

# A Survey of prefetching techniques

Nir Oren

July 18, 2000

## Abstract

*As the gap between processor and memory speeds increases, memory latencies have become a critical bottleneck for computer performance. To reduce the bottleneck, designers have had to create methods to hide these latencies. One popular method is prefetching. This method fetches the data from the memory system before being asked for by the processor, with the expectation that it will soon be referenced. An effective prefetching scheme reduces cache miss rates and therefore hides the memory latency. The aim of this paper is to provide a survey of hardware prefetching techniques. To achieve this goal, we provide a brief introduction to the concepts behind prefetching. An overview of software prefetching techniques is also given. We are then in a position to examine a number of instruction and data prefetching schemes that have previously been proposed.*

**Keywords:** *prefetching, caches*

**Computing Review Categories:** *B3.2, D3.4*

## 1 Introduction

Microprocessor speeds have increased dramatically over the past decade. This is contrasted by the much slower improvements in access times of main memory. The increasing gap between the two means that unless something is done to hide or reduce memory access latencies, future improvements in CPU speeds will be increasingly nullified by the time taken to retrieve data from memory.

The principle technique used to reduce memory access times is the use of some form of a cache hierarchy. Memory caches are most commonly constructed using fast static RAM, which so far has been able to keep abreast of CPU speeds. SRAM is not used as a computer's main memory system due to price constraints.

While accessing data in the cache closest to the processor does not carry much of a performance penalty, it is very likely that the required data will not be in the cache. This cache miss will cause the processor to stall until the desired data can be brought in from a higher level of the memory hierarchy. Cache misses are a common cause of CPU stalls in modern computers [Hennessy and Patterson, 1996], and are therefore a significant limitation to performance. By reducing their numbers, overall system speeds can be significantly improved.

Prefetching is used in an attempt to place data close to the processor before it is required, eliminating as many cache misses as possible.

### 1.1 Prefetching

Prefetching attempts to combat cache misses by adding a prefetch operation to the normal processor memory access instructions. Instead of fetching data on a cache miss, prefetching anticipates these misses and issues a fetch to

the memory system before the actual memory reference occurs. This prefetch occurs parallel to normal processor operations, allowing the memory system time to transfer the data from main memory to the cache. Ideally, the prefetch will complete just as the processor needs to access the required data, without stalling the processor. Since more prefetches could be issued while waiting for the first one to complete, prefetch operations are normally non-blocking.

The simplest technique, implemented by many processors today is the use of an explicit fetch instruction. This fetch instruction must at least specify what data should be brought into the cache. When this instruction is issued, the address is passed into the memory subsystem without causing the processor to wait for a response. The memory hierarchy then treats the instruction in a manner almost identical to an ordinary load instruction, with the exception that the result is not forwarded to the processor after it has been cached.

Ideally, the processor issues the fetch instruction so that the data arrives as it is required by the processor. In more realistic cases, prefetches are not issued in this perfect manner, and one of a number of scenarios may arise:

- The fetch instruction may be issued too late to avoid processor stalls from occurring. Some benefit may still be obtained as some time may still be saved.
- The fetch instruction may be issued too early, and will therefore remain in the cache for some time before it is used. During this time, the data is at risk from the normal cache replacement policy, and may end up being replaced before it is used. When this occurs, the prefetch is said to be *useless* as no performance benefit is gained from fetching the block early.
- When a block is fetched prematurely, it may evict data

that is currently in use, or about to be used, by the processor. This effect is referred to as *cache pollution*.

Memory subsystems that operate together with processors that implement prefetching should be designed to allow for a higher frequency of memory requests. This occurs due to the elimination of many of the processor stalls.

Software prefetching schemes make use of a fetch instruction embedded into the program to initiate a fetch before the processor requires the data. The performance gained from the reduction in processor stalls outweighs the extra instructions that must be executed by the processor.

Hardware prefetching circumvents the need for an explicit fetch instruction. This is achieved by utilizing specialized hardware that attempts to guess when a prefetch would be useful. Hardware prefetching therefore has no instruction overhead, but it often generates more unnecessary prefetches than software prefetching. This happens as hardware prefetching schemes speculate on future memory accesses without the advantage of compile time information.

The rest of the paper will examine prefetching techniques and issues related to their implementation. Section two provides a quick introduction to some of the theory of caches, and looks at research related to prefetching, while section three briefly examines software prefetching. Section four will then examine hardware prefetching schemes in detail.

## 2 Background and Related research

### 2.1 Caches

A cache operates within a memory hierarchy by providing lower levels of the hierarchy with faster access to portions of higher levels of the hierarchy by storing the data within low latency memory.

Caches work well due to the principle of locality: It is assumed that if a section of memory is accessed at a certain point in time, it is probable that more memory accesses will occur close to that area of memory. By storing the block of memory around this area within the cache, it is hoped that further memory accesses can circumvent the latencies associated with main memory accesses.

Modern CPUs ordinarily contain at least two levels of cache, a level one (L1) cache, with access times comparable to the speed of the CPU, and a larger, slower L2 cache, which sits between the L1 cache and main memory.

Caches can be classified by their type, associativity, size, and block size. While some caches store both data and instructions, many architectures separate these two roles. The type of cache has serious implications on the effectiveness of prefetching techniques. These will be described in a later section.

The associativity of a cache describes which cache block a memory block can occupy. Within a fully associative cache, a memory block can occupy any location in the cache. In a direct mapped cache, a memory block may

occupy only one area of the cache. A set associative cache consists of a number of sets, each containing an equal number of cache blocks. A single memory block may then occupy any of the blocks within one of the sets. Associativity affects amount of work needed to determine if a desired block of memory is stored in the cache, and therefore influences the maximum speed a cache can achieve, together with the cache's complexity.

A cache's size directly depends on the number of blocks it contains, and the block size. Clearly, a larger cache is more useful than a smaller cache, however, small caches with a higher associativity may perform better than large caches with low associativity. More detailed information about the operation and design of caches can be found in Hennessy and Patterson's book [Hennessy and Patterson, 1996].

### 2.2 Cache Misses

The fetch policy described above will always access main memory the first time a memory block is requested as the requested data must be brought into the cache. This is known as a cache miss. A number of different cache miss types can be identified; the one above is referred to as a *compulsory miss*. Other classes of cache misses include:

- *Capacity misses*. In this case, a request for data that was previously in the occurred, but this old data had already been removed to make way for new data.
- *Conflict misses* are similar to capacity misses, and occur in set associative and direct mapped caches when the desired block of data has been discarded from the cache to make way for a block that maps to the same set.

It should be intuitively clear that a large cache with full associativity will minimize the number of capacity and conflict misses that occur.

### 2.3 Alternatives to prefetching

Apart from prefetching, a number of other techniques have been proposed to overcome high memory latencies. Information about these methods can be found in Hennessy and Patterson's book [Hennessy and Patterson, 1996]:

- Cache hierarchies put information into faster, smaller and more expensive memory systems as the information gets closer to the processor.
- Increases in cache block sizes can reduce compulsory misses by taking advantage of locality. However, miss penalties are increased by this technique, and conflict misses are increased. This is in fact, a type of prefetch, and will be discussed in more detail later.
- Using highly associative and pseudo associative caches reduces the number conflict misses, but very often forces the cache to operate at a lower speed.

- Multiprocessors can reduce memory access times by relaxing their consistency requirements.
- Victim caches are small fully associative caches that store blocks that were discarded from a cache due to a miss. These caches are checked on a miss before accessing the next level of the memory hierarchy.
- Software based data placement schemes can increase the locality of data.
- Trace caches [Postiff et al., 1999] attempt to group spatially separate, but dynamically contiguous instructions together, thereby increasing the instruction fetch bandwidth of the processor.
- Alternative memory schemes [Machanick et al., 1998] have been designed to combat increasing memory latencies using a variety of novel techniques.

## 3 Software based prefetching

### 3.1 Overview

Software prefetching is based on the use of some form of explicit fetch instruction. Simple implementations may simply perform a nonblocking load (perhaps into an unused register), while more complex implementations might provide hints to the memory system as to how the prefetched blocks will be used. This information may be used to guide the cache replacement policy, and is often very useful in multiprocessor systems, where data can be prefetched in different sharing states.

Even though fetch instruction implementations may vary, all share some common characteristics: They are nonblocking memory operations, and therefore require a lockup free cache which will allow prefetches to bypass other outstanding memory operations in the cache. Most often, exceptions are suppressed for prefetches to remove possible overheads such as page faults.

Very little hardware needs to be introduced to take advantage of software prefetching. The difficulty in efficiently using this approach lies in the correct placement of the fetch instruction. The term *prefetch scheduling* refers to the task of choosing where to place the fetch instruction relative to the accompanying load or store instruction.

Uncertainties that cannot be predicted at compile time, such as variable memory latencies and external interrupts, make it impossible to precisely predict where in the program to position a prefetch so as to guarantee that a block arrives in the cache when it is required by the processor. It has been shown [Metcalf, 1993] that very few fetch instructions can have a very large effect on program performance, and therefore, it is possible to gain significant speed advantages by inserting a few fetch instructions manually in strategic portions of the program. More commonly however, a compiler is used to schedule prefetching during the optimization phase of compilation. Furthermore, indications are that with compiler assistance, speedups in data intensive scientific code may easily

reach 500% in systems with a memory latencies 120 cycles [Metcalf, 1993].

Most commonly, prefetch instructions are used within loops that reference large amounts of memory (most commonly, these types of loops perform large array calculations). These loops very often utilize caches very poorly, but have predictable memory access patterns. They are therefore prime candidates for prefetching optimizations, memory accesses that will occur in future iterations may be prefetched in the current iteration.

### 3.2 Disadvantages of Software Prefetching

Scientific applications ordinarily contain memory accesses with a predictable linear access pattern, which a compiler can take advantage of to insert prefetching instructions. General applications on the other hand are not as well suited to compiler based prefetching optimization for a number of reasons. Firstly, the compiler often has difficulty predicting the memory access pattern of an ordinary application due to its non linear (and irregular) nature. Secondly, most common applications have complex control and data structures which limit the ability to predict when and which items of data will be accessed. Finally, most general applications exhibit comparatively high temporal locality, yielding better cache utilization, and therefore reducing the benefits of prefetching.

One further drawback of software prefetching is the overhead on performance of utilizing an explicit fetch instruction. Apart from the extra execution cycles taken up by the instruction, the calculation of the fetch's target address may require additional processing. Furthermore, since the calculated address must later be used for load or store operations, it is often useful to store this address so as to prevent unneeded recalculations from occurring. This storage most commonly takes place within a register, meaning that the compiler will have less register space to use in performing other operations. This may mean that the program requires additional code to manage variables that are 'spilled' out into main memory due to insufficient register space. If memory latencies are large, this problem may be magnified, as more registers may be needed to keep track of multiple prefetches. If insufficient registers exist to keep track of all prefetching addresses, these addresses may have to be stored in memory, exacerbating the problem further.

The added fetch instructions, together with loop unrolling (which may aid data prefetching effectiveness), can result in significant code expansion, which may have a detrimental effect on instruction cache performance. One final issue with software based prefetching is that since it occurs at compile time, it has no way of detecting if prefetched data was removed from the cache before it was utilized, and cannot therefore issue another prefetch so as to reinsert it.

## 4 Hardware Prefetching

### 4.1 Overview

Hardware prefetching schemes have a few of advantages over software prefetching methods:

- They require no help from the programmer or compiler.
- Hardware prefetching may provide speedups to existing programs, without the need for recompilation.
- They do not increase the program size by inserting unnecessary instructions into the code. This in turn can often mean better utilization of the instruction cache and processor.
- Hardware prefetching techniques can operate on both data and instruction caches. Software techniques normally only prefetch data.

On the other hand, software prefetching techniques provide the following advantages over hardware prefetching schemes:

- Application-specific prefetching optimizations can be applied.
- The availability of compile time information means that software prefetching is often more accurate.
- Very often, less hardware is required for software prefetching, resulting in cheaper cost, and offering the possibility of lower cycle times.

From a performance perspective, it was found that hardware schemes introduce a large amount of memory traffic over the network, while software schemes bring in a non-negligible instruction execution overhead [Chen and Baer, 1994]. Benchmarking results on commercial systems [Acquaviva, 1999] seem to indicate that the overheads of software prefetching may actually sometimes hinder rather than aid performance, and, while hardware prefetching does not suffer from this problem, its performance gains are often more limited.

The techniques we will investigate in this section can broadly be divided into categories according to what type of caches they are suited to operate upon: Instruction prefetching methods, and data prefetching methods. Some techniques can operate on both types of caches, but, in this survey, will not be categorized separately.

Another possible subdivision subdivides hardware prefetching methods into two other categories, independent of the previously described division. Spatial methods, as defined by them are those methods which use access to the current block as the basis for prefetch decisions, while temporal methods utilize lookahead decoding of the instruction stream to decide what and when to prefetch [Chen and Baer, 1995].

### 4.2 Instruction prefetching techniques

The simplest form of instruction prefetching takes the form of utilizing longer cache lines. When a line is brought into the cache, future instructions are inserted into the cache before they are required by the CPU, reducing or eliminating miss delays. The disadvantages of this approach include increased memory traffic, together with more cache pollution due to the larger replacement granularity. Clearly, with this approach, cache misses can still occur when the program counter requests the instruction following the end of the current cache line.

#### 4.2.1 Next line prefetching schemes

A simple extension of this approach is the next line prefetching scheme (also known as ‘one block lookahead’ or OBL) [Smith, 1982]. This scheme attempts to fetch the next sequential cache line before it is needed by the processor. The prefetch for the next line is triggered when the program counter reaches a predefined point (known as the *fetchahead distance*, and measured from the end of the current cache line) in the current line. The effectiveness of this method depends greatly on the correct choice of fetchahead distance. With a small distance, the fetch request may not arrive in time to overcome memory latencies, while a large distance may increase cache pollution and result in useless prefetches.

A simple extension to next line prefetching, called *n*-line prefetching [Smith, 1982] attempts to increase the performance of the scheme. Here, the next *n* lines, rather than a single line, following the current line are brought into the cache. It has however been shown [VanderWiel and Lilja, 1997] that the additional traffic and cache pollution means that this form of prefetching is infeasible for values of  $n > 1$ .

The hardware required to implement these prefetching schemes is minimal due to their simplicity. Furthermore, even with their shortcomings, it has been shown that they can reduce CPU stalls due to memory latencies by up to 50%.

All of these methods take advantage of the spatial locality in most programs, making use of the fact that instructions that are executed soon after each other are very often close to each other in memory. They suffer from the fact that a branch instruction on processors implementing these schemes can easily cause a cache miss. These approaches implicitly make use of the assumption that any conditional branches will not be taken.

#### 4.2.2 Table-based prefetching

Target line prefetching enhances the accuracy of next line prefetching at the cost of additional complexity. The scheme takes note of the fact that unconditional jump and subroutine calls very often have fixed targets, and that conditional branches very often follow the same path as they followed when they were last executed. It therefore makes use of a heuristic that states that the prefetch behavior at

the current line should be based on the previous behavior of the current line. This method therefore prefetches the line that was requested next the last time the current line was executed. To implement this approach, a target prefetch table is used. This table, maintained in hardware, contains a list of tuples. Each tuple consists of the current line, and its successor. When the program counter moves from one cache line to another, two things occur in the prefetch table. Firstly, the successor entry of the previous line is updated to point to the new line. Secondly, the table is examined to determine if the new line has a successor. If this is the case, and the successor is not in the cache, a prefetch request is issued so as to bring the successor into the cache.

Hsu and Smith [Smith and Hsu, 1992] propose a scheme that combines target line prefetching and next line prefetching. This scheme implements next line prefetching as described above, together with a slightly modified form of target line prefetching: In their implementation, if the successor line was the next sequential line, it was not moved to the target prefetch table. This was done to save space within the table. Their results indicate that this scheme's effectiveness is approximately equal to the effectiveness of next line and target line prefetching combined. In the same paper, Hsu and Smith also investigated a modified target line prefetching approach. Here, instead of associating a cache line address with a succeeding cache line in the target prefetch table, an instruction address was stored. This was done in case a single cache line contained more than one branching instruction. Increases in performance were only found with very small caches combined with very large tables. For any other cache configurations, no significant performance increases were noticeable, with ordinary target prefetching very often outperforming this scheme.

When accessing a line for the first time, table-based approaches are unable to avoid compulsory misses, due to the fact that the table must still be set up. Furthermore, the table is often poorly used: the prefetch scheme is only utilized when a successor line that has been removed from the cache is required, when this is not the case (i.e. the line is still in the cache), no prefetch will be issued, but a table lookup still occurs, and table space is still occupied.

### 4.2.3 Wrong path prefetching

To overcome these deficiencies, Pierce and Mudge [Pierce and Mudge, 1996] propose a prefetching method which they refer to as 'wrong path prefetching'. This approach is similar to the hybrid scheme described previously, in that it combines both next and target line prefetching. Next line prefetching works as previously described, with an alternative method of implementing target prefetching being the heart of the scheme.

Wrong path prefetching does not utilize a target prefetch table. Instead, the line containing a branch target is prefetched immediately after the branch is recognized during the instruction decode stage. This means that both

directions of a branch are fetched using this scheme: The fall thru direction is fetched using next line prefetching, with the other direction fetched using target line prefetching.

Since the target prefetch only occurs after the decode stage, the prefetch is useless if the branch is taken, as there is not enough time to bring the desired memory into the cache if it is not already present. Similarly, using this type of target prefetching for unconditional jumps has no advantage. It should be clear that this approach will only have time to execute a useful prefetch for the path that is not taken (hence the name). This scheme becomes useful if the branch which had previously not been taken is taken at a slightly later stage. When this occurs, the new path should already have been prefetched into the cache.

The hardware requirements for this scheme are on par with those of next line prefetching, as the target addresses are generated by the already existing decoder, and no table is needed for address storage. The scheme does have a number of drawbacks: sometimes, this approach will bring lines into the cache that will not be used for a significant amount of time. The scheme therefore significantly increases cache pollution and memory traffic. The authors believe that this scheme will perform better than most other schemes for systems with long memory latencies coupled with fast CPUs. The simulations carried out by the authors show that wrong path prefetching improves performance by up to 14% over a machine with no prefetching, and performs better than next line prefetching and target prefetching. Furthermore, 75% of lines inserted into the cache by following the wrong path end up reducing miss times. As memory latencies increased, performance improvements rose up to 18%, this case is very important, as it reflects the increasing speed difference between CPU and memory.

### 4.2.4 Branch prediction based prefetching

As seen previously, it is difficult to use branch prediction methods to hide the memory latency of an instruction cache miss, as branch prediction is normally done at the instruction decode stage. In this subsection, we examine a number of schemes that run ahead of the main processor in some way, and can therefore initiate a prefetch of the instruction stream sufficiently early to hide memory latencies.

Young and Shekita [Young and Shekita, 1993] propose a scheme (also described in [Ebenezer, 1998]) built around predicting branches in program flow rather than individual instructions. This is achieved by grouping the program into flow blocks of equal size. The scheme performs branch prediction as soon as the first instruction from a flow block is accessed, trying to determine which flow block will be executed next, and prefetching that block. It is expected that the time spent in the block will be sufficient to prefetch the next block accessed. The fine tuning of the size of the flow block is very important in this scheme: small blocks may be too small to hide the

memory access latencies, while large blocks will yield inaccurate branch predictions as more than one branch instruction may appear within a block.

The scheme needs to maintain a flow block prediction table which keeps track of the flow-block address tag, the last branch target address, the flow block's history, and an indirect branch bit. To improve the accuracy of the scheme when using large flow blocks, it is possible to store more branch target addresses per flow block.

The paper also suggests that by keeping a separate, small and highly associative cache for lines prefetched by this scheme, it is possible to overcome many of the effects of cache pollution. Lines are only brought from this cache into the instruction cache if the line is actually accessed by the processor. Clearly, this approach will work for many other instruction prefetching methods.

Tests of this method were carried out using traces of a number of runs of commercial RDBMS systems, with an IBM RISC System/6000 hardware model. The metric used was the time to execute the benchmarks, and the results indicate a 60% speedup over the no prefetching approach, and a 20% time improvement over next line prefetching. One other interesting result was that eliminating misses altogether gave a further performance increase of 21 – 30%, indicating substantial further room for improvement. The paper indicates that the relative performance of this approach will improve further over time, as the processor clock rate to memory latency gap grows, and as the number of cycles per instruction decreases.

Chen et al. [Chen et al., 1997] propose another instruction prefetching method based on branch prediction. This method is based on their work done on data prefetching [Chen and Baer, 1995] (which will be described in the next section). This scheme utilizes a small autonomous prefetching unit that speculatively traverses the instruction stream as fast as possible and prefetches all instructions encountered along its path. If a branch is found, the likely path is predicted, recorded in a log, and execution then continues in this direction. As the main processor encounters branches, it checks the log for incorrect branch predictions. If an incorrect branch is found, the position of the prefetching unit is reset. The hardware required for this approach includes a program counter, a branch history table, an adder, and a return address stack.

Initially the prefetching unit's program counter is equal to the main execution unit's program counter. The prefetching unit is able to fetch one cache line per cycle from the L2 cache, but only fetches the next cache line when it has finished processing the current line. The entire cache line is then examined by the prefetching unit, and detects the first branch using predecoded information, or a few bits of the instruction's opcode. The target address of the branch is computed within the same cycle. If the branch is an indirect branch, the unit stalls and waits for the processor to catch up. Otherwise, the unit continues in the predicted direction: if a branch is predicted to not be taken, the unit examines the next branch in the line (in the next clock cycle), and if a branch is taken, its line

is brought into the cache in the next clock cycle, and the process begins again.

Whenever a branch is encountered by the prefetching unit, the unit's decision on whether to take the branch or not is recorded in a FIFO buffer. When the processor resolves the branch, the outcome is compared to the record in the buffer. If they match, the item is removed from the log, and execution continues. If a difference is detected (i.e. the prefetch unit has gone down an incorrect path), the log is flushed, the prefetch unit's program counter is reset, and the contents of the branch history register and return address stack are set to equal those of the main processor.

One other condition that needs to be checked for is whether the prefetch unit has fallen behind the current position of the main program counter. This can be detected by finding an empty log when the execution unit attempts to resolve a branch. If this occurs, the prefetch unit is reset to match the execution unit.

This scheme can only run ahead of the main execution unit if it can process more instructions per clock cycle than the main unit can handle. If the average length of a basic block were less than the issue rate of the processor, this scheme would not work. One way to apply this work to a highly superscalar machine would be to employ multiple branch prediction. One other enhancement to the basic scheme discussed in the paper was the addition of next  $n$  line prefetching. These prefetches were given very low priority, taking place only when the memory bus was free, and enhanced the performance of the scheme.

The scheme was tested against a part of the SPEC CINT95 benchmark suite. The results indicate that this scheme is up to 32% better than next- $n$  line prefetching, and between 34% and 44% better than wrong path prefetching when comparing stall overheads. Furthermore, this method generates more useful prefetches, and generates them earlier than the two other schemes. Its memory bandwidth utilization is comparable to that of next- $n$  line prefetching.

### 4.3 Data prefetching techniques

Data prefetching differs from instruction prefetching in a number of ways: firstly, access to data does not often occur every clock cycle, a program usually obtains some data, operates on it for some time, and then accesses more data. Secondly, spatial data access patterns are very rarely continuous, i.e. if a element of data is accessed at memory address  $x$ , it is very likely that the next piece of data accessed will not be at  $x + 1$ . This contrasts with instruction accesses, where memory is accessed in a continuous pattern until a branching instruction is reached. Lastly, data accesses can both read and write from/to the cache, while instructions are normally only read. These differences imply that a different approach to data prefetching is required for effective memory latency hiding.

We believe that it is possible to subdivide most data prefetching methods into one of four categories:

- *Sequential prefetching* As described previously, se-

quential prefetching involves bringing the block following the referenced block into the cache.

- *Branch prediction based prefetching* is very similar to the techniques described above for instruction prefetching.
- *Stride prefetching techniques* are aimed at providing prefetching for data accesses with a non sequential, but linear access pattern. A prime example of this type of access pattern occurs in matrix operations, where nested loops iterate over large amount of data.
- *Dependance based prefetching techniques* [Roth et al., 1998] attempts to prefetch linked data structures (such as linked lists, trees and graphs) which, due to their pointer based nature, other techniques have difficulty in prefetching.

Other techniques, which do not fit into the above categories do exist, and will be discussed.

### 4.3.1 Sequential prefetching

Sequential data prefetching methods are amongst the simplest and oldest of prefetching techniques. Sequential prefetching applies to data in the same manner as was described previously for instruction prefetching, with a number of minor differences: in data prefetching, no fetchahead distance exists. Instead, VanderWiel [VanderWiel and Lilja, 1999b] describes two approaches, originally summarized by Smith [Smith, 1982]. The prefetch-on-miss algorithm prefetches the next block as soon as an access to the current block results in a cache miss, while tagged prefetching associates a tag bit with every memory block. This bit is used to detect when a block is referenced for the first time (even if it is already in the cache), and when this occurs, the next sequential block is fetched. It was found that tagged prefetching is more than twice as effective as prefetch-on-miss in reducing miss ratios. This is due to the fact that a strictly sequential access pattern will cause every second block to miss with prefetch on miss, while only the first block will cause a cache miss with tagged prefetching.

If memory latency times are large, it is possible to increase the number of blocks prefetched, as in next  $n$  line prefetching.

One extension of sequential prefetching is adaptive sequential prefetching [VanderWiel and Lilja, 1997], which allows  $n$  to vary during program execution so as to closely follow the amount of spatial locality exhibited by the program during a specific point in its execution. This scheme operates using a prefetch efficiency metric. This is defined as the ratio of useful prefetches to the total number of prefetches. A prefetch is deemed useful whenever a request for data results in a cache hit within a prefetched block. The value of  $n$  is initialized to 1, and incremented when the prefetch efficiency increases over a predefined threshold value. Similarly, it is decremented whenever the efficiency drops below a fixed value. If  $n$  is reduced to 0,

prefetching is disabled. If this occurs, the prefetch mechanism begins to monitor how often cache misses to block  $b$  occur while block  $b - 1$  is cached, and restarts prefetching if the ratio of these two numbers exceeds the lower threshold of prefetch efficiency. While this scheme greatly reduced the ratio of cache misses when compared with next line prefetching, the increased memory traffic brought about by this scheme nullified a lot of its advantage when performance in run-time simulations was measured.

### 4.3.2 Stream buffers

To avoid cache pollution, Jouppi [Jouppi, 1990] proposed using a separate stream buffer to hold the prefetched cache blocks. This buffer operates on a FIFO basis: when the first buffer entry is referenced, it is brought into the cache, and a new block is prefetched into the tail position. Since the data is not immediately put into the cache, cache pollution is avoided, but if a cache miss occurs, and the desired block is also not at the head of the stream buffer, the buffer is flushed. The stream buffer consists of a number of entries, each containing a tag bit, an available bit (which is set to true once the data is in the stream buffer), and the data itself. Stream buffers appear very effective for instruction prefetching, removing 72% of instruction cache misses, but only removed 25% of data cache misses. The poor data cache performance was attributed to data being required from a number of sources. To enhance performance, the paper therefore suggests using a multi-way stream buffer, constructed out of four stream buffers in parallel. When a miss occurs that also misses all stream buffers, the least recently accessed stream buffer is cleared and begins to fetch at the new address. This improvement, on average, almost doubles the performance of this technique, removing 43% of data cache misses for the LINPACK benchmark. Another improvement that can be implemented is searching for the desired data anywhere within the stream buffer, rather than just at its head. This approach, referred to as a *quasi-sequential stream buffer*, reduces data cache misses by a total of 47% on average. This added accuracy comes at the cost of much more complex hardware, and it is argued that the added accuracy is not worth the extra cost.

Stream buffers have also been explored as an alternative to secondary cache memory [Palcharla and Kessler, 1994]. By using eight stream buffers, and fetching the two blocks following a cache miss, they discovered that between 50% and 90% of data requests that missed in the primary cache were satisfied by the stream buffers. To overcome the large number of unnecessary prefetches generated by this approach, a small history buffer was used to record primary cache misses. When this buffer indicates that a miss occurred for both blocks  $b$  and  $b + 1$ , a stream is allocated (based on a least recently used policy) to block  $b + 2$  and onwards. This technique has been used as an alternative to large secondary caches, and was incorporated into the Cray T3E processor.

Sequential prefetching techniques are useful in that they require no modifications to existing programs, and can be implemented with relatively modest hardware. Unfortunately, these methods perform badly compared to software prefetching when non-sequential memory access patterns exist. Any sort of access pattern which does not exhibit a high level of spatial locality will result in unnecessary, and often useless prefetches.

### 4.3.3 Stride and branch prediction based prefetching

Stride prefetching attempts to detect some sort of constant stride in a data access pattern, and to then prefetch based on this pattern. Accesses to data with a constant stride often occur in array accesses originating in a loop. The detection of patterns is done by comparing the targets of successive load and store instructions.

Chen and Baer [Chen and Baer, 1995] break down data accesses in nested loops into four categories:

1. *scalar*, where the same variable is repeatedly referenced.
2. *zero stride*, where the same element of memory is repeatedly referenced at the same level of the loop. The difference between this and a scalar reference is that here, the reference is an invariant only within an inner loop, it is modifiable by an outer loop.
3. *constant stride*, where the memory address changes in a linear manner based on the depth of the loop. The most common example of this is a for loop iterating over an array.
4. *irregular stride*, consists of all accesses which do not fall into the above categories, for example, linked list implementations would most probably fall into this category.

They propose three schemes of increasing complexity for data prefetching, with all the schemes attempting to satisfy a number of goals: Firstly, scalar, zero, and constant stride accesses should all be prefetched. Irregular access patterns should not generate useless prefetches. Lastly, the methods should not increase the cycle time of the processor for which they are implemented.

All three schemes are built around a reference prediction table (RPT) that stores access patterns of load/store instructions. The first scheme executes a prefetch for data that should be used in the next iteration when it is accessed during the current iteration. The second scheme, referred to as the *lookahead* prefetching method, uses a look ahead program counter to follow the probable execution path and trigger the prefetches. Finally, the *correlated* scheme attempts to handle data accesses correlated according to the level of the loop.

Using the first two schemes, the code listing in figure 1 would cause an incorrect prefetch to occur every time the inner *j* loop would finish. Correlated prefetching associates the change in branch logic with the end

```
char A[100,100];

for (int i=0;i<100;i++)
  for (int j=0;j<i;j++)
    k+=A[i][j];
```

Figure 1: Triangular memory access pattern

of the loop, and would therefore not issue the wrong prefetch. Benchmarking showed very little difference between the performance of lookahead prefetching, and correlated prefetching. Both methods noticeably outperformed the basic scheme. Only lookahead prefetching will be described here.

From an implementation point of view, the hardware required for lookahead prefetching is a superset of that required for the basic prefetching scheme.

Lookahead prefetching requires the following hardware:

- A Look ahead program counter (LA-PC), consisting of some form of pointer to an instruction in memory.
- A branch prediction mechanism, consisting of a branch prediction table (BPT). The branch prediction features of the main program counter are also utilized.
- A Reference Prediction Table to store access patterns.
- A method to store the outstanding request list (ORL) to prevent duplicate prefetches from taking place.

As the program executes, the processor's program counter (PC) will encounter branch and data read/write instructions. In the former case, a record of the branches will be kept in the BPT, while in the latter case, information will be recorded in the RPT.

The LA-PC acts as a program counter for the prefetching unit, and attempts to stay  $\delta$  cycles ahead of the real PC, where  $\delta$  is the number of cycles required to bring data into the cache. Initially, the LA-PC is set to point to the instruction following the current PC, and is incremented by 1 during each cycle. When the processor stalls fetching data, the LA-PC will continue incrementing, until it is sufficiently far away from the real PC, once it has reached this distance, it will attempt to maintain it by stalling as necessary.

If the instruction pointed to by the LA-PC is contained within the BPT, it is a branch. The processor's branch prediction mechanism will then be used to determine the new position for the LA-PC. If it is later determined that the branch was incorrectly predicted, the LA-PC is reset to point one cycle ahead of the PC.

The RPT contains a number of fields: A *tag* field is used to store the address of the instruction. The *times* field records how many iterations the LA-PC is ahead of the PC. This field is incremented each time the same RPT entry is accessed by the LA-PC. It is decremented each time the PC encounters the RPT entry, and set to 0 if the predicted prefetch address is incorrect. The *previous address* tag stores the address of the operand that the instruction

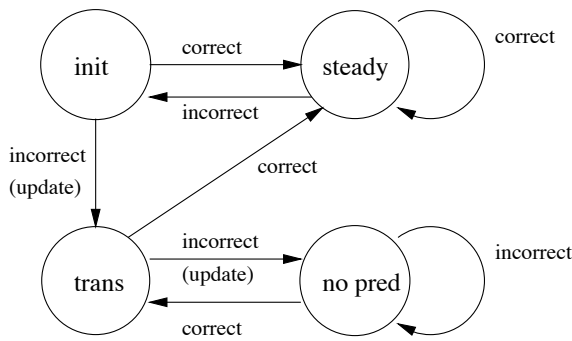


Figure 2: A graphical representation of possible states and their associated transitions.

last accessed. A *stride* entry keeps track of the difference between the last two operand addresses. A two bit *state* field associates the entry with one of four states.

An entry may be in one of four possible states:

- *initial* which occurs when an item is first loaded into the RPT, or when a misprediction for this item has occurred.
- the *transient* state is entered when the system is unsure whether the predicted stride is correct. The stride is calculated when in this state by subtracting the previous operand address from the current address.
- *steady* indicates a stable state, it occurs while the prediction is correct.
- *no prediction* occurs when no stride pattern can be detected. It is entered from the transient state when a stride calculation proves incorrect.

Therefore, when a load/store instruction is first encountered, it is recorded in the RPT and associated with the *initial* state. When the LA-PC encounters the instruction for the second time, it computes a stride, and the entry's state is set to *transient*. When the instruction is next detected, its stride is recalculated. If it matches, the state changes to *steady*, and will remain so until a different stride is encountered. When this occurs, it is reset to the *initial* state. If a different stride is encountered while in the *transient* state, the entry will be moved into the *no prediction* state. The state will then shift between *transient* and *no prediction* states until a pattern is detected. It is possible to move from the *initial* state to the *steady* state if a stride is already associated with the entry, and the stride proves to be correct. This is illustrated in figure 2

Using the RPT, the predicted prefetch address is equal to  $(prev\_address + stride \times times)$ . A prefetch is generated if the entry is in the *initial*, *transient* or *steady* states. A prefetch is only generated if the request is not already in the cache, and is also not present in the ORL. If the ORL is full, the LA-PC will stall.

In the course of normal processing, if a cache miss occurs, the cache controller first checks the ORL for the presence of the request. If the request is present, then the processor will stall, but for a shorter length of time than

if no prefetching had occurred. If the request is not in the ORL, then a normal load request will be issued, and will be carried out at a higher priority than any outstanding prefetching requests.

The performance benefits gained by implementing lookahead prefetching with a moderately sized RPT greatly outweigh those gained by expending the same resources in increasing the size of the data cache.

Roth et al. [Roth et al., 1998] present a branch prediction based scheme which handles prefetching in pointer intensive applications. Like other BP based schemes, this approach runs ahead of the program counter, and attempts to prefetch data before it is requested. This approach is novel in that it tries to discover load operations whose results are in turn used as targets of further load operations. This producer-consumer access pattern is typical of pointer intensive applications. In a manner similar to that described in the previous approach, a table is used to store the addresses of possible producer-consumer pairs. When such a relation is detected, a prefetch for the result of the load of a producer address can be issued.

Using a pointer intensive benchmark, this scheme proved very accurate, and load times were reduced by up to 25%. This improvement is much greater than would be achieved by adding an additional 32KB to the data cache, at approximately the same hardware complexity cost. The memory bandwidth between the L1 and L2 caches was on average 15% greater than when no prefetching occurred.

#### 4.3.4 Markov prefetching

A very different approach to prefetching is illustrated in a paper by Joseph and Grunwald [Joseph and Grunwald, 1997]. This prefetch engine is designed to act as an interface between on-chip and off-chip cache, and can therefore be added to existing computer designs. Furthermore, it is capable of prefetching both instructions and data. This prefetching scheme can successfully generate prefetches for data accesses where most other prefetching schemes would fail. Unlike other prefetching schemes, this scheme issues multiple prefetch requests simultaneously, with different priorities associated with each request. Another noticeable difference between this method and the previously described methods is that Markov prefetching requires a large amount of memory to operate. In their original paper, Joseph and Grunwald's prefetcher had 1 megabyte of memory available for its data structures. Lastly, their work assumes that prefetched data does not displace blocks currently in the cache by making use of a small on chip prefetch buffer.

As the name implies, Markov prediction based prefetchers utilize a modified Markov model to determine what blocks should be brought in from the higher level cache. A Markov process is a stochastic system for which the occurrence of a future state depends on the immediately preceding state, and only on it. A Markov process can be represented as a directed graph, with probabilities

Figure 3: A sample cache miss stream, with different letters indicating different memory references.

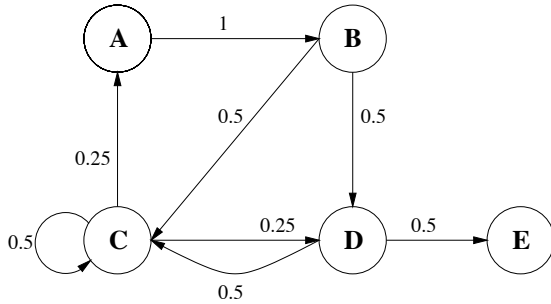


Figure 4: The Markov model created from the miss stream described in figure 3

associated with each vertex. Each node represents a specific state, and state transition is described by traversing an edge from the current node to a new node.

The prefetcher attempts to build a Markov model of the miss stream originating from the on chip cache. As an example, the miss stream described in figure 3 may result in the Markov model described in figure 4. This example uses one previous reference to predict the next reference, but more generally, it is possible to use a  $n$ -history Markov model. The prefetching performance of the more complex model is not significantly better than that of the 1-history model, and for this reason, only 1-history models are used.

If the same program were to be run again and issue the same memory references, it should be possible to use the Markov model to predict miss references. A number of problems prevent the use of a pure Markov model for prediction. In real world scenarios, programs don't repeat the same memory reference pattern between executions, and the transitions 'learned' by the Markov model during one run may not carry through to the next. Also, problems exist with representing Markov models in memory: transition probabilities are real values, each node may have many possible transitions, and the number of nodes may be too large to easily store and manipulate.

To make the Markov approach feasible, a number of design decisions had to be taken. Firstly, to overcome the problem of relearning, the scheme continuously rebuilds and utilizes the model as the program executes. Limits are placed on the number of next states a single node can have, and an upper limit is also placed on the number of possible nodes. More concretely, a table is set up with a miss address field, which stores the address for which a fetch request has been issued. Associated with this miss address are a fixed number of next address prediction registers. The number of entries in the table is limited by the amount of memory allocated to the prediction scheme.

When a cache miss occurs, and the address requested is in the miss address field, fetches requests may be issued for all addresses in the next address prediction registers. Requests are stored in a prefetch request queue, together with an associated priority, and high priority requests may

displace lower priority requests. The request queue is also utilized by fetches from the processor, and these fetches have a higher priority than any of the requests from the prefetching mechanism. If the request queue fills up, low priority requests are discarded. Requests fulfilled on behalf of the prefetching unit are put into the on chip prefetch buffers, while processor requests are placed into the on chip cache as usual. If the prefetch buffer is full, entries are discarded in using a least recently used scheme.

Priorities can be calculated in a number of ways. It was discovered that associating the priority with the age of the reference produced the best results. To do this, the next address registers for each miss address were ordered by age. The youngest register was associated with the most recent successfully used prefetch, and has the highest priority. When another prefetch associated with the miss address is used by the processor, its address is inserted into the youngest register, and the other addresses are shifted towards the oldest register. The address occupying the oldest register may thus fall out. This scheme proved much more successful than using the true probabilities, and has the added advantage of simplicity.

The Markov prefetcher was able to reduce the number of cycles spent accessing memory per instruction (MCPI) by an average of 54%. While this means that this prefetching scheme is not as effective as some of the previously discussed schemes, the coverage of this method and correct timing of its prefetches are much better than those of the previous techniques. Furthermore, because of its position in the memory hierarchy, it is very easy to use Markov prefetching together with other prefetching schemes, possibly gaining the advantages of both.

#### 4.3.5 Hybrid schemes

Hybrid prefetching schemes, combining both hardware and software components have been proposed numerous times. Chen [Chen, 1995] proposed an extension to his previously described reference prediction table scheme consisting of a programmable prefetch engine. In this scheme, the tag, address, and stride information are inserted into the table by the program just before it enters loops that will be able to take advantage of the prefetch scheme. Once programmed, prefetches occur when the processor's program counter matches one of the tag fields in the prefetch unit.

VanderWiel and Lilja [VanderWiel and Lilja, 1999a] propose a hybrid approach using a Data Prefetch Controller (DPC). The DPC works in parallel with the main processor by issuing prefetch requests on its behalf.

The DPC consists of a small processor, together with its own cache. The DPC's instruction set consists of simple integer instructions that can be used to calculate addresses, together with a prefetch instruction. The main processor controls the DPC by writing data and control words to a set of memory mapped registers associated with the DPC. Data is passed to the DPC during run time to enable various address calculations, and control instructions can be used

to toggle between a number of prefetch strategies. Prefetch methods are inserted into the DPC by a separate program that is generated by the compiler when the main executable is compiled.

Once the prefetch stream has been initiated by the main processor, the DPC issues prefetch requests on behalf of the processor to a shared L2 cache. Prefetched blocks are inserted into both L1 and L2 caches. To prevent the DPC from polluting the cache with requests required too far into the future, a control mechanism is suggested whereby the DPC will fetch a number of blocks and tag one of these a trigger block. A trigger queue is used to store the addresses of the trigger blocks on a FIFO basis. Once the queue is full, the DPC will stall. When a block is accessed by the main processor, its address is compared with the first address stored in the trigger queue. If they match, the address will be dropped from the queue, and all other addresses in the queue will be moved forward. On a match, a signal is also sent to the DPC, enabling it to continue prefetching.

To prevent redundant prefetches from occurring, a small filter buffer is used. The filter buffer retains the last few recently prefetched addresses. If a prefetch request is found to have the same address as an entry in the filter buffer, it is discarded. A filter buffer containing 256 entries was found to reduce redundant prefetches to less than 5% of all issued prefetches.

Benchmarks were simulated on runs of part of the SPECfp95 suite, and compared to hardware based RPT prefetching and a compiler generated software prefetching scheme. The hybrid approach gave better prefetch accuracy, and higher prefetch coverage. Furthermore, in most cases, L2 cache miss rates were significantly improved by this approach, while for most cases, a moderate improvement over the other two approaches was seen in the L1 cache miss rate. In almost all cases, some improvement in overall program execution times was noted. To justify the additional cost of hardware when prefetching using this approach, the benchmarks were also run against a simulated software prefetching system with double the L1 data cache. Performance improvements between the smaller and larger data cache systems were only noted in one benchmark. Therefore, the additional costs caused by the DPC approach appear to provide greater benefits than those gained by doubling the data cache size.

## 5 Conclusions

Prefetching schemes are diverse. A number of divisions can be proposed in an attempt to classify them, but many schemes may fall into a number of categories. Classification is useful as specific types of schemes may be applicable to specific situations.

Once a prefetching scheme is chosen, it is natural to attempt to compare it to other schemes. Unfortunately, no standard metric exists to compare the effectiveness of various schemes, and the development of such a measurement

is hindered by the many architectural assumptions made by most methods.

Even with all these issues, some general statements can be made. Instruction prefetching schemes appear to be converging on branch prediction based schemes, with improvements being incremental rather than revolutionary. Instruction prefetching techniques also appear to be relatively mature, with some schemes achieving performances moderately close to the theoretical best.

Data prefetching techniques are still very varied, with most schemes performing well for only very specific data access patterns. Hardware based data prefetching for general programs appears to be an understudied but potentially very beneficial area of research.

With the increasing gap between memory speeds and processor speeds showing no sign of slowing down, it appears as if prefetching will play a larger role in system performance. Due to the nature of the instruction scheme, a small increase in instruction prefetching efficiency may yield noticeable performance benefits. It should also be apparent that the potential exists for large improvements in the capabilities of data prefetching schemes. Therefore, the need for new prefetching schemes is likely to continue, keeping it an active area of research.

## References

- [Acquaviva, 1999] Acquaviva, J. (1999). Data prefetching efficiency on two commercial systems. In *Proceedings of the fifth European SGI/Cray MPP Workshop*.
- [Chen et al., 1997] Chen, I., Chih-Chieh, L., and Mudge, T. (1997). Instruction prefetching using branch prediction information. In *International conference on Computer Design, VLSI in Computers and Processors*, pages 593–601.
- [Chen, 1995] Chen, T.-F. (1995). An effective programmable prefetch engine for on-chip caches. In *Proceedings of the 28th International Symposium on Microarchitecture*, pages 237–242.
- [Chen and Baer, 1994] Chen, T.-F. and Baer, J.-L. (1994). A performance study of software and hardware prefetching schemes. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 223–232.
- [Chen and Baer, 1995] Chen, T.-F. and Baer, J.-L. (1995). Effective hardware-based data prefetching for high-performance processors. *IEEE transactions on computers*, 44(5):609–623.
- [Ebenezer, 1998] Ebenezer, A. (1998). Hardware based prefetching methods. Masters project, University of Minnesota, Department of Electrical and Computer Engineering.

- [Hennessy and Patterson, 1996] Hennessy, J. L. and Patterson, D. A. (1996). *Computer Architecture: A Quantitative approach*. Morgan Kaufman Publishers.
- [Joseph and Grunwald, 1997] Joseph, D. and Grunwald, K. (1997). Prefetching using markov predictors. In *Proceedings of the 24th international symposium of computer architecture*, pages 252–263.
- [Jouppi, 1990] Jouppi, N. P. (1990). Improving direct mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *The 17th Annual International Symposium on Computer Architecture*, pages 364–373.
- [Machanick et al., 1998] Machanick, P., Salverda, P., and Pompe, L. (1998). Hardware-software trade-offs in a direct rambus implementation of the RAMpage memory hierarchy. In *Proc. ASPLOS-VIII Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 105–114.
- [Metcalf, 1993] Metcalf, C. (1993). Data prefetching: a cost/performance analysis. <http://www.incert.com/~metcalf/papers/prefetch/>.
- [Palcharla and Kessler, 1994] Palcharla, S. and Kessler, R. (1994). Evaluating stream buffers as a secondary cache replacement. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 24–33.
- [Pierce and Mudge, 1996] Pierce, J. and Mudge, T. (1996). Wrong path instruction prefetching. In *Proceedings of the international symposium on microarchitecture*, pages 165–175.
- [Postiff et al., 1999] Postiff, M., Tyson, G., and Mudge, T. (1999). Performance limits of trace caches. *Journal of Instruction-Level Parallelism*. <http://www.jilp.org/vol1>.
- [Roth et al., 1998] Roth, A., Moshovos, A., and Sohi, G. S. (1998). Dependence based prefetching for linked data structures. In *Proc. ASPLOS-VIII Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 115–126.
- [Smith, 1982] Smith, A. J. (1982). Cache memories. *ACM Computing Surveys*, pages 473–530.
- [Smith and Hsu, 1992] Smith, J. E. and Hsu, W. (1992). Prefetching in supercomputer instruction caches. In *Proceedings of the 1992 conference on Supercomputing*, pages 588–597.
- [VanderWiel and Lilja, 1999a] VanderWiel, S. and Lilja, D. (1999a). A compiler-assisted data prefetch controller. In *IEEE International Conference on Computer Design*.
- [VanderWiel and Lilja, 1997] VanderWiel, S. P. and Lilja, D. J. (1997). When caches are not enough: Data prefetch techniques. *IEEE Computer*, pages 23–27.
- [VanderWiel and Lilja, 1999b] VanderWiel, S. P. and Lilja, D. J. (1999b). Data prefetch mechanisms. To be published in *ACM Computing Surveys*, available at <http://www-mount.ee.umn.edu/svw/survey.ps>.
- [Young and Shekita, 1993] Young, H. and Shekita, E. (1993). An intelligent i-cache prefetch mechanism. In *Proceedings of the IEEE international Conference on Computer Architecture*, pages 44–49.