Hardware Prefetching Schemes

Authors: Pirouz Pourdowlat, Peyman Tajbakhsh,
Ahmed Yehia, Behshad Khatambakhsh
Professor: Dr. N. Mekheil
Course: EE8207
Presentation Date: April 18, 2005

Abstract

In recent years, Microprocessors execution speed has improved rapidly, but memory access time has improved in much slower rate. Consequently, Microprocessor performance has been affected by the much slower memory access. Microprocessors waste multiple clock cycles waiting for data availability form the memory. Cache memories were added to Microprocessor architecture to overcome this deficiency. Cache improvement is limited to the hit and miss rate. Prefetching mechanism improves cache usage efficiency. Using an effective prefetching technique may improve cache hit rate significantly. We studied six different Hardware/Software schemes [1][2][3][4][5][6] here in this report we present and compare three of them [1][2][3] and improve the one that is the most effective with applications involving Linked Data Structures (LDS) [3].

1 Introduction

Cache implementation in microprocessors architecture promised to improve the memory access latency. However, the cache performance is compromised by on demand memory fetch policy. The data is fetched into the cache only after the microprocessor requests it. This method results in a cache miss for the first time access to a cache block. The hit and miss rate are parameters that represent cache effectiveness.

The idea behind Data Prefetching is that instead of waiting for a cache miss to perform memory fetch, design an algorithm (or a method) to predict future data request by microprocessor. By doing that, the data could be prefetched before microprocessor request. Hence reduce miss rate (or increase hit rate). Ideally, the Prefetch proceeds in parallel with processor computation and completes in time for the processor to access the needed data in the cache.

Generally, prefetching can be implemented in hardware or software. The software prefetching is normally implemented as an instruction in processors instruction (like fetch instruction). The purpose of this project is to discuss the hardware prefetching. Moreover, we present three different hardware prefetching techniques. They are Sequential Prefetching, Dependence Based Prefetching for Linked Data Structures, and Hardware-based Pointer Data Prefetcher.

We briefly discussed the Sequential Prefetching and Dependence Based Prefetching for Linked Data Structures. Their advantages and disadvantages are shown. Then we explore Hardware-based Pointer Data Prefetcher. Lastly, we introduce the addition of small cache called Victim cache to a part of the Hardware-based Pointer Data Prefetcher (Address cache) to improve the performance.

2 Sequential Prefetching

This method utilizes the principle of spatial locality to prefetch data that is likely to be referenced in the near future. Cache fetch is normally done in blocks to take advantage of spatial locality. The processor fetches number of memory locations called cache block each time accesses the cache. The simplest Sequential Prefetching schemes are based on the one block lookahead approach (OBL). This approach prefetches block n+1, when block n is accessed. Sequential Prefetching is designed to benefit form the spatial locality without introducing some of the problems associated with large cache blocks. There are several different OBL approaches depending on what type of access to block n triggers the Prefetch of n+1. We discuss three approaches: Prefetch-on-miss, Tagged Prefetch, and Sequential Prefetch with K=2, where K is the degree of prefetching.

The Prefetch-on-miss scheme simply prefetches block n+1 whenever block n is accessed as a result of a cache miss. If n+1 is already cached, no memory access is initiated. This algorithm is effective whenever a cache miss occurs. Since this scheme requires a cache miss prior to Prefetch, it is only effective less than %50.

The Tagged Prefetch algorithm assigns a tag bit to every memory block. The tag bit is utilized to detect when a block is demand-fetched or prefetched block is referenced for the first time. In both cases, the next sequential block is fetched. Since this algorithm does not require a cache miss to trigger Prefetch, it is more effective. It was found that in some cases it reduces cache miss between %50 to %90 for a set of trace-driven simulation [1].

One shortcoming of the OBL approach is that the Prefetch may not be initiated far enough in advance to avoid a processor memory stall. To solve this problem, it is possible to increase the number of demand fetch from 1 to K, K is degree of prefetching. Upon memory access to block n, blocks n+1 to n+K are prefetched. K is chosen empirically and is highly depending on the degree of spatial locality. For small K result may be insufficient performance improvement, and for large K cache pollution may occur. Ideally, an adaptive Sequential Prefetching must allow K to vary during program execution. The value of K at any given number matched the degree of spatial locality exhibited by a specific program at a specific point in time.

Figure 1 illustrates the behavior of each approach when accessing three contiguous blocks.

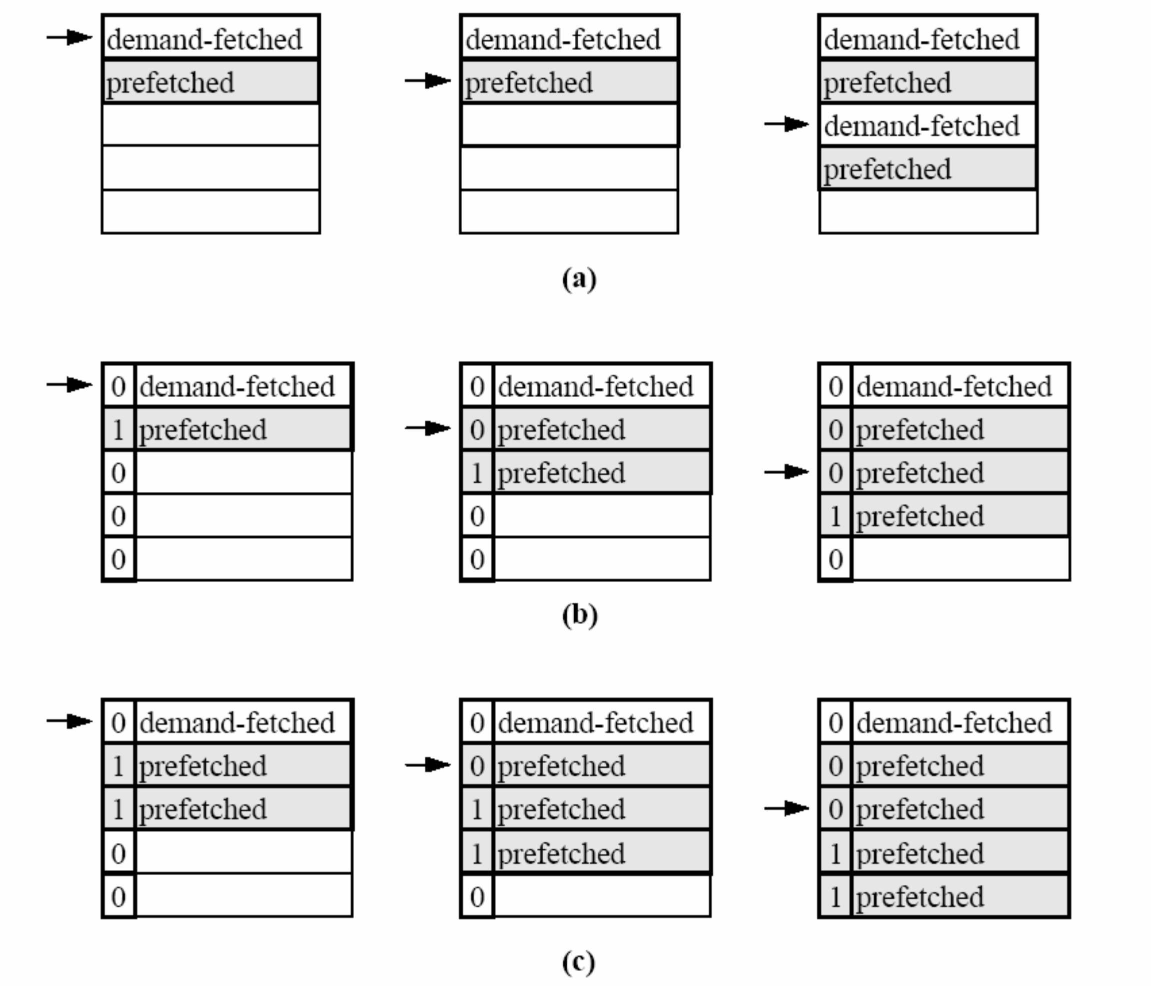


Figure 1 Three forms of Sequential Prefetching: a)Prefetch-on-miss, b)Tagged Prefetch, and c)Sequential Prefetching with K=2.

It is clear that Prefetch-on-miss will result in a cache miss for every other cache block. Hence, it can only improve performance by %50 at most. But, the same access pattern results in only one cache miss when Tagged Prefetch algorithm or Sequential Prefetch with K=2 is used.

3 Dependence Based Prefetching for Linked Data Structures (LDS)

A growing number of important nonnumeric applications employ **L**inked **D**ata **S**tructures (LDSs). For example, databases commonly use index trees and hash tables to provide quick access to large amounts of data. For some other applications, such as C++, JAVA or other object-oriented programs, their data structures may be allocated and de-allocated dynamically during the execution time. Current trends show these types of codes will become increasingly important relative to scientific workloads on future high-performance platforms.

These types of programs usually define an object as a node and use pointer variables to link different objects or nodes together to build the complex data structure. LDS could have single link and arranged in a linear order which is normally called a single linear linked-list. Other LDS are double linked-list, binary tree and multi-way tree.

The Dependence Based Prefetching [2] is one hardware prefetching scheme which addresses the issue of effectively prefetching data in applications with LDS.

The main idea of this method is based on finding “pointer loads” and establish the dependence relationship between the loads producing addresses and the loads consuming these addresses. A pointer load is a load instruction whose address register is the target register of a previous load instruction. The first load instruction whose target register is the source register for the second load is called the “producer” and the second load which uses the target register of the first load as its source register is called the “consumer” load. As a producing load is executed based on its target value, a prefetching is issued. This way hopefully the consuming load will find the data ready in a higher level in the memory hierarchy [2].

3.1 Main Components

Dependence Based Prefetching has four main components:

1. Correlation Table (CT): this is the component responsible for storing dependence information about the loads. It could be implemented as a cache with some degree of associativity
2. Potential Producer Window (PPW): this is the component which maintains the list of the most recently loaded values and the corresponding load instruction. It could be implemented as a Queue, or a cache.
3. Prefetch Request Queue (PRQ): this component buffers request for prefetch until data is available to service them.

4. Prefetch Buffer (PB): this component is a small data cache that temporarily holds prefetched blocks

3.2 Prefetching Process:

Prefetch requests are issued to the PRQ when an address load completes in the processor. A completed load probes the CT in search of potential consumers. Upon a match, a prefetch address is formed by applying the address generation formula to the value just loaded and the request is enqueued on to the PRQ. Prefetch requests are dequeued by PRQ and serviced by the memory system when a data cache port is free. The PB attempts to extract the block from the first level cache, issuing a request to the second level cache on a miss.

The dependency information is stored in CT in the following format:

prod.add , con.add , tmpl

Where;

prod.add is the producing load instruction’s PC address

con.add is the consuming load instruction’s PC address

tmpl is a condensed form of consuming load itself

To establish producer-consumer relationship and thus create CT entries, we need to maintain a list of recently loaded values and their corresponding load instructions. These are stored in PPW in the following format:

addrval , prod.add

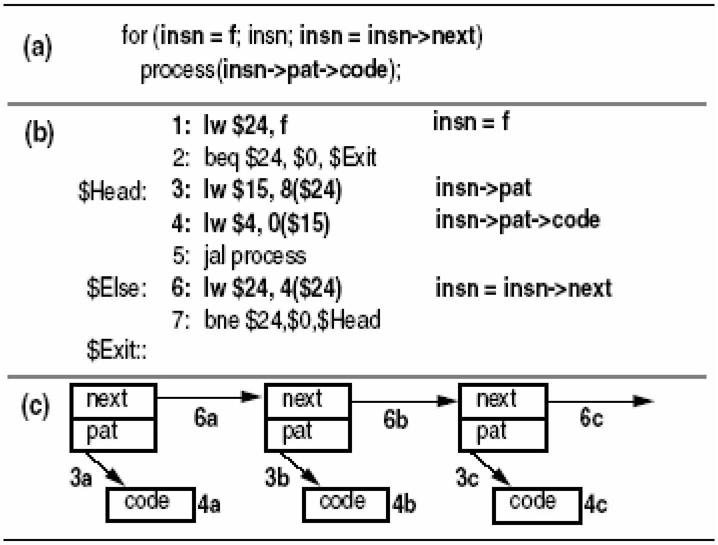
Where;

addrval is the target value created by the producer (i.e. the base address used by the consumer)

3.3 The Process of creating correlations in CT:

Correlations are created at instruction commit time. As a load commits, its base address value is checked against entries in PPW to find a potential producer for that load. Upon a match, a correlation entry is created in CT. At last, the value loaded by the current load is entered together with the current PC as a new addrval-prod.add pair in PPW.

Figure 2 shows a piece of C code and its assembly equivalent which is responsible for traversing an LDS



```
(a)       for (insn = f; insn; insn = insn->next)
            process(insn->pat->code);
```

```
(b)                1:  lw $24, f             insn = f
                   2:  beq $24, $0, $Exit
        $Head:     3:  lw $15, 8($24)        insn->pat
                   4:  lw $4, 0($15)         insn->pat->code
                   5:  jal process
        $Else:     6:  lw $24, 4($24)        insn = insn->next
                   7:  bne $24,$0,$Head
        $Exit::
```

Figure2. LDS traversal example. (a) Source and (b) machine Code that traverses a linked list (c) List layout in memory

Figure 3 shows how different PPW and CT are filled up as the assembly code given in figure 2 is executed assuming that the memory has the content given on the right column of the figure 3.

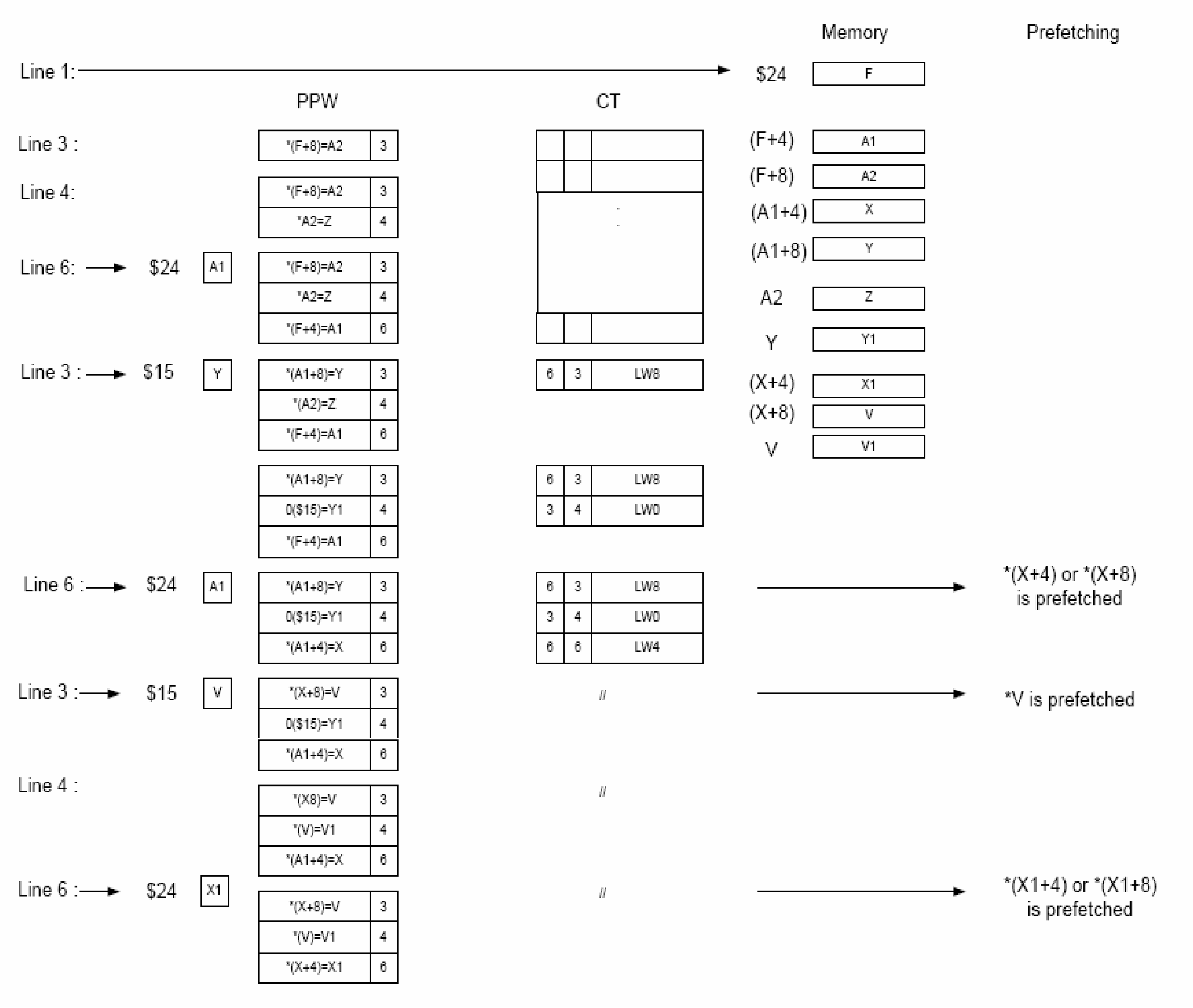


Figure3. How PPW and CT get filled up and prefetching is initiated as the assembly code in figure 2 is executed.

3.4 Performance Evaluation:

According to [2] The average speedup for a system based on Dependence Based Prefetching with a 1KB PB is 10% -significantly outperforming a basic system with an extra 32KB of data cache. However it should be noted that CT is implemented as a multi-way set associative cache and not all pointer loads are actually address loads. As a result of this latter issue, unnecessary prefetches maybe initiated in this scheme. The hardware prefetching scheme which is discussed next, uses a direct map cache as its storage unit for dependence information and also is capable of distinguishing address loads from regular data loads.

4 Hardware Based Pointer Data Prefetcher

Sequential prefetching is suitable for applications which show some type of address stream regularity. For example applications where there is a constant stride between any two sequentially accessed memory locations [1]. This happens usually when there is an array data structure involved and its members are being “indexed” according to some arithmetic formula. However as it was explain in great detail at the beginning of the section 3, in Linked Data Structures (LDS), new “elements” are created and added to the data structure “dynamically”, therefore, there is no address stream regularity between sequentially accessed memory locations, to be captured via an arithmetic formula. Among the there prefetching schemes we studied in detail, only Dependence Based Prefetching and Hardware Based Pointer Data Prefetcher seemed to be suitable for applications involving LDS.

However, as it was briefly entioned in the section 3.4, Dependence Based Prefetching has the following shortcomings when compared to Hardware Based Pointer Data Prefetcher:

A) While Hardware Based Pointer Data Prefetcher stores correlation information in a direct mapped cache, Dependence Based Prefetching will require multi-way set associative caches to store the “correlation” information between members of the LDSs where there is multi-way trees are involved.

B) The Dependence Based Prefetching may confuse “Data loads” with “Address loads” (the classification of different types of loads will be explained later), and this will cause unnecessary prefetches. However, as it will be explained later, Hardware Based Pointer Data Prefetcher is capable of successfully distinguishing “address loads” from the “data loads” and therefore, will not initiate any unnecessary prefetches on the account of “Data loads”.

Also it is the case that most hardware prefetching schemes require two large tables to identify potential pointer loads and predict their addresses. One table records on-the-fly load operations and another table keeps track of the

link between producer and consumer loads. In Hardware Based Pointer Data Prefetcher, on the other hand there is only one major table (AC) to store the information necessary for establishing the correlation between different load instructions.

Considering these two advantages of Hardware Based Pointer Data Prefetcher over the Dependence Based Prefetching scheme made us choose the former scheme for studying and improving.

4.1 General Idea

The essence of this approach is based on the fact that although a pointer elements location is unclear until its previous element is known, most of the time these elements locations are fixed. These locations are saved in a unit called

Address Cache (AC). As the location of the next memory block to be accessed is “speculated”, this memory location is prefetched into a separate unit called “Prefetch Buffer” (PFB). Once the target address of any load instruction has been calculated both the PFB and L1D cache are probed simultaneously, and if the memory location pointed by the target address has been previously prefetched it will be accessed through the PFB.

4.2 Classification of Different Types of Loads

As it can be seen, the Hardware Based Pointer Data Prefetcher scheme (similar to Dependence Based Prefetching) mainly relies on the ability to find the “load” instructions which are responsible for the element=element->next operation in the C language (or p=p.next in Java). Therefore different load instructions in the assembly code must be categorized into different groups, in a way that the specific load instructions responsible for loading of the address of the next LDS element to be accessed could be recognized and a “prefetching event” should only be initiated for these loads.

Therefore a “pointer load” is defined as a “load operation” whose producer is another load operation. However, it should be noted that not all “pointer loads” are responsible for the “element=element->next” operation. As a result the pointer loads are categorized into the following three groups:

A) Address Loads: These are pointer loads whose consumers are also load operations.

B) Data Loads: These are pointer loads whose consumers could be any operation except loads, branches and stores.

C) Data-Address Loads: These are pointer loads whose consumers include both load operations and non-load operations (except branch and stores).

The following piece of code is a good example to explain how different types of pointer loads are categorized:

```
1.     Loop: lw r3, 0(r4)   # Data load
2.     be r3, r0, Exit
3.     sub r10, r3, r9
4.     bne r10, r0, Right
5.     lw r4, 4(r4)         #Address load
6.     jmp Loop
7.     Loop: lw r3, 0(r4)   #Data load
8.     be r3, r0, Exit
9.     sub r10, r3, r9
10.    bne r10, r0, Right
11.    Right:lw r4, 8(r4)   #Address load
12.    jmp Loop
13.    Loop:lw r3, 0(r4)    #Data load
14.    be r3, r0, Exit
```

Assuming that r4 has got its value by another load operation before the instruction 1, instructions 1, 5, 7, 11 and 13 are all pointer loads. This is because, for example, in case of the instruction 5 -based on our assumption- r4 (the source register of this instruction), is the target register of some other load instruction (executed before instruction 1). In case of instruction 7, its source register (r4) is the target register of the load on line 5.

As for the classification of the pointer loads into the aforementioned three subgroups: The load on the line 1 is a data load because its consumer is the “sub” instruction on line 3. The load on line 5 is an address load because its consumer is the load on line 7 and so on and so forth. Were there -for example- another instruction: lw r8,0(r3) right between the line 8 and the line 9, the load instruction on line 7 would become a Data-Address load, because its first consumer would be the load instruction that we just hypothetically inserted in the code and its second consumer would be the sub instruction on line 9. On the other hand if the same load were to be inserted between instructions on line 9 and 10 then the load instruction on line 7 would still remain as a Data-load, because the value of r3 would be arithmetically manipulated before being used as a base address by the load instruction we inserted between lines 9 and 10.

The piece of code we presented above has been extracted in [3] from the tsp bench mark. Lets assume that the memory has the content shown in figure 4:

It could be seen from this code why we try to distinguish address loads (and data-address loads) from the data loads. The load instructions responsible for moving from one LDS element to another are the ones which load the “address” to the next memory element. That leaves us with address loads and data-address loads. Data loads (eg. A load which would load the locations 200, 300 or 400 is not p=p->next) are not moving us between elements in LDS.

100	Data1
104	200
108	300
…	
200	Data2
204	NIL
208	400
…	
300	Data3
304	NIL
308	NIL
…	
400	Data4
404	NIL
408	NIL

Figure4. Memory content corresponding to a small binary tree

To further see the significance of distinguishing between the loads see the figure 5 which has been taken from [3] and shows the percentage of the number of each type of load to the sum total number of all pointer loads in the bench mark.

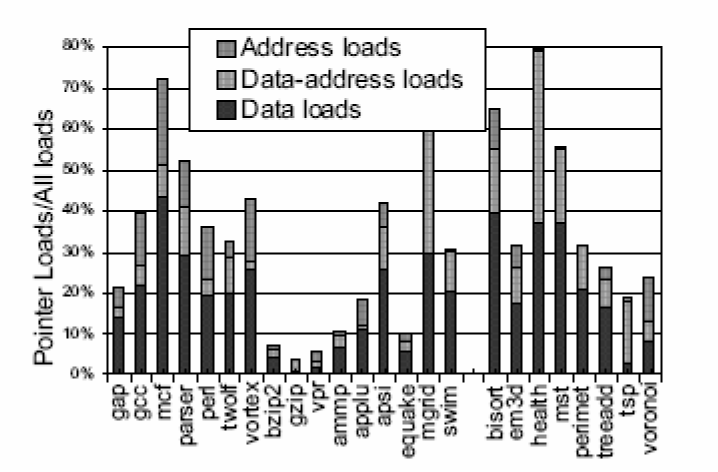


Figure 5: Pointer loads distributions for different benchmarks

4.3 Main Components

Hardware Based Pointer Data Prefetcher scheme has the following main components;

1) Address Cache (AC): This is where the history of target addresses of the pointer loads is recorded. It could be implemented as a simple direct mapped cache.

As a load instruction's effective address is calculated, the AC is probed using this address. If in the AC, the entry in AC corresponding to this effective address is used to prefetch the next element. After the pointer loads the target value is ready, the effective address is used to index into AC and update the target value. This means that AC correlates target addresses to effective addresses for "address loads" and "data-address loads".

2) Target Register Bitmap (TRB): This is a register used to recognize pointer load instructions. Its bit number is equal to the number of the registers in the microprocessor architecture for which prefetching mechanism is being designed. Once a load instruction is decoded, its source register would be used to index its corresponding bit in TRB (r0 would index bit 0 in TRB, r1 would index bit1 and so on and so forth). If the corresponding bit in TRB is 1 then the load is recognized as a pointer load, otherwise the load will not be categorized as a pointer load.Originally all bits in TRB are reset to 0. As each instruction is decoded the bits in TRB are manipulated according to the following;

i. Every load operation set the TRB indexed by its target register id.

ii. ii. Every non-load register write reset the TRB indexed by its target register id.

iii. iii. Since branch and store operations have no effect on register value, these two types of instructions will not update TRB.

iv. iv. Since move operation perform data migration from one register to another register, the source bit value is copied to the bit of destination register id in TRB2.

Figure 6 gives a good example as to how the TRB is manipulated as a sequence of instructions are decoded. Assumptions are that the number of registers in the microprocessor architecture is 8 and that TRB contains the value 01000000 before the first instruction in this sequence is decoded.

Lw r5, 0(r1)	**0**	**1**	**0**	**0**	**0**	**1**	**0**	**0**	**Index TRB[1]** **Set TRB[5]**
Move r2,r5	**0**	**1**	**1**	**0**	**0**	**1**	**0**	**0**	**Index TRB[5]** **Set TRB[2]**
Lw r7,4(r2)	**0**	**1**	**1**	**0**	**0**	**1**	**0**	**1**	**Index TRB[2]** **Set TRB[7]**
addi r7,r7,2	**0**	**1**	**1**	**0**	**0**	**1**	**0**	**0**	**Reset TRB[7]**
Lw r5, 4(r7)	**0**	**1**	**1**	**0**	**0**	**1**	**0**	**0**	**Index TRB[7]** **Set TRB[5]**

Figure 6: An example of how a pice of code would manipulate TRB bits

3) **Bitmap Queue (BQ):** TRB by itself is not enough to recognize the “pointer loads” this is because the instructions after a branch instruction are fetched based on branch “prediction”. Therefore, speculative instructions fetched and decoded after a mispredicted branch, will erroneously set/reset the TRB bits. To address this problem a circular queue called Bitmap Queue (BQ) is used to store the current value of the TRB as a branch instruction is decoded. If a few instructions later, it turns out that the branch was mispredicted, the entry in BQ prior to the branch will be restored into TRB. Of course a correctly predicted branch is retired from BQ by removing the corresponding TRB values from the head of the BQ.

4) Prefetch Buffer (PFB): This is where the prefetched data is stored. It is separate from the data cache to avoid cache pollution. PFB is implemented as a 1 cycle access buffer and it works as follows: whenever AC outputs a base address of consumer pointer load operation, the data is prefetched from the L1 data cache to the PFB whenever the cache port is idle. If the access to the cache results in a miss in L1-D cache, this prefetch request would go down the memory hierarchy as long as the port is idle. However, this L1-D cache miss request will not update the content of L1-D cache but fill the PFB when the data is returned. In general, after a load is decoded, it will go though rename, schedule, register read and address generation stages. If no memory dependency exists, the read request would send to cache memory and PFB at the same time. If it hits the PFB, the request to L1-D cache will be ignored. Since L1-D cache and PFB are separate physical memory structures, a store operation not only update L1-D cache but also update PFB content as well (if write request hit PFB) to maintain data coherency.

5) pload, aload and dload bits: In addition to the aforementioned four components, there are three additional bits *for* each instruction in the Re Ordering Buffer *(ROB)*. Loads are categorized according to these bits into the groups explained in 4.2. Figure 7 points out the truth table used to identify different types of loads:

PLoad	DLoad	ALoad	Pointer Load Types
0	X	X	Not A Pointer Load
1	0	0	Address Load
1	0	1	
1	1	0	Data Load
1	1	1	Data-Address load

Figure 7:Truth table for distinguishing different pointer loads from eachother

Pload bit is set/reset using TRB as it was mentioned previously in this section. The remaining two bits are set/reset according to the following algorithm: (note that producer.op means the producer operation for the operation currently being decoded.)

```
If [ OP== Branch or OP==store or (producer.op not exist)]
        then exit;
else if [OP==load and producer.op==load] then
        producer.op.ROB.aload=1; end;
else if [OP==non-memory-reference-operation and producer.op==load] then
        producer.op.ROB.dload=1; end;
```

5 Performance Enhancement to Hardware Based Pointer Data Prefetcher

As it was mentioned before, Hardware Based Pointer Data Prefetcher has two main advantages over the Dependence Based Prefetching scheme: the use of a simple direct mapped cache for AC (as opposed to a multi-way set associative cache needed for CT) and the ability to distinguish the data loads from the address loads. However, this scheme has its own shortcomings as well. For example it does not concern itself with "how frequently" a portion of the LDS maybe accessed.

One shortcoming of Hardware Based Prefetching scheme is that if the addresses of two different parts of LDS which are accessed very frequently happen to map into the same location in the AC, then as the pointer loads responsible for loading the address of these locations are executed, they will keep replacing each other in the AC and therefore every time we probe the AC for the address of these memory locations, it will result in a miss and no prefetching will be initiated.

We propose to add a small “victim cache” to mitigate this problem. The small victim cache we propose could be implemented as a fully associative cache with only a few entries. The exact size of the victim cache which would bring a meaningful performance enhancement could only be determined after all features of this prefetching architecture have been implemented and simulated in simple scalar.

6 Features Implemented in Simple Scalar

Up to now the partial code for the following has been added to the sim-outorder.c ;

a) The code for TRB

b) The code for adding pload, aload and dload bits to the entries in the reordering buffer

c) The code/pseudo-code for setting/resetting pload, aload and dload

It must be mentioned that naturally the code for the parts mentioned above require some debugging. Due to shortage of time we were unable to finish the necessary debugging however, the main challenge, namely to find out where to write the code and where to call which functions has been over.

TRB is reset at the very beginning of the main function. In the function ruu_dispatch(), as instructions are brought from the fetch_data queue and put into RUU and LSQ queues, their pload bits are set according to the value of the

TRB (as explained in section 4.3). Also the aload and dload bits for each instruction is appropriately set in the same place, according to the algorithm given in the section 4.3 for setting these bits.

All our additions have been made between commented lines of: /*MY CHANGE*/ and /*END MY CHANGE*/

7 Summary and Future Work

From the three hardware based prefetching methods discussed in this report, the first one is effective in applications where there is a address stream regularities which could be captured by a arithmetic formula [1] (eg. A constant stride between sequentially accessed memory locations). The last two schemes are effective in applications with LDS, however, between the two, Hardware Based Pointer Data Prefetcher, is both less complicated and more effective, therefore it is a better prefetching scheme.

In this scheme the pointer loads are classified into three types: address load, data-Address load and data load. Most of the pointer loads in the benchmarks used to simulate this scheme are data-address loads and data loads. In order to identify the pointer load during execution, a small bitmap structure is used. Only address loads and address-data loads are kept in a small address cache to profile the history of address patterns. The prefetched data are placed into a prefetch buffer in order not to pollute the L1-D cache.

From the simulation results presented in [3], the following main conclusion could be drawn: a system adopting this method with a smaller cache outperforms the base system with twice the L1 D cache by about 6%. It also outperforms a system adopting the method presented in [2] with an equal amount of L1D cache and an equal amount of prefetching hardware. These results can all be seen in figure 8 (IPC improvement for each bench mark application) which has been obtained from [3].

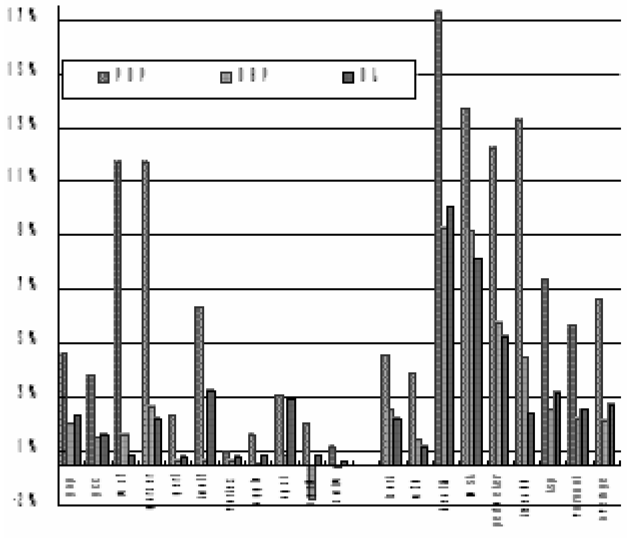


Figure 8: Performance results for Hardware Based Pointer Data Prefetcher

Although Hardware Based Pointer Prefetching provides us with performance improvement compared to Dependence Based Prefetching, it still has its own draw backs. The disadvantage in this scheme which we have tried to mitigate is the fact that two frequently accessed LDS portions whose addresses may map into the same location in AC will kick each other out making it impossible to prefetch either one of them. We propose the addition of a small victim cache to AC to further improve the performance.

Due to shortage of time the code for only parts of the Hardware Based Pointer Prefetching has been implemented in simple scalar. As a result no simulation has been done as yet. Implementation of parts such as BQ, PFB, AC and victim cache, as well as the simulation of the overall system remain as the future work.

References

1. Steven P. Vanderweil and David J.Lilja. Data Prefetching Mechanisms.ACM Computing Surveys, Vol 32, No 2, June 2000
2. Amir Roth, Andreas Moshovos and Gurindar S. Sohi. Dependence Based Prefetching for Linked Data Structures. University of Wisconsin.
3. Shih-Chang Lai and Shih-Lien Lu. Hardware-based Pointer Data Prefetcher. 2003, Proceedings of the 21st International Conference on Computer Design
4. Rodric M. Rabbah, Hariharan Sandanagobalane, Mongkol Ekpanyapong, Weng-Fai Wong. Complier Orchestrated Prefetching via Speculation and Prediction. 2004
5. Ming Zhu, Harsha Narravula, Constantine Katsinis, Diana Hecht. Priority-Driven Active Data Prefetching. Drexel University, Philadelphia.
6. Thomas Lexander and Gershon Kedem, Distributed Prefetch-buffer / Cache Design for High Performance Memory Systems. Duke University, Durham NC, 1996.