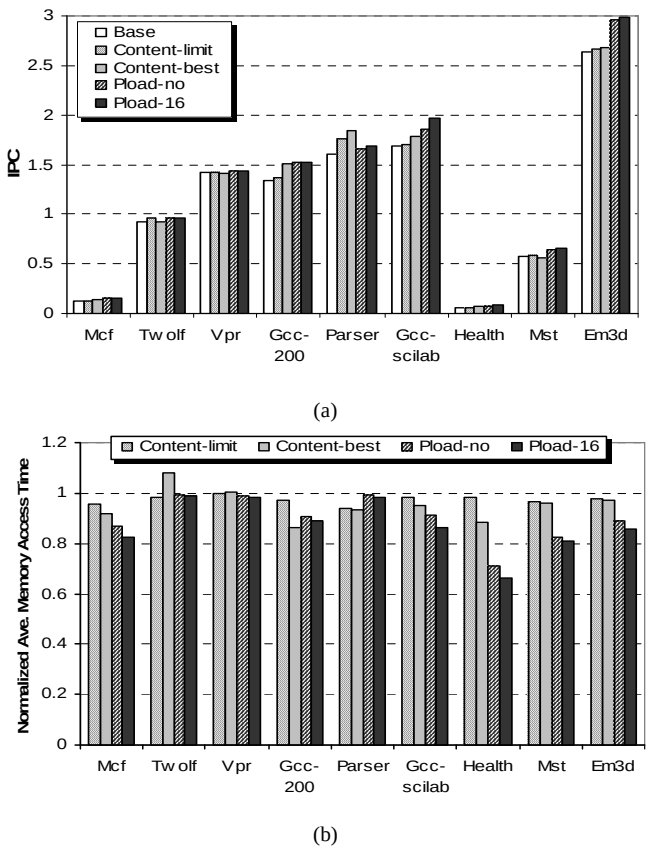


Fig. 6. Performance comparisons: (a) IPCs; (b) Average memory access time

without (*Pload-no*) and with (*Pload-16*) a 16-entry P-load buffer. Generally, the P-load shows better performance. Compared with the *base* model, the *Pload-16* shows speedups of 28%, 5%, 2%, 14%, 6%, 17%, 50%, 13% and 14% for the respective workloads. In comparison with the *Content-best*, the *Pload-16* performs better by 11%, 4%, 2%, 2%, -8%, 11%, 25%, 16%, and 12%. The P-load is most effective when the workload traverses linked data structures with tight load-load dependences such as *Mcf, Gcc-200, Gcc-scilab, Health, Mst,* and *Em3d*. The content-aware scheme, on the other hand, can prefetch more load-load dependent blocks beyond the instruction window. For example, the traversal lists in *Parser* are very short, and thus provide limited room for issuing P-loads. For this workload, the *Content-best* shows better improvement. Lastly, the results show that a 16-entry P-load buffer provides about 1-7% performance improvement with an average about 3%.

To further understand the P-load effect, we compare the normalized average memory access times of various schemes in Fig. 6 (b). The improvement of the average memory access time matches the IPC improvement very well. In general, the P-load reduces the memory access delays significantly. We observe 10-30% reductions of memory access delay for *Mcf, Gcc-200, Gcc-scilab, Health, Mst,* and *Em3d.*

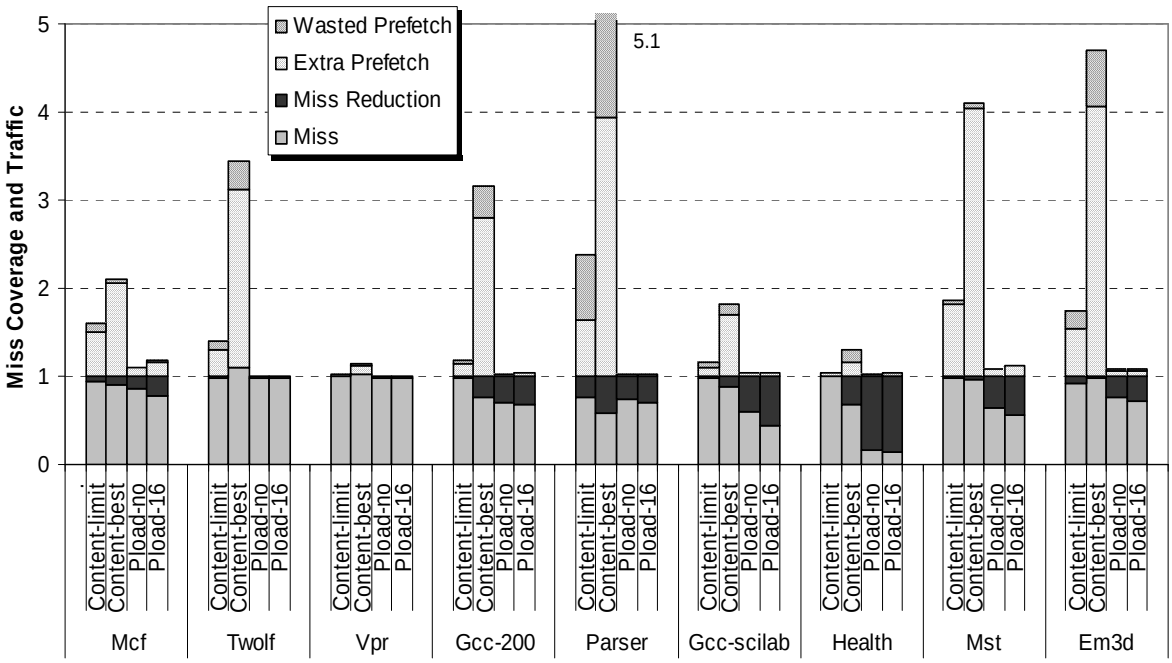


Fig. 7. Miss coverage and extra traffic

5.2 Miss Coverage and Extra Traffic

In Fig. 7, the miss coverage and total traffic are plotted. The total traffic is classified into four categories: misses, miss reductions (i.e. successful P-load or prefetches), extra prefetches, and wasted prefetches. The sum of the misses and miss reductions is equal to the baseline misses without prefetching, which is normalized to 1. The extra prefetch represents the prefetched blocks that are replaced before any reference. The wasted prefetches are referring to the prefetched blocks that are presented in cache already.

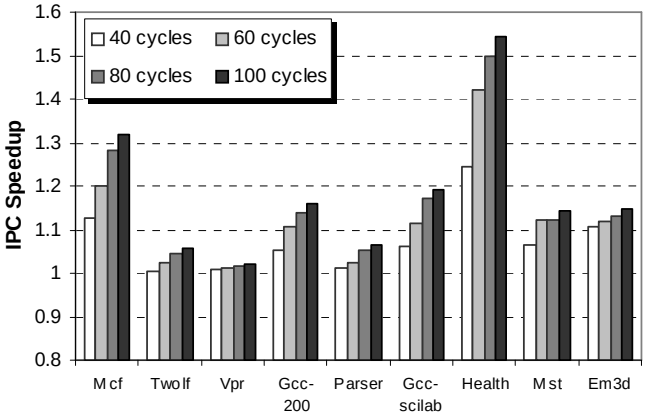
Except for *Twolf* and *Vpr*, the P-load reduces 20-80% overall misses. These miss reductions are accomplished with little extra data traffic because the P-load is issued according to the instruction stream. Among the workloads, *Health* has the highest miss reduction. It simulates health-care systems using a 4-way B-tree structure. Each node in the B-tree consists of a link-list with patient records. At the memory controller, each pointer-advance P-load usually wakes up a large number of dependent P-loads ready to access DRAM. At the processor side, the return of a parent load normally triggers dependent loads after their respective blocks are available from early P-loads. *Mcf*, on the other hand, has much simpler operations on each node visit. The return of a parent load may initiate the dependent loads before the blocks are ready from early P-loads. Therefore, about 20% of the misses have reduced penalties due to the early P-loads. *Twolf* and *Vpr* show insignificant miss reductions because of very small amount of tight load-load dependences.

The content-aware prefetcher generates a large amount of extra traffic for aggressive data prefetching. For *Twolf* and *Vpr*, such aggressive and incorrect prefetching actually increase the overall misses due to cache pollution. For *Parser*, the *Content-best* out-performs the *Pload-16* that is accomplished with 5 times memory traffic. In many workloads, the *Contest-best* generates high percentages of wasted prefetches. For example for *Parser*, the cache prediction at the memory controller is very accurate with only 0.6% false-negative prediction (predicted hit, actual miss) and 3.2% false-positive prediction (predicted miss, actual hit). However, the total predicted misses are 10%, which makes 30% of the return P-load blocks wasted.

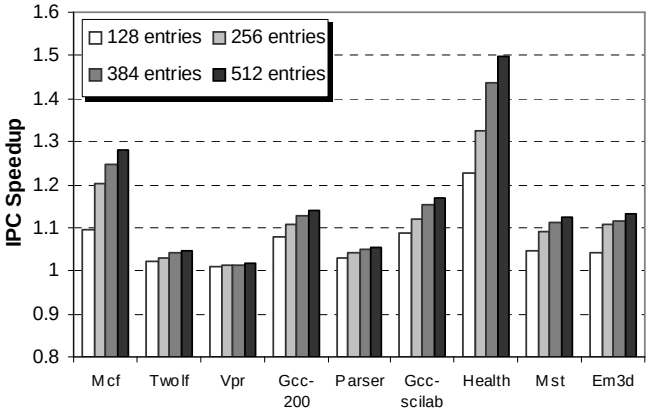
5.3 Sensitivity Study

To reduce memory latency, a recent trend is to integrate the memory controller into the processor die with reduced interconnect delay [2]. However, in a multiple processor-die system, significant interconnect delay is still encountered in accessing another memory controller located off-die. In Fig. 8 (a), the IPC speedups of the P-load with different interconnect delays are plotted. The delay indeed impacts the overall IPC significantly. Nevertheless, the P-load still demonstrates performance improvement even with fast interconnect. The average IPC improvements of the nine workloads are 18%, 16%, 13%, and 8% with 100-, 80-, 60-, and 40-cycle delays respectively.

The scope of the MLP exploitation with P-load is confined within the instruction window. In Fig. 8 (b), the IPC speedups of the P-load with four window sizes: 128, 256, 384, and 512 are plotted. The advantage of larger windows is very obvious since the bigger the instruction window, the better the MLP can be exploited using the P-load. It is important to point out that



(a)



(b)

Fig. 8. Sensitivity of P-load with respect to: (a) interconnect delay; (b) instruction window size

issuing P-loads is independent of the issue window size. In our simulation, the issue window size remains at 32 for all four instruction windows.

6 Related Work

There have been many software and hardware oriented prefetching proposals for alleviating performance penalties on cache misses [13, 6, 16, 12, 27, 25, 4, 22, 9, 8, 26, 28, 11]. Traditional hardware-oriented sequential or stride-based prefetchers work well for applications with regular memory access patterns [6, 13]. However, in many modern applications and runtime environments, dynamic memory allocations and linked data structure accesses are very common. It is difficult to accurately prefetch due to their irregular address patterns. Correlated and Markov prefetchers [5, 12] record patterns of miss addresses and use the past miss correlations to predict future cache misses. These approaches require a huge history table to record the past miss correlations. Besides, these prefetchers also face challenges in providing accurate and timely prefetches.

A memory-side correlation-based prefetcher moves the prefetcher to the memory controller [22]. To handle timely prefetches, a chain of prefetches based on a pair-wise correlation history can be pushed from memory. Accuracy and memory traffic, however, remain difficult issues. To overlap load-load dependent misses, a cooperative hardware-software approach called push-pull uses a hardware prefetch engine to execute software-inserted pointer-based instructions ahead of the actual computation to supply the needed data [27, 28]. A similar approach has been presented in [11].

A stateless, content-aware data prefetcher identifies potential pointers by examining word-based content of a missed data block and eliminates the need to maintain a huge miss history [9]. After the prefetching of the target memory block by a hardware-identified pointer, a match of the block address with the content of the block can recognize any other pointers in the block. The newly identified pointer can trigger a chain of prefetches. However, to overlap long-latency in sending the request and receiving the pointer data for a chain of dependent load-loads, the stateless prefetcher needs to be implemented at the memory side. Both virtual and physical addresses are required in order to identify pointers in a block. Furthermore, by prefetching all identified pointers continuously, the accuracy issue still exists. Using the same mechanism to identify pointer loads, the pointer-cache approach [8] builds a correlation history between heap pointers and the addresses of the heap objects they point to. A prefetch is issued when a pointer load misses the data cache, but hits the pointer cache. Additional complications occur when the pointer values are updated.

The proposed P-load abandons the traditional approach of predicting prefetches with huge miss histories. It also gives up the idea of using hardware and/or software to discover special pointer instructions. With deep instruction windows in future out-of-order processors, the proposed approach easily identifies existing load-load dependences in the instruction stream that may delay the dependent loads. By issuing a P-load in place of the dependent load, any pointer-chasing, or indirect addressing that causes serialized memory access, can be overlapped to effectively exploit memory-level parallelism. The execution-driven P-load can precisely preload the needed block since it does not involve any prediction.

7 Conclusion

Processor performance is significantly hampered by limited MLP exploitation due to the serialization of loads that are dependent on one another and miss the cache. The proposed special P-load has demonstrated its ability to effectively overlap these loads. Instead of relying on miss predictions of the requested blocks, the execution-driven P-load precisely instructs the memory controller in fetching the needed data block non-speculatively. The simulation results demonstrate high accuracy and significant speedups using the P-load.

8 Acknowledgement

This work is supported in part by an NSF grant EIA-0073473

and by research and equipment donations from Intel Corp. Anonymous referees provide helpful comments.

References

[1] H. Akkary, R. Pajwar, and S. T. Srinivasan, “Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors,” *Proc. of the 36th Int’l Conf. on Microarchitecture*, Dec. 2003, pp. 423-434.

[2] AMD Opteron Processors, http://www.amd.com.

[3] D. Burger and T. Austin, “The SimpleScalar Tool Set, Version 2.0,” *Technical Report #1342*, CS Department, University of Wisconsin-Madison, June 1997.

[4] B. Cahoon, K.S. McKinley, "Data Flow Analysis for Software Prefetching Linked Data Structures in Java", *Proc. of the 10th Int'l Conf. On Parallel Architectures and Compilation Techniques*, 2001, pp. 280-291.

[5] M. Charney and A. Reeves. "Generalized Correlation Based Hardware Prefetching," *Technical Report EE-CEG-95-1*, Cornell University, February 1995.

[6] T. Chen, J. Baer, "Reducing Memory Latency Via Non-Blocking and Prefetching Caches," *Proc. of 5th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 1992, pp. 51-61.

[7] Y. Chou, B. Fahs, and S. Abraham, “Microarchitecture optimizations for exploiting memory-level parallelism,” *Proc. of the 31st Int’l Symp. on Computer Architecture*, June 2004, pp. 76-87.

[8] J. Collins, S. Sair, B. Calder and D. M. Tullsen, “Pointer Cache Assisted Prefetching,” in *Proc. of the 35th Int'l Symp. on Microarchitecture*, Nov. 2002, pp. 62-73.

[9] R. Cooksey, S. Jourdan, D. Grunwald, “A Stateless, Content-Directed Data Prefetching Mechanism,” *Proc. of the 10th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002, pp. 279-290.

[10] J. Fu, J.H. Patel, and B.L. Janssens, "Stride directed prefetching in scalar processors," *Proc. of the 25th Annual Int’l Symp. on Microarchitecture*, Dec. 1992, pp. 102-110.

[11] H. J. Hughes and S. V. Adve, “Memory-Side Prefetching for Linked Data Structures for Processor-in-Memory Systems,” *Journal of Parallel and Distributed Computing*, 65 (4), April 2005, pp. 448 – 463.

[12] D. Joseph, and D. Grunwald, "Prefetching Using Markov Predictors," *Proc. of 26th Int'l Symp. on Computer Architecture*, Jun 1997, pp. 252-263.

[13] N. P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," *Proc. of 17th Int'l Symp. on Computer Architecture*, May 1990, pp. 364-373.

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

[15] W.-F. Lin, S. K. Reinhardt, and D. Burger, “Reducing DRAM Latencies with an Integrated Memory Hierarchy Design,” In *Proc. of the 7th Int’l Symp. on High Performance Computer Architecture*, Jan. 2001. pp. 301–312.

[16] C. Luk, T. C. Mowry, "Compiler-Based Prefetching for Recursive Data Structures", *Proc. of the 7th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, Oct. 1996, pp. 222-233.

[17] T. C. Mowry, M. S. Lam, A. Gupta, "Design and Evaluation of a Compiler Algorithm for Prefetching", *Proc. of the 5th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 1992, pp. 62-73.

[18] Olden Benchmark, http://www.cs.princeton.edu/~mcc/olden.html.

[19] A. Roth, A. Moshovos, and G. Sohi, “Dependence based prefetching for linked data structure,” *Proc. of the 8th Int’l conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 1998, pp. 115-126.

[20] S. Sair and M. Charney, "Memory behavior of the SPEC2000 benchmark suite," *Technical Report, IBM Corp*., Oct. 2000.

[21] SimpleScalar website, http://www.eecs.umich.edu/~chriswea/benchmarks/spec2000.html.

[22] Y. Solihin, J. Lee, and J. Torrellas, “Using a User-Level Memory Thread for Correlation Prefetching,” *Proc. of the 29th Annual Int’l Symp. on Computer Architecture*, May 2002, pp.171-182.

[23] SPEC 2000 benchmark. http://www.spec.org/osg/cpu2000/.

[24] L. Spracklen and S. Abraham, “Chip Multithreading: Opportunities and Challenges,” *Proc. of the 11th Int'l Symp. on High Performance Computer Architecture*, Feb. 2005, pp. 248-252.

[25] S. Vanderwiel, and D. Lilja, "Data Prefetch Mechanisms," ACM Computing Surveys, June 2000, pp. 174-199.

[26] Z. Wang, D. Burger, K. S. McKinley, S. K. Reinhardt and C. C. Weems, “Guided Region Prefetching: a Cooperative Hardware/Software Approach,” *Proc. of the 30th Int’l Symp. on Computer Architecture*, June 2003, pp. 388-398.

[27] C.-L. Yang and A. R. Lebeck, “Push vs. Pull: Data Movement for Linked Data Structures,” *Proc. of the 14th Int'l Conf. on Supercomputing*, May 2000, pp. 176-186.

[28] C.-L. Yang and A. R. Lebeck, “Tolerating Memory Latency through Push Prefetching for Pointer-intensive Applications”, *ACM Transactions on Architecture and Code Optimization, vol. 1, No. 4*, Dec. 2004, pp 445-475.