Comprehensive Review of Data Prefetching Mechanisms

[1]Sneha Chhabra, [2]Raman Maini

[1]University College of Engineering, Punjabi University, Patiala

[2]Associate Professor, University College of Engineering, Punjabi University, Patiala

er.snhchhabra@hotmail.com, research_raman@yahoo.com

Abstract: *The expanding gap between microprocessor and DRAM performance has necessitated the use of some other techniques designed to reduce or hide the latency of main memory accesses. Although large cache hierarchies have proven to be effective in reducing this latency for the most frequently used data, but it is still not so efficient.*

This paper proposed a technique i.e. ***DataPrefetching Technique*** *for hiding the access latency of data referencing patterns that defeat caching strategies. Rather than waiting for a cache miss to initiate a memory fetch, data prefetching anticipates such misses and issues a fetch to the memory system in advance of the actual memory reference. With data prefetching, memory system call data into cache before processor needs it, thereby reducing memory access latency. The following survey examines several alternative approaches and discusses the design tradeoffs involved when implementing a data Prefetch strategy.*

Keywords: *Prefetching, Memory Latency.*

Introduction

Processor performance has been increasing much faster than that of memory performance over the last three decades. While processor performance followed Moore's law [3] and improved by 50% annually till 2004 and 20% since, memory performance has been improving by a mere 9% a year. This has caused a large gap and made memory a performance bottleneck. The problem is only getting worse with the rapid growth of multicore processors as multiple cores contend for accessing data from memory, which is shared by these cores. Traditionally cache memories were used to improve the performance of memory accesses using the principle of locality. Yet, optimizations are necessary to improve the usage of cache memories and to reduce cache misses. Data prefetching has been considered an effective way to mask data access latency. Data prefetching is a data access latency hiding technique, which decouples and overlaps data transfers and computation. In order to reduce CPU stalling on a cache miss, data prefetching predicts future data accesses, initiates a data fetch, and brings the data closer to the computing processor before it is requested. Memory latency has always been a major issue in sharedmemory multiprocessors. This is even more true as the gap between processor and memory speeds continues to grow. This expanding gap between microprocessor and DRAM performance has necessitated the use of increasingly aggressive techniques designed to reduce or hide the large latency of memory accesses [1].In order to fully utilize such systems, it is essential to use the memory hierarchy [2] effectively, in order to reduce memory latency. In this paper, we study how data prefetching into the first-level cache can eliminate cache misses. In addition, our techniques can generalize to data prefetching in other levels of the memory hierarchy [2]. There are two main classes of data prefetching. In hardware prefetching, the hardware alone decides what data to Prefetch and when and where to Prefetch the data. In software prefetching, the hardware supports a prefetching instruction. The user, or compiler, then directs prefetching by inserting prefetching instructions into the code. Both hardware and software prefetching have been studied extensively, and have been shown to be effective; however, both types of prefetching have their shortcomings. For example, hardware prefetching can require complex and expensive hardware, while software prefetching requires extra CPU instructions. In this paper, we propose a method for integrated hardware/software prefetching. Now, what is necessary to design a data prefetching strategy? Traditionally, prefetching considered two issues: what to Prefetch and when to prefetch. What to Prefetch decides what data a processor might need in the future. When to prefetch decides how early data has to be fetched and avoid any negative effects because of prefetching.

VARIOUS DATA PREFETCHING STRATEGIES

There are various data prefetching techniques:

- Software Prefetching.
- Hardware Prefetching
- Integrated Software and Hardware Prefetching.

SOFTWARE DATA PREFETCHING

It is increasingly common for designers to implement Data prefetching by including a fetch instruction in a microprocessor's instruction set. A fetch specifies the address of a data word to be brought into the cache. Upon execution, the fetch instruction passes this address to the memory system, which forwards the word to the cache. Because the processor does not need the data yet, it can continue computing while the memory system brings the requested data to the cache.

Prefetch scheduling

Choosing where to place a fetch instruction relative to the corresponding **load** or **store** instruction is known as *Prefetch scheduling*. Software prefetching can often take advantage of compile-time information to schedule prefetches more accurately than hardware techniques. In practice, it is not possible to predict exactly when to schedule a Prefetch so that data arrives in the cache exactly when it will be requested by the processor. The execution time between the Prefetch and the matching **load** or **store** may vary, as will memory latencies. These uncertainties must be considered when deciding where in the program to place fetch instructions. If the compiler schedules fetches too late, the data will not be in the cache when the processor needs it. If the compiler schedules the fetches too early, the cache may evict the data before it can be used to make room for data that the controller fetches later. Early prefetches also might displace data in the cache that the processor is using at the time. This situation, in which there is a miss that would not have occurred without prefetching, is called *cache pollution*.

Prefetching for loops

Prefetching is most often used within loops responsible for large array calculation. Such loops, which are common in scientific code, provide excellent prefetching opportunities because they exhibit poor cache utilization and often have predictable memory-referencing patterns.

Simple prefetching. The code segment in shown in figures. This loop sums the elements of array a. If we assume a four-word cache block, this code segment will cause a cache miss every fourth iteration. We can try to avoid these cache misses by using the Prefetch directives added. The Prefetch of the array element to be used in the next loop iteration is scheduled just before computation for the current iteration begins. However, prefetching every iteration of this loop is unnecessary because each fetch actually brings four array elements into the cache. Prefetching should thus be done only every fourth iteration. One solution is to surround the **fetch** directives with an **if** condition that tests when **(i modulo 4) = 0** is true.

Unrolling the loop. Unrolling the loop by a factor of *r* (where *r* is equal to the number of words to be prefetched per cache block) is more effective than using an explicit Prefetch predicate. Unrolling a loop involves replicating the loop body *r* times and increasing the loop increment stride from one to *r*. In the process the compiler doesn't replicate fetch directives, but it does change the *array index*, which it uses to calculate the Prefetch address, from $i + 1$ to $i + r$. Nonetheless, cache misses will occur during the
Loop's first iteration because prefetches are never issued for this iteration. In addition, unnecessary prefetches will occur in the unrolled loop's last

iteration, in which the fetch command tries to access data past array boundaries.

Software pipelining: Software pipelining techniques can solve these problems. In this figure, we have extracted some code segments from the loop body and placed them on either side of the original loop. We have added a *loop prologue* consisting of fetch statements to the beginning of the main loop to prefetch data for the first iteration. We added an epilogue to the end of the main loop to execute the final computations without initiating unnecessary prefetch instructions.

```
for  ( i = 0; I < N; I ++)
ip = ip + a[i]*b[i];
```

(a)

```
for (i = 0; i < N; i++){
fetch( &a[i+1]);
fetch( &b[i+1]);
ip = ip + a[i]*b[i];
}
```

(b)

```
for (i = 0; i < N; i+=4){
fetch( &a[i+4]);
fetch( &b[i+4]);
ip = ip + a[i]*b[i];
ip = ip + a[i+1]*b[i+1];
ip = ip + a[i+2]*b[i+2];
ip = ip + a[i+3]*b[i+3];
}
```

(c)

```
fetch( &ip);
fetch( &a[0]);
fetch( &b[0]);
for (i = 0; i < N-4; i+=4){
fetch( &a[i+4]);
fetch( &b[i+4]);
ip = ip + a[i] *b[i];
ip = ip + a[i+1]*b[i+1];
ip = ip + a[i+2]*b[i+2];
ip = ip + a[i+3]*b[i+3];
}
for ( ; i < N; i++)
ip = ip + a[i]*b[i];
```

(d)

Inner product calculation using a) no prefetching, b) simple prefetching, c) prefetching with loop unrolling and d) software pipelining.

Hardware Prefetching

Hardware-based prefetching techniques do not incur the instruction overhead associated with the use of explicit fetch instructions. These techniques do not require changes to existing executables, so there is no need for programmer or compiler intervention.

Sequential prefetching

Sequential prefetching can take advantage of spatial locality without introducing some of the problems associated with large cache blocks. The simplest sequential prefetching schemes are:-

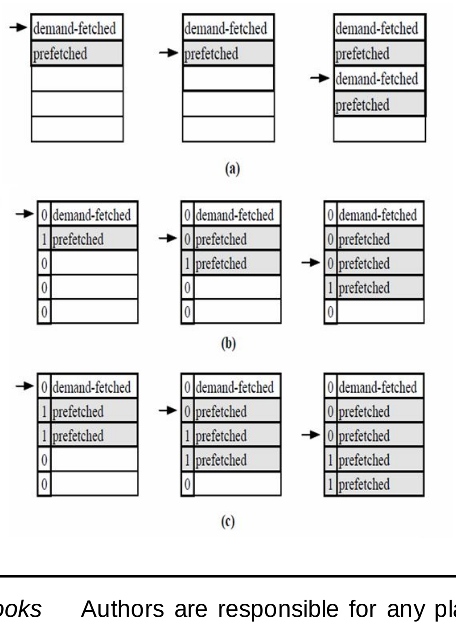
OBL approach. The simplest sequential prefetching schemes are variations upon the *one-block-look ahead* (OBL) approach, which automatically initiates a prefetch for block b + 1 when block b is accessed.

Smith [2] summarizes several of these approaches of which the *prefetch-on-miss* and *tagged prefetch* algorithms will be discussed here. This approach differs from simply doubling the block size because the demand-fetched block and the prefetched block are considered separate items for cache replacement and coherency purposes. The use of separate, smaller blocks means the computer does not have to evict large amounts of data each time it replaces items in the cache. Smaller blocks also reduce the chance of false sharing.OBL implementations differ depending on what type of access to block b initiates the prefetch of b + 1.

The ***prefetch-on-miss*** algorithm initiates a prefetch for block b + 1 whenever an access for block b results in a cache miss. If b + 1 is already cached, no memory access is initiated.

• The ***tagged prefetch*** algorithm associates a tag bit with every cache block. This bit detects when a block is fetched or when a prefetched block is referenced for the first time. In either case, the next block in memory is fetched. Tagged prefetching is more expensive to implement because of the addition of the tag bits to the cache and the need for a more complex cache controller design.

Tagged prefetching, on the other hand, can initiate a "domino effect" that avoids all but the first miss. So, when the prefetched block b + 1 is accessed, a prefetch for b + 2 is initiated, and when b + 2 is accessed, b + 3 is prefetched. The process continues until the sequential access stream terminates. A shortcoming of OBL schemes is that the memory system may not initiate a prefetch for data far enough in advance of the data's actual use to avoid a processor memory stall. A sequential access stream resulting from a tight loop, for example, may not allow sufficient time between the use of blocks b and b + 1 to completely hide the memory latency.



**Three forms of sequential prefetching: a) Prefetch on miss, b) tagged prefetch and
c) sequential prefetching with *K* = 2.**

Adaptive sequential prefetching Dahlgren and Stenström [4] proposed an adaptive sequential prefetching policy that allows the value of *K* to vary during program execution in such a way that *K* is matched to the degree of spatial locality exhibited by the program at a particular point in time. To do this, a *Prefetch efficiency* metric is periodically calculated by the cache as an indication of the current spatial

locality characteristics of the program. Prefetch efficiency is defined to be the ratio of useful prefetches to total prefetches where a useful prefetch occurs whenever a prefetched block results in a cache hit. The value of *K* is initialized to one, incremented whenever the prefetch efficiency exceeds a predetermined upper threshold and decremented whenever the efficiency drops below a lower threshold as shown in Figure 2. Note that if *K* is reduced to zero, prefetching is effectively disabled. At this point, the prefetch hardware begins to monitor how often a cache miss to block *b* occurs while block *b*-1 is cached and restarts prefetching if the respective ratio of these two numbers exceeds the lower threshold of the prefetch efficiency.

Simulations of a shared memory multiprocessor found that adaptive prefetching could achieve appreciable reductions in cache miss ratios over tagged prefetching.

Comparing approaches: Simulations on a shared memory multiprocessor found that adaptive prefetching reduced cache misses more effectively than tagged prefetching but did not significantly reduce runtime. Dahlgren and Stenström [4] compared tagged and RPT prefetching in the context of a distributed shared memory multiprocessor. Adaptive sequential prefetching's lower miss ratio was partially offset by the increased memory traffic and contention created by the additional unnecessary prefetches[2].Tagged prefetching is simpler, offers good performance, and is an attractive option when cost and simplicity are important design considerations.

Prefetching with arbitrary strides

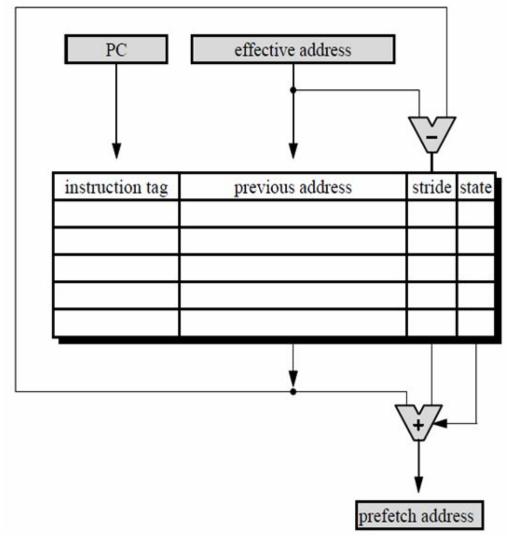
When the processor's referencing pattern strides through nonconsecutive memory blocks, sequential prefetching will cause needless prefetches and will thus become ineffective. We need more elaborate prefetching techniques to take advantage of both small and large stride array-referencing patterns while ignoring references that are not array-based.

One such technique employs special prefetch hardware that monitors the processor's address-referencing pattern and infers prefetching opportunities by comparing successive addresses used by **load** or **store** instructions. If the prefetch hardware detects that a particular **load** or **store** is generating a predictable memory-addressing pattern, it will automatically issue prefetches for that instruction. To illustrate one aggressive scheme, 10 assume that a memory instruction, *m*i, references addresses *a*1, *a*2, and *a*3 during three successive loop iterations. The Prefetch hardware initiates a Prefetch for *m*i if $(a2 - a1) = D\ {}^{1}\ 0$, where D is the stride of a series of array accesses. The first prefetch address will then be $A3 = a2 + D$, where *A*3 is the predicted value of *a*3. Prefetching continues in this way until the equality $An = an$ is no longer true.

At this point, prefetching for instruction *m*i ends.

Reference Prediction Table

To implement this approach, it is necessary to store the previous address used by a memory instruction along with the last detected stride, if there has been one. It is clearly impossible to record the reference histories of every memory instruction. Instead, a separate cache called the *reference prediction table* (RPT) holds this information for most of the recently used memory instructions. Table entries contain a memory instruction's address, the previous address accessed by the instruction, a stride value for entries that have established a stride, and a field that records the entry's current state [6]. The table is indexed by the CPU's program counter. When the CPU executes memory instruction *m*i for the first time, it enters the instruction in the RPT with its state set to initial. This shows that the RPT has not initiated prefetching for this instruction. If *m*i is executed again before its RPT entry has been evicted, the RPT calculates a stride value by subtracting the instruction's most recent address stored in the RPT from the current address.



Integrating Hardware and Software Prefetching

Software prefetching relies exclusively on compile-time analysis to schedule fetch instructions within the user program. Zheng and Torrellas [4] suggest an integrated technique that enables prefetching for irregular data structures. In contrast, the hardware techniques discussed thus far infer prefetching opportunities at run-time without any compiler or processor support. Noting that each of these approaches has its advantages, some researchers have proposed mechanisms that combine elements of both software and hardware prefetching. A special fetch instruction is provided that prefetches the specified block into the cache and then sets the tag bit and the value of the *PD* field of the cache entry holding the prefetchedblock [9]. The first *K* blocks of a sequential reference stream are prefetched using this instruction. When a tagged block, *b*, is demand fetched, the value in its

PD field, *K*b, is added to the block address to calculate a prefetch address. The *PD* field of the newly prefetched block is then set to *K*b and the tag bit is set. This insures that the appropriate value of *K* is propagated through the reference stream. Prefetching for non-sequential reference patterns is handled by ordinary fetch instructions. VanderWiel and Lilja [5] propose a prefetch engine that is external to the processor. The engine is a general processor that executes its own program to prefetch data for the CPU. Through a shared second-level cache, a producer-consumer relationship is established between the two processors in which the engine prefetches new data blocks into the cache only after previously prefetched data have been accessed by the compute processor. The processor also partially directs the actions of the prefetch engine by writing control information to memory-mapped registers within the prefetch engine's support logic.

Conclusion

Data prefetching has been shown to be a very promising technique for tolerating the large memory latencies common in shared-memory multiprocessors. Once a prefetch mechanism has been specified, it is natural to wish to compare it with other schemes. Both hardware and software data prefetching schemes have been proposed and evaluated; however, both types of prefetching have their shortcomings. Unfortunately, a comparative evaluation of the various proposed prefetching techniques is hindered by widely varying architectural assumptions and testing procedures. Software prefetching handles both types of referencing patterns but introduces instruction overhead. We propose an integrated hardware/ software prefetching scheme that incorporates the best aspects of both forms of prefetching. Integrated schemes attempt to reduce instruction overhead while still offering better prefetch coverage than pure hardware techniques. These two facets are handled by the software support in our integrated scheme; therefore, our hardware support is simpler than that of other hardware prefetching schemes. However, in our scheme, most data accesses can still be handled in hardware.

References

1. Gupta, A., Hennessy, J., Gharachorloo, K., Mowry, T. and Weber, W.-D., "Comparative Evaluation of Latency Reducing and Tolerating Techniques," *Proc. 18th International Symposium on Computer Architecture*, Toronto, Ont., Canada, May 1991, p. 254-263.
2. Smith, A.J., "Cache Memories," *Computing Surveys*, Vol.14, No.3, September 1982, p. 473-530.
3. F. Dahlgren, M. Dubois and P. Stenstrom, "Fixed and Adaptive Sequential Prefetching in Shared-memory Multiprocessors," *Proc. International Conference on Parallel Processing*, St. Charles, IL, August 1993, p. I-56-63.
4. F. Dahlgren and P. Stenstrom, "Effectiveness of Hardware-based Stride and Sequential Prefetching in Shared-memory Multiprocessors," *Proc. First IEEE Symposium on High-Performance Computer Architecture*, Raleigh, NC, Jan. 1995, p. 68-77.
5. Oberlin, S., R. Kessler, S. Scott and G. Thorson, *The Cray T3E Architecture Overview*, Cray Research Inc., Eagan, MN, 1996.
6. S. Palacharla and R.E. Kessler, "Evaluating Stream Buffers as a Secondary Cache Replacement," *Proc. 21st International Symposium on Computer Architecture*, April 1994.
7. R.H Patterson and G.A. Gibson, "Exposing I/O concurrency with informed prefetching," *Proc. Third International Conf. on Parallel and Distributed Information Systems*, Austin, TX, September 1994, p. 7-16.

* * * * *