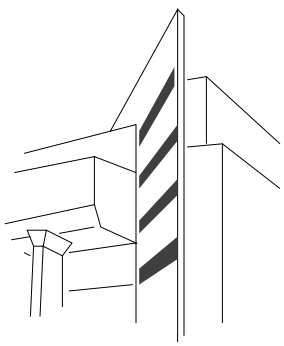
A Survey of Data Prefetching Techniques

Technical Report No: HPPC-96-05 October 1996

Steve VanderWiel
David J. Lilja



UNIVERSITY OF MINNESOTA
High-Performance Parallel Computing Research Group
Department of Electrical Engineering • Department of Computer Science • Minneapolis • Minnesota • 55455 • USA

A Survey of Data Prefetching Techniques

Steven VanderWiel
svw@ee.umn.edu

David J. Lilja
lilja@ee.umn.edu

Department of Electrical Engineering
University of Minnesota
200 Union St. SE
Minneapolis, MN 55455

ABSTRACT

The expanding gap between microprocessor and DRAM performance has necessitated the use of increasingly aggressive 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, it is still not uncommon for scientific programs to spend more than half their run times stalled on memory requests. Data prefetching has been proposed as a technique for hiding the access latency of data referencing patterns that defeat caching strategies. Rather than waiting for a cache miss to perform a memory fetch, data prefetching anticipates such misses and issues a fetch to the memory system in advance of the actual memory reference. To be effective, prefetching must be implemented in such a way that prefetches are timely, useful, and introduce little overhead. Secondary effects in the memory system must also be taken into consideration. Despite these obstacles, prefetching has the potential to significantly improve overall program execution time by overlapping computation with memory accesses. Prefetching strategies are diverse and no single strategy has yet been proposed which provides optimal performance. The following survey examines several alternative approaches and discusses the design tradeoffs involved when implementing a data prefetch strategy.

Note to the Reviewer *: The survey style of this paper has necessarily led to a larger number of references than is normally allowed in IEEE Computer articles. The authors ask your assistance in identifying any references which may be excluded, either because they seem unnecessary or because the referenced topic adds little to the paper.*

1. Introduction

The past decade has seen enormous strides in microprocessor fabrication technology and design methodologies. By most metrics, CPU performance has increased at a dramatic rate since the mid-eighties. In contrast, main memory dynamic RAM (DRAM) technology has increased at a much more leisurely rate, as shown in Figure 1. 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 [11].

Chief among the latency reducing techniques is the use of cache memory hierarchies. The static RAM (SRAM) memories used in caches have managed to keep pace with processor memory request rates but continue to cost about a factor of four more than DRAMs making them unacceptable for a main store technology. Although the use of large cache hierarchies has proven to be effective in reducing the access latency for the most frequently used data, it is still not uncommon for scientific programs to spend more than half their run times stalled on memory requests [19]. The large, dense matrix operations that form the basis of many scientific applications typically exhibit poor cache utilization due to a lack of locality (see Sidebar 1).

The poor cache utilization of scientific applications is partially a result of the "on demand" memory fetch policy of most caches. This policy fetches data into the cache from main memory only after the processor has requested a word and found it absent from the cache. The situation is illustrated in Figure 2a where computation, including memory references satisfied within the cache hierarchy, are represented by the upper time line while main memory access time is represented by the lower time line. In this figure, the data blocks associated with memory references r1, r2, and r3 are not found in the cache hierarchy and must therefore be fetched from main memory. Assuming the referenced data word is needed immediately, the processor will be stalled while it waits for the corresponding cache block to be fetched. Once the data returns from main memory it is cached and forwarded to the processor where computation may again proceed.

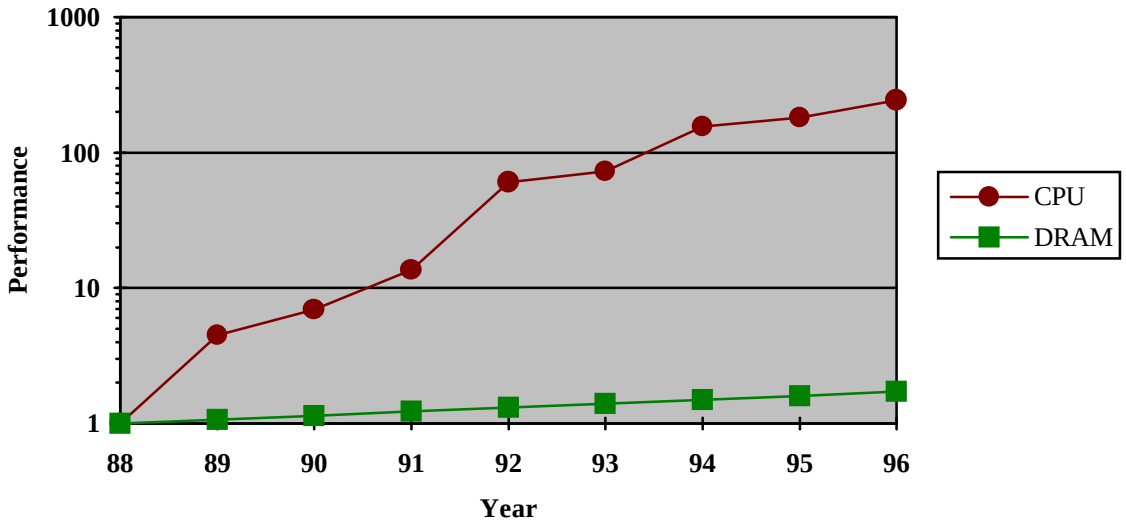


Figure 1. CPU and DRAM performance since 1988. CPU performance is measured by SPECfp92 and DRAM performance by row access times. All values are normalized to their 1988 equivalents (source: The CPU Info Center, http://infopad.eecs.berkeley.edu/CIC/).

Note that this fetch policy will always result in a cache miss for the first access to a cache block since only previously accessed data are stored in the cache. Such cache misses are known as *cold start* or *compulsory* misses. Also, if the referenced data is part of a large array operation, it is likely that the data will be replaced after its use to make room for new array elements being streamed into the cache. When the same data block is needed later, the processor must again bring it in from main memory incurring the full main memory access latency. This is called a *capacity* miss.

Many of these cache misses can be avoided if we augment the demand fetch policy of the cache with the addition of a data prefetch operation. Rather than waiting for a cache miss to perform a memory fetch, data prefetching anticipates such misses and issues a fetch to the memory system in advance of the actual memory reference. This prefetch proceeds in parallel with processor computation, allowing the memory system time to transfer the desired data from main memory to the cache. Ideally, the prefetch will complete just in time for the processor to access the needed data in the cache without stalling the processor.

An increasingly common mechanism for initiating a data prefetch is an explicit `fetch` instruction issued by the processor. At a minimum, a `fetch` specifies the address of a data word to be brought into cache space. When the `fetch` instruction is executed, this address is simply passed on to the memory system without stalling the processor to wait for a response. The cache responds to the `fetch` in a manner similar to an ordinary `load` instruction with the exception that the

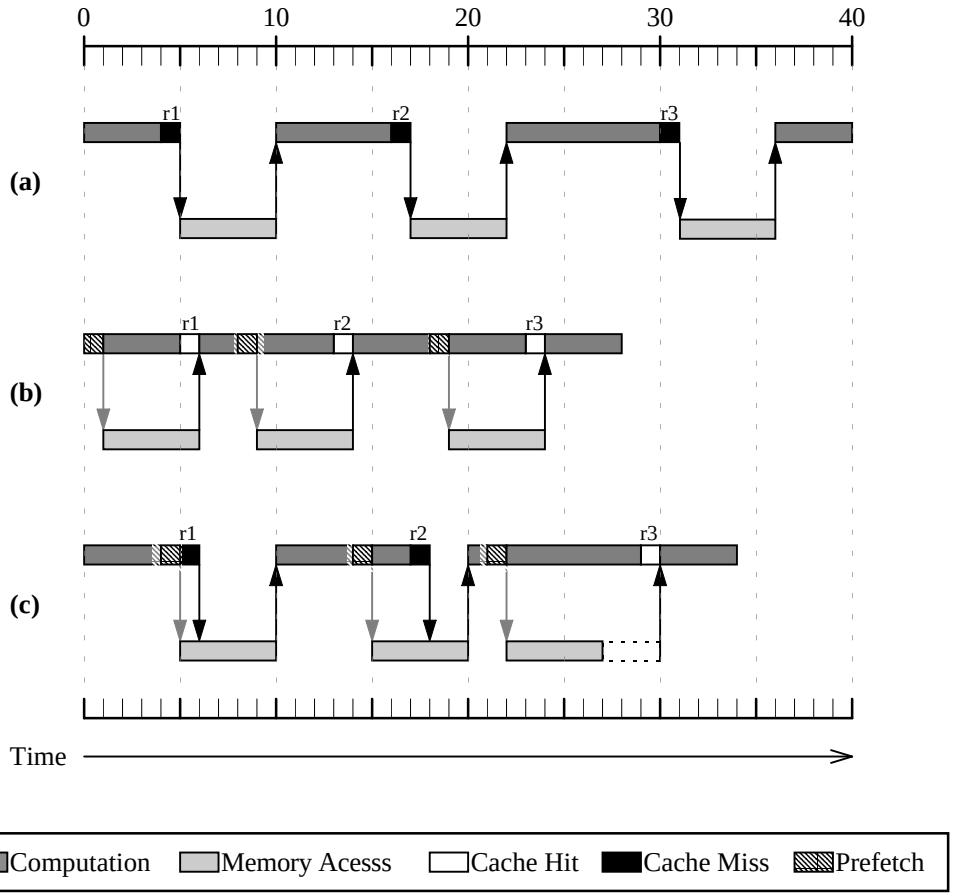


Figure 2. Execution diagram assuming a) no prefetching, b) perfect prefetching and c) degraded prefetching.

referenced word is not forwarded to the processor after it has been cached. Figure 2b shows how prefetching can be used to improve the execution time of the demand fetch case given in Figure 2a. Here, the latency of main memory accesses is hidden by overlapping computation with memory accesses resulting in a reduction in overall run time. This figure represents the ideal case when prefetched data arrives just as it is requested by the processor.

A less optimistic situation is depicted in Figure 2c. In this figure, the prefetches for references r1 and r2 are issued too late to avoid processor stalls although the data for r2 is fetched early enough to realize some benefit. Note that the data for r3 arrives early enough to hide all of the memory latency but must be held in the processor cache for some period of time before it is used by the processor. During this time, the prefetched data are exposed to the cache replacement policy and may be evicted from the cache before use. Moreover, the prefetched data may displace data in the cache that is currently in use by the processor.

This last situation is an example of *cache pollution*. Note that this effect should be distinguished from normal cache replacement misses. A prefetch that causes a miss in the cache that would not have occurred if prefetching was not in use is defined as cache pollution. If, however, a prefetched block displaces a cache block which is referenced after the prefetched block has been used, this is an ordinary replacement miss since the resulting cache miss would have occurred with or without prefetching.

A more subtle side effect of prefetching occurs in the memory system. Note that in Figure 2a the three memory requests occur within the first 31 time units of program startup whereas in Figure 2b, these requests are compressed into a period of 19 time units. By removing processor stall cycles, prefetching effectively increases the frequency of memory requests issued by the processor. Memory systems must be designed to match this higher bandwidth to avoid becoming saturated and nullifying the benefits of prefetching.

It is also interesting to note that software-initiated prefetching can achieve a reduction in run time despite adding instructions into the execution stream. In Figure 3, the memory effects from Figure 2 are ignored and only the computational components of the run time are shown. Here, it can be seen that the three prefetch instructions actually increase the amount of work done by the processor.

Several hardware-based prefetching techniques have also been proposed which do not require the use of explicit `fetch` instructions. These techniques will be referred to as *hardware-initiated* prefetching because they employ special hardware which monitors the processor in an attempt to infer prefetching opportunities. Although hardware-initiated prefetching incurs no instruction overhead, it often generates more unnecessary prefetches than software-initiated schemes. These unnecessary prefetches are a result of the need to speculate on future memory accesses without the benefit of compile-time information. If this speculation is incorrect, cache blocks that are not actually needed will be brought into the cache. Although these unnecessary prefetches do not affect correct program behavior, they can result in cache pollution and will consume memory

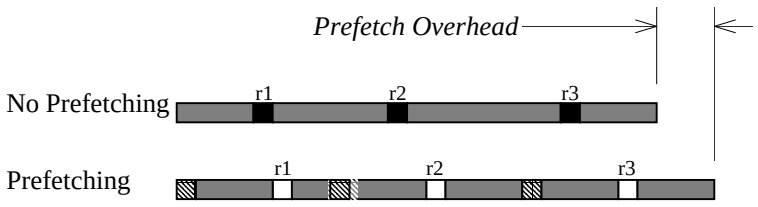


Figure 3. Software prefetching overhead.

bandwidth.

To be effective, data prefetching must be implemented in such a way that prefetches are timely, useful, and introduce little overhead. Secondary effects in the memory system must also be taken into consideration when designing a system which employs a prefetch strategy. Despite these obstacles, data prefetching has the potential to significantly improve overall program execution time by overlapping computation with memory accesses. Prefetching strategies are diverse and no single strategy has yet been proposed which provides optimal performance. In the following sections, alternative approaches to prefetching will be examined by comparing their relative strengths and weaknesses.

2. Background

Prefetching, in some form, has existed since the mid-sixties. Early studies [1] of cache design recognized the benefits of fetching multiple words from main memory into the cache. In effect, such block memory transfers prefetch the words surrounding the current reference in hope of taking advantage of the spatial locality of memory references. Hardware-initiated prefetching of separate cache blocks was later implemented in the IBM 370/168-3 and Amdahl 470/V. Software-initiated techniques are more recent. Smith alluded to the idea in his survey of cache memories [23] but at that time doubted its usefulness. Later, Porterfield [21] proposed the idea of a "cache load instruction" with several RISC implementations following shortly thereafter.

Prefetching is not restricted to fetching data from main memory into a processor cache. Rather, it is a generally applicable technique for moving memory objects up in the memory hierarchy before they are actually needed by the processor. Prefetching mechanisms for instructions and file systems are commonly used to prevent processor stalls, for example [24,20]. For the sake of brevity, only techniques that apply to data objects residing in memory will be considered here.

Non-blocking `load` instructions share many similarities with data prefetching. Like prefetches, these instructions are issued in advance of the actual use to take advantage of the parallelism between the processor and memory subsystem. Rather than loading data into the cache, however, the specified word is placed directly into a processor register. Non-blocking loads are an example of a *binding prefetch*, so named because the value of the prefetched variable is bound to a named location (a processor register, in this case) at the time the prefetch is issued. Although non-blocking loads will not be discussed further here, other forms of binding prefetches will be examined.

Data prefetching has received considerable attention in the literature as a potential means of boosting performance in multiprocessor systems. This interest stems from a desire to reduce the particularly high memory latencies often found in such systems. Memory delays tend to be high in multiprocessors due to added contention for shared resources such as a shared bus and memory modules in a symmetric multiprocessor (SMP). Memory delays are even more pronounced in distributed-memory multiprocessors where memory requests may need to be satisfied across an interconnection network. By masking some or all of these significant memory latencies, prefetching can be an effective means of speeding up multiprocessor applications (see Sidebar 2).

Due to this emphasis on prefetching in multiprocessor systems, many of the prefetching mechanisms discussed below have been studied either largely or exclusively in this context. Because several of these mechanisms may also be effective in single processor systems, multiprocessor prefetching is treated as a separate topic only when the prefetch mechanism is inherent to such systems.

3. Software-Initiated Data Prefetching

Most contemporary microprocessors support some form of `fetch` instruction which can be used to implement prefetching. The implementation of a `fetch` can be as simple as a `load` into a processor register that has been hardwired to zero. Slightly more sophisticated implementations provide hints to the memory system as to how the prefetched block will be used. Such information may be useful in multiprocessors where data can be prefetched in different sharing states, for example.

Although particular implementations will vary, all `fetch` instructions share some common characteristics. `Fetch`es are non-blocking memory operations and therefore require a lockup-free cache [15] that allows prefetches to bypass other outstanding memory operations in the cache. Prefetches are typically implemented in such a way that `fetch` instructions cannot cause exceptions. Exceptions are suppressed for prefetches to insure that they remain an optional optimization feature and do effect program correctness or initiate large and potentially unnecessary overhead such as page faults or other memory exceptions.

The hardware required to implement software-initiated prefetching is modest compared to other prefetching strategies. Most of the complexity of this approach lies in the judicious placement of `fetch` instructions within the target application. The task of choosing where in the program to place a `fetch` instruction relative to the matching `load` or `store` instruction is known as *prefetch scheduling.*

In practice, it is not possible to precisely predict when to schedule a prefetch so that data arrives in the cache at the moment it will be requested by the processor, as was the case in Figure 2b. The execution time between the prefetch and the matching memory reference may vary as will memory latencies. These uncertainties are not predictable at compile time and therefore require careful consideration when statically scheduling prefetch instructions.

Fetch instructions may be added by the programmer or by the compiler during an optimization pass. Unlike many optimizations which occur too frequently in a program or are too tedious to implement by hand, prefetch scheduling can often be done effectively by the programmer. Studies have indicated that adding just a few prefetch directives to a program can substantially improve performance [18]. However, if programming effort is to be kept at a minimum, or if the program contains many prefetching opportunities, compiler support may be required.

Whether hand-coded or automated by a compiler, prefetching is most often used within loops responsible for large array calculations. Such loops provide excellent prefetching opportunities because they are common in scientific codes, exhibit poor cache utilization and often have predictable array referencing patterns. By establishing these patterns at compile-time, `fetch` instructions can be placed inside loop bodies so that data for a future loop iteration can be prefetched during the current iteration.

As an example of how loop-based prefetching may be used, consider the code segment shown in Figure 4a. This loop calculates the inner product of two vectors, `a` and `b`, in a manner similar to the innermost loop of a matrix multiplication calculation. If we assume a four-word cache block, this code segment will cause a cache miss every fourth iteration. We can attempt to avoid these cache misses by adding the prefetch directives shown in Figure 4b. Note that this figure is a source code representation of the assembly code that would be generated by the compiler.

```
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)

Figure 4. Inner product calculation using a) no prefetching, b) simple prefetching, c) prefetching with loop unrolling and d) software piplining.

This simple approach to prefetching suffers from several problems. First, we need not prefetch every iteration of this loop since each fetch actually brings four words (one cache block) into the cache. Although the extra prefetch operations are not illegal, they are unnecessary and will degrade performance. Assuming `a` and `b` are cache block aligned, prefetching should only be done on every fourth iteration. One solution to this problem is to surround the `fetch` directives with an `if` condition that tests when `i modulo 4 = 0` is true. The introduction of such an explicit *prefetch predicate*, however, would likely offset the benefits of prefetching and should therefore be avoided. A better solution is to unroll the loop by a factor of *r* where *r* is equal to the number of words to be prefetched per cache block. As shown in Figure 4c, unrolling a loop involves replicating the loop body *r* times within the loop body and increasing the loop stride to *r*. Note that

the `fetch` directives are not replicated and the index value used to calculate the prefetch address is changed from `i+1` to `i+`*r*.

The code segment given in Figure 4c removes most cache misses and unnecessary prefetches but further improvements are possible. Note that cache misses will occur during the first iteration of the loop since prefetches are never issued for the initial iteration. Unnecessary prefetches will occur in the last iteration of the unrolled loop where the `fetch` commands attempt to access data past the loop index boundary. Both of the above problems can be remedied by using *software piplining* techniques as shown in Figure 4d. In this figure, we have extracted select code segments out of the loop body and placed them on either side of the original loop. Fetch statements have been prepended to the main loop to prefetch data for the first iteration of the main loop, including `ip`. This segment of code is referred to as the loop *prolog*. An *epilog* is added to the end of the main loop to execute the final inner product computations without initiating any unnecessary prefetch instructions.

The code given in Figure 4d is said to *cover* all loop references because each reference is preceded by a matching prefetch. However, one final refinement may be necessary to make these prefetches effective. The examples in Figure 4 have been written with the implicit assumption that prefetching one iteration ahead of the data's actual use is sufficient to hide the latency of main memory accesses. This may not be the case when loops contain small computational bodies. For such loops, it may be necessary to initiate prefetches δ iterations before the data is referenced where δ is known as the *prefetch distance* [14, 19] and is expressed in units of loop iterations:

$$\delta = \left\lceil \frac{l}{s} \right\rceil$$

Here, *l* is the average cache miss latency, measured in processor cycles, and *s* is the estimated cycle time of the shortest possible execution path through one loop iteration, including the prefetch overhead. By choosing the shortest execution path through one loop iteration and using the ceiling operator, this calculation is designed to err on the conservative side so that prefetched data are likely to be cached before being requested by the processor.

Returning to the main loop in Figure 4d, let us assume an average miss latency of 100 processor cycles and a loop iteration time of 45 cycles so that $\delta = 3$. Figure 5 shows the final version of the inner product loop which has been altered to handle a prefetch distance of three. Note that the prolog has been expanded to include a loop which prefetches several cache blocks for the initial three iterations of the main loop. Also, the main loop has been shortened to stop prefetching three iterations before the end of the computation. No changes are necessary for the epilog which carries out the remaining loop iterations with no prefetching.

The loop transformations outlined above are fairly mechanical and, with some refinements, can be applied recursively to nested loops. Sophisticated compiler algorithms based on this approach have been developed to automatically add prefetching during an optimization pass of a compiler, with varying degrees of success [3] (see Sidebar 3).

To reduce the latency of remote accesses in a distributed-shared multiprocessor, Mowry and Gupta [18] studied the effect of software prefetching data into a *remote access cache* (RAC) located in the network interface of each processing node. In this way, prefetched remote data could be accessed at a speed comparable to that of local memory while the processor cache hierarchy was reserved for demand-fetched data. Although simulations indicated that the use of the RAC produced noticeable speedups, greater improvements were seen when the RAC was removed and data were prefetched directly into the processor cache hierarchy. Despite significantly increasing

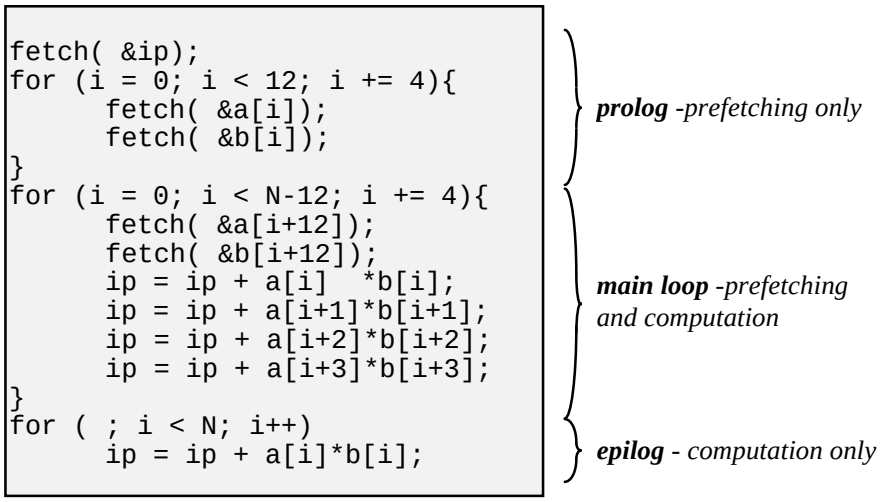


Figure 5. Final inner product loop transformation.

cache contention and reducing overall cache space, this latter approach resulted in higher cache hit rates, which proved to be the more dominate performance factor.

Gornish, et al. [10] noted that the transfer of individual cache blocks across the interconnection network of a multiprocessor yielded low network efficiency and therefore proposed transferring prefetched data in larger units. In this approach, a compiler schedules a single prefetch command before the loop is entered rather than software piplining prefetches within a loop. This command initiates a transfer of large blocks of remote memory used within the loop body to the local memory of the requesting processor. Data are prefetched into local memory rather than the processor cache to prevent excessive cache pollution. Note that this is a binding prefetch since data stored in a processor's local memory are not exposed to any coherency policy which allows updates by other processors to be reflected in the local copy of that data. This binding imposes constraints on the use of prefetched data which, in turn, limits the amount of remote data that can be prefetched. Gornish also found that memory hot-spotting and network contention arising from the use of larger prefetch units further diminished performance to the point that prefetching offered very limited performance improvements.

The above prefetching techniques largely target loops which iterate over large arrays, particularly when a compiler is used. Prefetching is normally restricted to loops containing array accesses whose indices are linear functions of the loop indices because a compiler must be able to reliably predict memory access patterns when scheduling prefetches. Such loops are relatively common in scientific codes but far less so in general applications.

Attempts at establishing similar software prefetching strategies for general applications with irregular data structures have met with limited success [17]. Given the complex control structures typical in such applications, there is often a limited window in which to reliably predict when a particular datum will be accessed. Moreover, once a cache block has been accessed, there is less of a chance that several successive cache blocks will also be requested when data structures such as graphs and linked lists are used. Finally, the comparatively high temporal locality of many general applications often result in high cache utilization thereby diminishing the benefit of prefetching.

Even when restricted to well-conformed looping structures, the use of explicit `fetch` instructions exacts a performance penalty that must be considered when using software-initiated prefetching.

`Fetch` instructions add processor overhead not only because they require extra execution cycles but also because the `fetch` source addresses must be calculated and stored in the processor. Ideally, this prefetch address should be retained so that it need not be recalculated for the matching `load` or `store` instruction. By allocating and retaining register space for the prefetch addresses, however, the compiler will have less register space to allocate to other active variables. The addition of `fetch` instructions is therefore said to increase *register pressure* which, in turn, may result in additional *spill code* to mange variables "spilled" out to main memory due to insufficient register space. The problem is exacerbated when the prefetch distance is greater than one since this implies either maintaining δ address registers to hold multiple prefetch addresses or storing these addresses in memory if the required number of address registers are not available.

Comparing the transformed loop in Figure 5 to the original loop, it can be seen that software prefetching also results in significant code expansion which, in turn, may degrade instruction cache performance. Finally, because software-initiated prefetching is done statically, it is unable to detect when a prefetched block has been prematurely evicted and needs to be re-fetched.

4. Hardware-Initiated Data Prefetching

Several hardware-initiated prefetching schemes have been proposed which add prefetching capabilities to a system without the need for programmer or compiler intervention. No changes to existing executables are necessary so instruction overhead is completely eliminated. Hardware-initiated prefetching also can take advantage of run-time information to potentially make prefetching more effective.

4.1 Sequential prefetching

Most (but not all) prefetching schemes are designed to fetch data from main memory into the processor cache in units of cache blocks. It should be noted, however, that multiple word cache blocks are themselves a form of data prefetching. By grouping consecutive memory words into single units, caches exploit the principle of spatial locality to implicitly prefetch data that is likely to be referenced in the near future.

The degree to which large cache blocks can be effective in prefetching data is limited by the ensuing cache pollution. That is, as the cache block size increases, so does the amount of useful data displaced from the cache to make room for the new block. In shared-memory multiprocessors with private caches, large cache blocks may also cause *false sharing* which occurs when two or more processors wish to access different words within the same cache block and at least one of the accesses is a `store`. Although the accesses are logically applied to separate words, the cache hardware is unable to make this distinction since it operates only on whole cache blocks. The accesses are therefore treated as operations applied to a single object and *cache coherence* traffic is generated to ensure that the changes made to a block by a `store` operation are seen by all processors caching the block. In the case of false sharing, this traffic is unnecessary since only the processor executing the `store` references the word being written. Increasing the cache block size increases the likelihood of two processors sharing data from the same block and hence false sharing is more likely to arise.

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 variations upon the *one block lookahead* (OBL) approach which initiates a prefetch for block *b+1* when block *b* is accessed. This approach differs from simply doubling the block size in that the

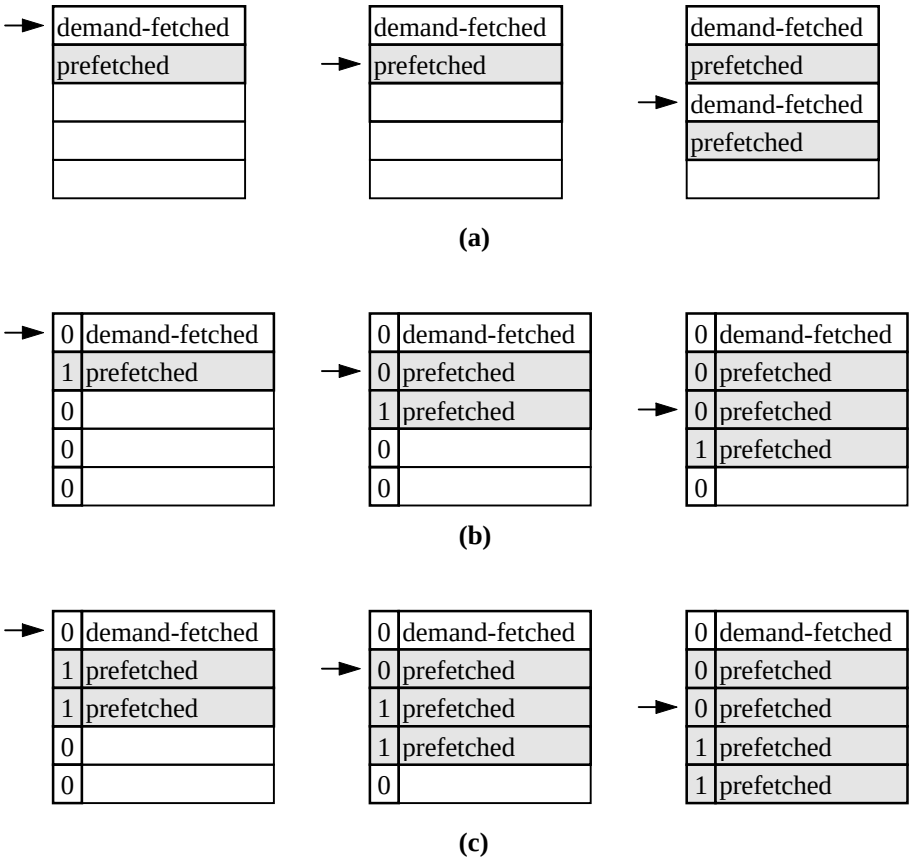


Figure 6. Three forms of sequential prefetching: a) Prefetch on miss, b) tagged prefetch and c) sequential prefetching with *K* = 2.

prefetched blocks are treated separately with regard to the cache replacement and coherency policies. For example, a large block may contain one word which is frequently referenced and several other words which are not in use. Assuming an LRU replacement policy, the entire block will be retained even though only a portion of the block's data is actually in use. If this large block were replaced with two smaller blocks, one of them could be evicted to make room for more active data. Similarly, the use of smaller cache blocks reduces the probability that false sharing will occur.

OBL implementations differ depending on what type of access to block *b* initiates the prefetch of *b+1*. Smith [23] summarizes several of these approaches of which the *prefetch on miss* and *tagged prefetch* algorithms will be discussed here. The *prefetch on miss* algorithm simply 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 memory block. This bit is used to detect when a block is demand-fetched or a prefetched block is referenced for the first time. In either of these cases, the next sequential block is fetched.

Smith found that *tagged prefetching* reduced cache miss ratios in a unified (both instruction and data) cache by between 50 and 90% for a set of trace-driven simulations. *Prefetch on miss* was less than half as effective as *tagged prefetching* in reducing miss ratios. The reason *prefetch on miss* is less effective is illustrated in Figure 6 where the behavior of each algorithm when accessing three contiguous blocks is shown. Here, it can be seen that a strictly sequential access pattern will result in a cache miss for every other cache block when the *prefetch on miss* algorithm is used but

this same access pattern results in only one cache miss when employing a *tagged prefetch* algorithm (see also Sidebar 3).

Note that one shortcoming of the OBL schemes is that the prefetch may not be initiated far enough in advance of the 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. To solve this problem, it is possible to increase the number of blocks prefetched after a demand fetch from one to *K*, where *K* is known as the *degree of prefetching*. Prefetching $K > 1$ subsequent blocks aids the memory system in staying ahead of rapid processor requests for sequential data blocks. As each prefetched block, *b*, is accessed for the first time, the cache is interrogated to check if blocks $b+1, \ldots b+K$ are present in the cache and, if not, the missing blocks are fetched from memory. Note that when $K = 1$ this scheme is identical to tagged prefetching.

Jouppi [13] proposed a similar approach where *K* prefetched blocks are brought into a FIFO *stream buffer* before being brought into the cache. As each buffer entry is referenced, it is brought into the cache while the remaining blocks are moved up in the queue and a new block is prefetched into the tail position. Note that since prefetched data are not placed directly into the cache, this approach avoids cache pollution. However, if a miss occurs in the cache and the desired block is also not found at the head of the stream buffer, the buffer is flushed. This approach therefore requires that prefetched blocks be accessed in a strictly sequential order to take advantage of the stream buffer.

Although increasing the degree of prefetching reduces miss rates in sections of code that show a high degree of spatial locality, additional traffic and cache pollution are generated by sequential prefetching during program phases that show little spatial locality. This overhead tends to make this approach unfeasible for values of *K* larger than one [22].

Dahlgren and Stenström [7] 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 7. 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*

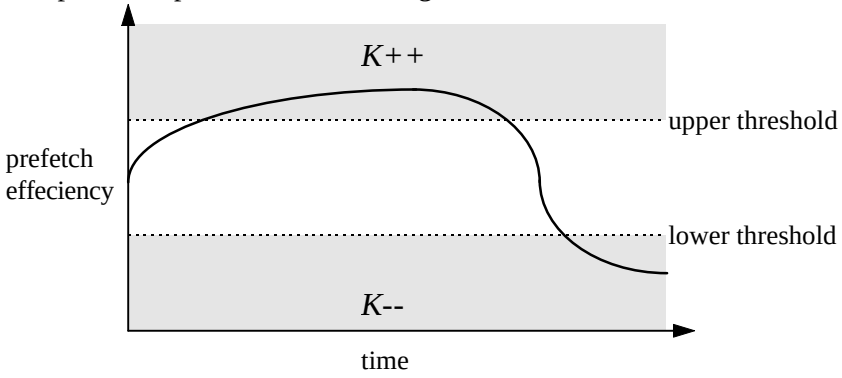


Figure 7. Sequential adaptive prefetching

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. However, simulated run-time comparisons showed only slight differences between the two schemes. The lower miss ratio of adaptive sequential prefetching was found to be partially nullified by the associated overhead of increased memory traffic and contention.

Sequential prefetching requires no changes to existing executables and can be implemented with relatively simple hardware. Of the sequential prefetching policies, tagged prefetching appears to offer both simplicity and performance. Compared to software-initiated prefetching, sequential prefetching will tend to generate more unnecessary prefetches, however. Non-sequential access patterns, such as scalar references or array accesses with large strides, will result in unnecessary prefetch requests because they do not exhibit the spatial locality upon which sequential prefetching is based. To enable prefetching of strided and other irregular data access patterns, several more elaborate hardware prefetching techniques have been proposed. The next section describes these techniques and contrasts them with sequential and software-based schemes.

4.2 Prefetching with arbitrary strides

The simplest way for prefetch hardware to identify a strided array referencing pattern is to explicitly declare when such a pattern occurs within the program and then pass this information on to the hardware. This is possible when programs are explicitly vectorized so that computation is described as a series of vector and matrix operations by the programmer, as is commonly done when programming machines which contain vector processors. Given such a program, array address and stride data can be passed to prefetch logic which may then use this information to calculate prefetch addresses. Fu and Patel [8] established this approach to investigate prefetching opportunities for the Alliant FX/8 multiprocessor. In their scheme, a miss to address a resulting from a vector `load` or `store` instruction with a stride of Δ prompts the prefetch hardware to issue fetches for addresses a, $a + \Delta$, $a + 2\Delta$, ..., $a + K\Delta$, where K is again the degree of prefetching. Unlike similar sequential prefetching techniques, Fu and Patel's approach was found to be effective for values of K up to 32 due to the more informed prefetches that vector stride information affords.

When such high-level information cannot be supplied by the programmer, prefetch opportunities can be detected by hardware which monitors the processor's instruction stream or addressing patterns. Lee, et al. [16] examined the possibility of looking ahead in the instruction stream to find memory references for which prefetches might be dispatched. This approach requires that instructions be brought into a buffer and decoded early so that the operand addresses may be calculated for data prefetching. Instruction lookahead is allowed to pass beyond branches using a branch prediction scheme so that some prefetches will be issued speculatively. This speculation results in some unnecessary prefetches if the branch prediction is incorrect in which case the buffer holding the speculatively loaded instructions must be flushed. Although this scheme allows for prefetching of arbitrary data access patterns, the prefetch window is limited by the depth of the decoded instruction buffer.

Several techniques have been proposed which employ special logic to monitor the processor's address referencing pattern to detect constant stride array references originating from looping structures [2,9]. This is accomplished by comparing successive addresses used by `load` or `store` instructions. The scheme proposed by Chen and Baer [4] is perhaps the most aggressive

thus far. To illustrate its design, assume a memory instruction, m_i, references addresses a_1, a_2 and a_3 during three successive loop iterations. A prefetch for m_i will be initiated if

$$(a_2 - a_1) = \Delta \neq 0$$

where Δ is now assumed to be the stride of a series of array accesses. The first prefetch address will then be $A_3 = a_2 + \Delta$ where A_3 is the predicted value of the observed address, a_3. Prefetching continues in this way until the equality $A_n = a_n$ no longer holds true.

Note that this approach requires the previous address used by a memory instruction to be stored along with the last detected stride, if any. Recording the reference histories of every memory instruction in the program is clearly impossible. Instead, a separate cache called the *reference prediction table* (RPT) holds this information for only the most recently used memory instructions. The organization of the RPT is given in Figure 8. Table entries contain the address of the memory instruction, the previous address accessed by this instruction, a stride value for those entries which have established a stride and a state field which records the entry's current state. The full state diagram for RPT entries is given in Figure 9.

The table is indexed by the CPU's program counter (PC). When memory instruction m_i is executed for the first time, an entry for it is made in the RPT with the state set to *initial* signifying that no prefetching is yet initiated for this instruction. If m_i is executed again before its RPT entry has been evicted, a stride value is calculated by subtracting the previous address stored in the RPT from the current effective address. To illustrate the functionality of the RPT, consider the matrix multiply code and associated RPT entries given in Figure 10.

In this example, only the `load` instructions for arrays `a`, `b` and `c` are considered and it is assumed that the arrays begin at addresses 10000, 20000 and 30000, respectively. For simplicity, one word cache blocks are also assumed. After the first iteration of the innermost loop, the state of the

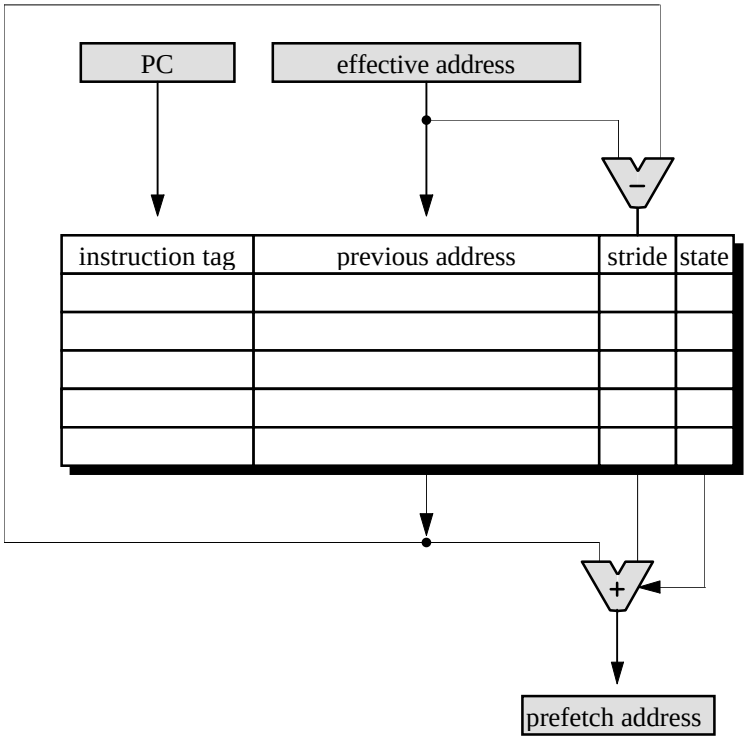


Figure 8. The organization of the reference prediction table.

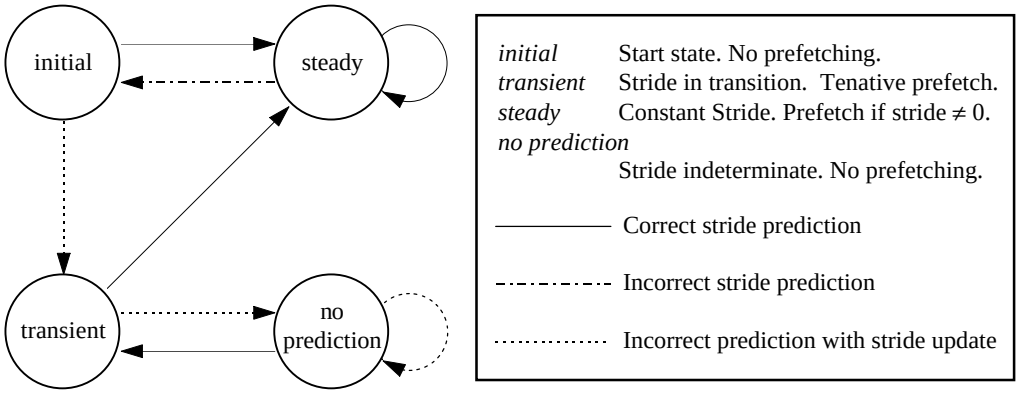


Figure 9. State transition graph for reference prediction table entries.

RPT is as given in Figure 10b where instruction addresses are represented by their pseudo-code mnemonics. Since the RPT does not yet contain entries for these instructions, the stride fields are initialized to zero and each entry is placed in an *initial* state. All three references result in a cache miss.

After the second iteration, strides are computed as shown in Figure 10c. The entries for the array references to b and c are placed in a *transient* state because the newly computed strides do not match the previous stride. This state indicates that an instruction's referencing pattern may be in transition and a tentative prefetch is issued for the block at address *effective address* + *stride* if it is not already cached. The RPT entry for the reference to array a is placed in a *steady* state because the previous and current strides match. Since this entry's stride is zero, no prefetching will be issued for this instruction. Although the reference to array a hits in the cache due a demand fetch in the previous iteration, the references to arrays b and c once again result in a cache miss.

During the third iteration, the entries of array references b and c move to the *steady* state when the tentative strides computed in the second iteration are confirmed. The prefetches issued during the second iteration result in cache hits for the b and c references, provided that a prefetch distance of one is sufficient.

From the above discussion, it can be seen that the RPT improves upon sequential policies by correctly handling strided array references. However, as described above, the RPT still limits the prefetch distance to one loop iteration. To remedy this shortcoming, a *distance* field may be added to the RPT which specifies the prefetch distance explicitly. Prefetch addresses would then be calculated as

$$\textit{effective address} + (\textit{stride} \times \textit{distance})$$

The addition of the *distance* field requires some method of establishing its value for a given RPT entry. To calculate an appropriate value, Chen and Baer decouple the maintenance of the RPT from its use as a prefetch engine. The RPT entries are maintained under the direction of the PC as described above but prefetches are initiated separately by a pseudo program counter, called the *lookahead program counter* (LA-PC) which is allowed to precede the PC. The difference between the PC and LA-PC is then the prefetch distance, δ. Several implementation issues arise with the addition of the lookahead program counter and the interested reader is referred to [2] for a complete description.

In [5], Chen and Baer compared RPT prefetching to Mowry's software-initiated approach and found that neither method showed consistently better performance on a simulated shared memory

multiprocessor. Instead, it was found that performance depended on the individual program characteristics of the four benchmark programs upon which the study was based. Software-initiated prefetching was found to be more effective with certain irregular access patterns for which an indirect reference is used to calculate a prefetch address. In such a situation, the RPT may not be able to establish an access pattern for an instruction which uses an indirect address since the instruction may generate effective addresses which are not separated by a constant stride. Also, the RPT is less efficient at the beginning and end of a loop. Prefetches are issued by the RPT only after an access pattern has been established. This means that no prefetches will be issued for array data for at least the first two iterations. Chen and Baer also noted that it may take several iterations for the RPT to achieve a prefetch distance which completely masks memory latency. Finally, the RPT will always prefetch past array bounds because an incorrect prediction is necessary to stop subsequent prefetching. However, during loop steady state, the RPT was able to dynamically adjust its prefetch distance to achieve a better overlap with memory latency than the software-initiated scheme for some array access patterns. Also, software-initiated prefetching incurred instruction overhead resulting from prefetch address calculation, fetch instruction execution and spill code.

Dahlgren and Stenström [6] compared tagged and RPT prefetching in the context of a distributed shared memory multiprocessor. By examining the simulated run-time behavior of six benchmark programs, it was concluded that RPT prefetching showed limited performance benefits over tagged

```
float a[100][100], b[100][100], c[100][100];

...

for ( i = 0; i < 100; i++)
    for ( j = 0; j < 100; j++)
        for ( k = 0; k < 100; k++)
            a[i][j] += b[i][k] * c[k][j];
```

(a)

Tag	Previous Address	Stride	State
ld b[i][k]	20,000	0	initial
ld c[k][j]	30,000	0	initial
ld a[i][j]	10,000	0	initial

(b)

Tag	Previous Address	Stride	State
ld b[i][k]	20,004	4	transient
ld c[k][j]	30,400	400	transient
ld a[i][j]	10,000	0	steady

(c)

Tag	Previous Address	Stride	State
ld b[i][k]	20,008	4	steady
ld c[k][j]	30,800	400	steady
ld a[i][j]	10,000	0	steady

(d)

Figure 10. The RPT during execution of matrix multiply.

prefetching. which tends to perform as well or better for the most common memory access patterns. For example, Dahlgren showed that most array strides were less than the block size and therefore were captured by the tagged prefetch policy. In addition, the spatial locality of scalar references is captured by the tagged prefetch policy but not by the RPT mechanism which only issues prefetches for previously referenced data. If memory bandwidth is limited, however, it was conjectured that the more conservative RPT prefetching mechanism may be preferable since it tends to produce fewer unnecessary prefetches.

As with software prefetching, the majority of hardware prefetching mechanisms focus on very regular array referencing patterns. There are some notable exceptions, however. Harrison and Mehrotra [12] have proposed extensions to the RPT mechanism which allow for the prefetching of data objects connected via pointers. This approach adds fields to the RPT which enable the detection of indirect reference strides arising from structures such as linked lists and sparse matrices. Zheng and Torrellas [25] suggest tagging memory in such a way that a reference to one element of a data object initiates a prefetch of either other elements within the referenced object or objects pointed to by the referenced object. This approach relies upon some compiler support to initialize the tags in memory, but the actual prefetching is hardware-initiated.

5. Conclusions

Prefetching schemes are diverse. To help categorize a particular approach it is useful to answer three basic questions concerning the prefetching mechanism: 1) *When* are prefetches initiated, 2) *where* are prefetched data placed, and 3) *what* is the unit of prefetch?

- Prefetches can be initiated either by an explicit fetch operation within a program (software-initiated) or by logic that monitors the processor's referencing pattern to infer prefetching opportunities (hardware-initiated). In either case, prefetches must be timely. If a prefetch is issued too early there is a chance that the prefetched data will displace other useful data from the higher levels of the memory hierarchy or be displaced itself before use. If the prefetch is issued too late, it may not arrive before the actual memory reference and thereby introduce processor stall cycles. Prefetching mechanisms also differ in their precision. The software-initiated approach issues prefetches only for data that is likely to be used while hardware schemes tend to fetch more data unnecessarily.
- The decision of where to place prefetched data in the memory hierarchy is a fundamental design decision. Clearly, data must be moved into a higher level of the memory hierarchy to provide a performance benefit. The majority of schemes place prefetched data in some type of cache memory. When prefetched data are placed into named memory locations, such as processor registers, the prefetch is said to be binding and additional constraints must be imposed on the use of the data. Finally, multiprocessor systems can introduce additional levels into the memory hierarchy which must be taken into consideration.
- Data can be prefetched in units of single words, cache blocks or larger blocks of memory. Often, the amount of data fetched is determined by the organization of the underlying cache and memory system. Cache blocks may be the most appropriate size for uniprocessors and SMPs while larger memory blocks may be used to amortize the cost of initiating a data transfer across an interconnection network of a large, distributed memory multiprocessor.

These three questions are not independent of each other. For example, if the prefetch destination is a small processor cache, data must be prefetched in a way that minimizes the possibility of polluting the cache. This means that precise prefetches will need to be scheduled shortly before the

actual use and the prefetch unit must be kept small. If the prefetch destination is large, the timing and size constraints can be relaxed.

Once a prefetch mechanism has been specified, it is natural to wish to compare it with other schemes. A comparative evaluation of the various proposed prefetching techniques is hindered by widely varying architectural assumptions and testing procedures. However, some general observations can be made.

The majority of prefetching schemes and studies concentrate on numerical, array-based applications. These programs tend to generate memory access patterns that, although comparatively predictable, do not yield high cache utilization and therefore benefit more from prefetching than general applications. As a result, automatic techniques which are effective for general programs remain largely unstudied.

To be effective, a prefetch mechanism must perform well for the most common types of memory referencing patterns. Scalar and unit-stride array references typically dominate in most applications and prefetching mechanisms should capture this type of access pattern. Sequential prefetching techniques concentrate exclusively on these access patterns. Although comparatively infrequent, large stride array referencing patterns can result in very poor cache utilization. RPT mechanisms sacrifice some scalar performance in order to cover strided referencing patterns. Software-initiated prefetching handles both types of referencing patterns but introduces instruction overhead.

Finally, memory systems must be designed to match the added demands prefetching imposes. Despite a reduction in overall execution time prefetch mechanisms tend to increase average memory latency. This is a result of effectively increasing the memory reference request rate of the processor thereby introducing congestion within the memory system. This particularly can be a problem in multiprocessor systems where buses and interconnect networks are shared by several processors.

Within these application and system constraints, data prefetching techniques have produced up to nearly two-fold performance improvements on commercial systems. Efforts to improve and extend these known techniques to more diverse architectures and applications is an active and promising area of research. The need for new prefetching techniques is likely to continue to be motivated by ever-increasing memory latencies arising from both the widening gap between microprocessor and memory performance and the use of larger multiprocessor systems.

Sidebar 1. Why do scientific applications exhibit poor cache utilization?

Cache memory designs are based on the principal of *locality* which states that memory references tend to cluster in both time and space. The respective types of locality are known as temporal and spatial locality:

- *Temporal locality* refers to the tendency of programs to reuse recently referenced data. Caches attempt to store only the most recently referenced memory locations in order to take advantage of temporal locality. Cache blocks that have not been recently referenced are therefore displaced from the cache to make room for more active data.
- *Spatial locality* refers to the tendency of most programs to reference memory locations in close proximity to other recently referenced locations. Caches exploit spatial locality by transferring data from memory to the cache in units of cache blocks (also called *lines*) which contain several successive memory words.

Scientific applications often posses little locality in their memory referencing patterns and therefore can defeat attempts to reduce latency through the use of cache memories. The traversal of large data arrays is often at the heart of this problem. Temporal locality is not exhibited by array computations because once an element has been used to compute a result, it is often not referenced again before it is displaced from the cache to make room for additional array elements. Although sequential array accesses patterns exhibit a high degree of spatial locality, many other types of array access patterns do not. For example, in a language which stores matrices in row-major order, a row-wise traversal of a matrix will result in consecutively referenced elements being widely separated in memory. Such *strided* reference patterns result in low spatial locality if the stride is greater than the cache block size. In this case, only one word per cache block is actually used while the remainder of the block remains untouched even though cache space has been allocated for it.

Sidebar 2. Multiprocessor architectures

Processors in a symmetric multiprocessor (SMP) share a common main memory via a shared bus but generally maintain private caches (see Figure S2-1). These caches are kept coherent by hardware which monitors the bus to insure updates to a memory location made by one processor are seen by all processors caching that location. Average memory latencies in SMPs tend to be higher than those of single processor systems due to contention for the shared bus and memory modules. Cache coherency actions also tend to increase memory latency due to the additional bus traffic and cache interference the coherency hardware generates.

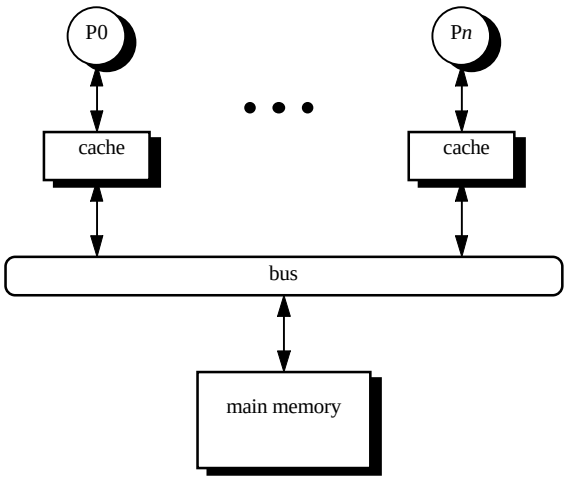


Figure S2-1. A symmetric multiprocessor with *n* processors and private caches.

Distributed memory multiprocessor architectures vary in how memory is distributed and accessed across the system. The nodes of a distributed-shared multiprocessor (DSM) each contain a portion of the total system memory but are able to reference any memory location within the system through an interconnection network (Figure S2-2). Private processor caches are typically kept coherent by passing data and coherency commands between caches. Another possible distributed memory configuration keeps node memory private and attaches global shared memory modules to the interconnection network. The distinction between local, private memory and global, shared memory in such systems complicates cache coherency enough that only local memory is typically cached within a node.

Regardless of the particular distributed memory architecture used, remote memory latencies will be much longer than those of local memory due to the more complex interface to the interconnection network and delays in the network itself. When data are requested or delivered across the interconnection network, network packets must be constructed at the sender's network interface and then processed again at the receiver. Packets being routed across the network may also contend for a particular node or switch thereby introducing further delays.

Because multiprocessors often experience significant memory latencies, prefetching has the potential to provide significant speedups for such systems by overlapping lengthy memory operations with computation. However, care must be taken to prevent the prefetch requests from actually exacerbating the problem. If network or bus bandwidth is limited, unnecessary prefetches or memory hot-spots resulting from several prefetches directed at the same memory location can degrade performance.

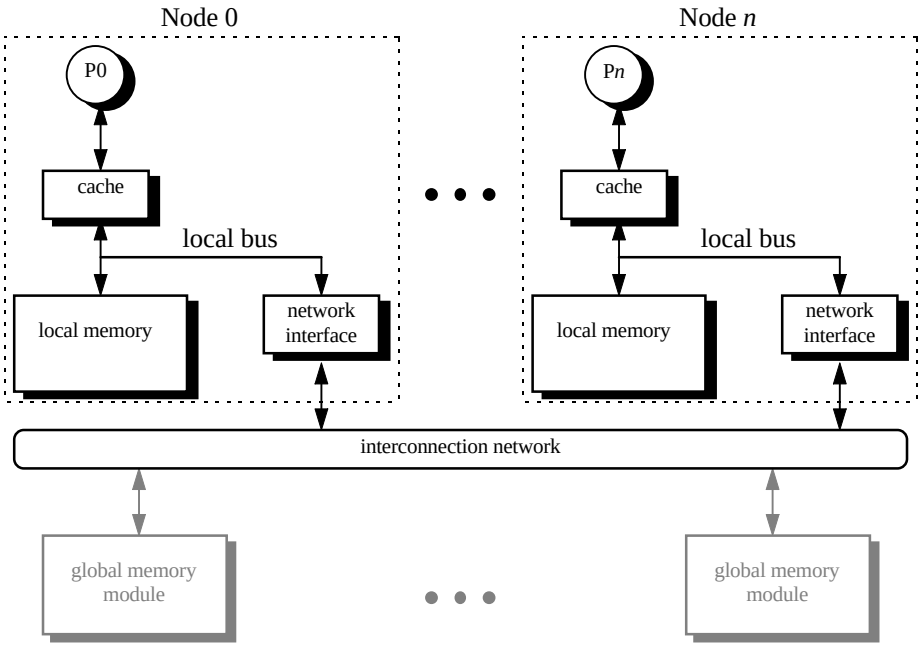


Figure S2-2. Two possible distributed memory multiprocessor configurations with *n* processing nodes. Distributed-shared systems physically distribute memory across processing nodes but make no logical distinction between local and global addresses. Other distributed memory systems place shared memory modules across the interconnection network and keep local memory private.

Sidebar 3. Case study: Prefetching in the HP PA-RISC family.

Steven VanderWiel, Wei Hsu (Hewlett-Packard Corporation) and David J. Lilja

Two recent superscalar implementations of Hewlett-Packard's PA-RISC architecture serve as examples of how hardware and software data prefetching are supported in contemporary microprocessors. The PA7200 [2] implements a tagged, hardware-initiated mechanism while the more recent PA8000 [3] employs explicit prefetch instructions inserted by a production-level compiler.

The PA7200 implements the tagged prefetch scheme using either a *directed* or an *undirected* mode. In the undirected mode, the next sequential line (i.e. OBL) is prefetched. In the directed mode, the prefetch direction (forward or backward) and distance can be determined by the pre/post-increment amount encoded in the `load` or `store` instructions. That is, when the contents of an address register are auto-incremented, the cache block associated with a new address is prefetched.

Whether in directed or undirected mode, the PA7200 prefetches data into an on-chip *assist cache*. This small, fully associative cache combines with a larger, direct-mapped, off-chip cache to form an L1 data cache. Figure S3-1 shows the organization of this level of the cache hierarchy. Lines requested from memory as a result of either a cache miss or a prefetch are initially moved into the assist cache. Lines are moved out of the assist cache in a FIFO order and can be conditionally moved into the off-chip cache if a temporal locality hint is specified by the corresponding memory instruction. The 72000 allows up to four data prefetch requests to be outstanding at one time.

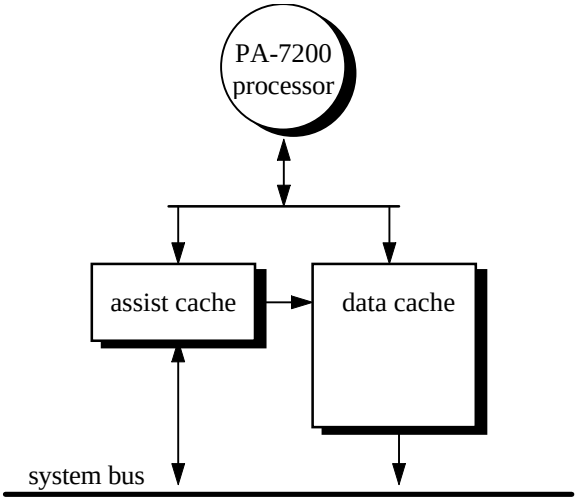


Figure S3-1. The organization of the PA7200 level-1 cache hierarchy.

The PA8000 is the first implementation of the PA 2.0 architecture [4]. This architecture supports a set of explicit data prefetch instructions which are inserted by the HP-PA production compiler. Prefetches are generated and scheduled by this compiler in a manner similar to the scheme proposed by Mowry, et. al. [5] with some differences in miss coverage analysis and prefetch distance calculations.

Prefetched data are placed in the PA8000's large, direct-mapped, off-chip cache which allows up to 10 outstanding cache misses. With a relatively large reorder buffer, out-of-order execution and a non-blocking cache, the PA8000 allows memory operations to be overlapped with other

computations. Software-initiated prefetches are used to exploit this overlap to mask the relatively high miss penalty resulting from the processor's high clock rate and wide issue processing power.

Figure S3-2 shows the speedup achieved with data prefetching on both the PA7200 and PA8000. In this figure, speedups are calculated by dividing the run time of each SpecFP95 benchmark without prefetching by the run time of the same benchmark with prefetching turned on. The test system for the PA7200 was configured with a 256KB data cache and clocked at 120MHz. The PA8000 test system contained a 1MB data cache and ran at 180MHz. Both systems used the same high performance multiprocessor Runway bus [1].

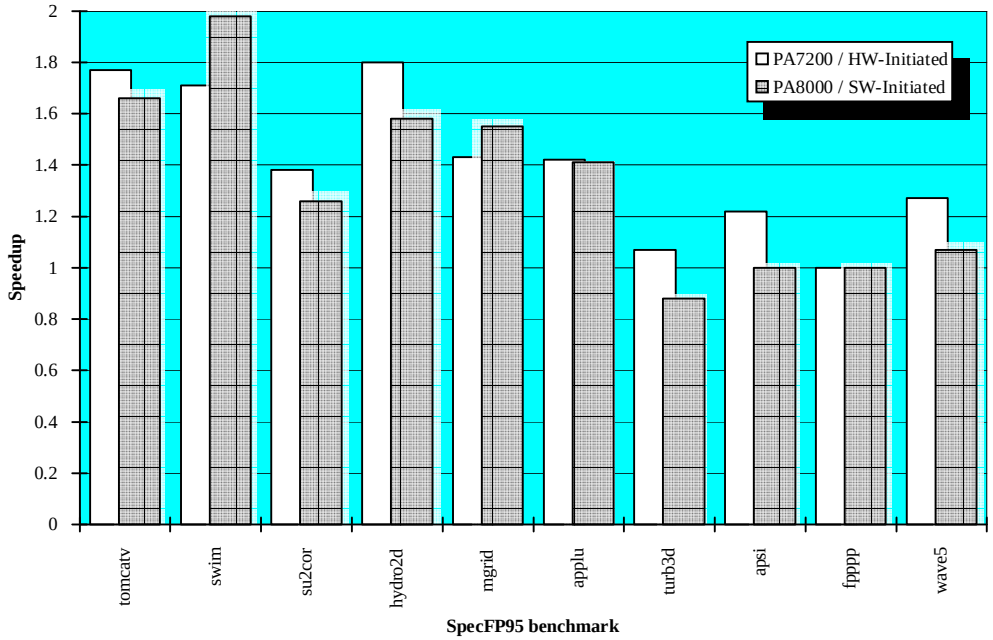


Figure S3-2. SpecFP95 prefetch speedups for the PA7200 and PA8000.

Overall, 16 of the 20 trials produced speedups greater than one with three trials offering no speedup and one trial (turb3d on the PA8000) showing a slowdown (i.e. a speedup of less than one). The application of data prefetching resulted in a average speedup of 1.41 for the PA7200 and 1.35 for the PA8000. Both the 7200 and 8000 achieve high speedups for applications which typically exhibit poor cache utilization such as tomcatv, swim, su2cor, hydro2d, mgrid and applu. However, for programs with intrinsically better cache utilization, the lower overhead of hardware prefetching results in slightly better speedups for the PA7200 than the current implementation of software prefetching for the PA8000. The added overhead of software prefetching in the 8000 actually degraded performance for the turb3d benchmark program. However, since software prefetching can be controlled by users on a per program basis, the lost performance can be regained simply by turning prefetching off for a particular program.

References for Sidebar 3

1. Bryg, W.R., K.K. Chan and N.S. Fiduccia, "A High-Performance, Low-Cost Multiprocessor Bus for Workstations and Midrange Servers," *Hewlett-Packard Journal*, Vol. 47, No. 1, February 1996, p. 18-24.

2. Chan, K.K., et al., “Design of the HP PA 7200 CPU,” *Hewlett-Packard Journal*, Vol. 47, No. 1, February 1996, p. 25-33.

3. Hunt, D, “Advanced Performance Features of the 64-bit PA 8000,” *Digest of Papers Compcon ‘95*, San Francisco, CA, March 1995, p. 123-128.

4. Kane, G., *PA-RISC 2.0 Architecture*, Prentice-Hall, Englewood Cliffs, NJ, 1996.

5. Mowry, T.C., Lam, S. and Gupta, A., “Design and Evaluation of a Compiler Algorithm for Prefetching,” Proc. Fifth International Conf. on Architectural Support for Programming Languages and Operating Systems, Boston, MA, Sept. 1992, p. 62-73.

6. References

1. Anacker, W. and C. P. Wang, "Performance Evaluation of Computing Systems with Memory Hierarchies," *IEEE Transactions on Computers*, Vol. 16, No. 6, December 1967, p. 764-773.

2. Baer, J.-L. and T.-F. Chen, "An Effective On-chip Preloading Scheme to Reduce Data Access Penalty," *Proc. Supercomputing '91*, Albuquerque, NM, Nov. 1991, p. 176-186.

3. Bernstein, D., C. Doron and A. Freund, "Compiler Techniques for Data Prefetching on the PowerPC," *Proc. International Conf. on Parallel Architectures and Compilation Techniques*, June 1995, p. 19-16.

4. Chen T-F. and J-L. Baer, "Effective Hardware-Based Data Prefetching for High Performance Processors," *IEEE Transactions on Computers*, Vol.44, No.5, May 1995, p. 609-623.

5. Chen, Tien-Fu and J. L. Baer, "A Performance Study of Software and Hardware Data Prefetching Schemes," *Proc. of the 21st Annual International Symposium on Computer Architecture*, Chicago, IL, April 1994, p. 223-232.

6. Dahlgren, F. 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.

7. Dahlgren, F., M. Dubois and P. Stenstrom, "Fixed and Adaptive Sequential Prefetching in Shared-memory Multiprocessors," *Proc. of the 1993 International Conference on Parallel Processing*, St. Charles, IL, August 1993, p. I-56-63.

8. Fu, J.W.C. and J.H. Patel, "Data Prefetching in Multiprocessor Vector Cache Memories," *Proc. of the 18th International Symposium on Computer Architecture*, Toronto, Ont., Canada, May 1991, p. 54-63.

9. Fu, J.W.C., J.H. Patel and B.L. Janssens, "Stride Directed Prefetching in Scalar Processors," *Proc. 25th Annual International Symposium on Microarchitecture*, Portland, OR, Dec. 1992, p. 102-110.

10. Gornish, E.H., E.D. Granston and A.V. Veidenbaum, "Compiler-directed Data Prefetching in Multiprocessors with Memory Hierarchies," *Proc. 1990 International Conference on Supercomputing*, Amsterdam, Netherlands, June 1990, p. 354-68.

11. Gupta, A., Hennessy, J., Gharachorloo, K., Mowry, T. and Weber, W.- D., "Comparative Evaluation of Latency Reducing and Tolerating Techniques," *Proc. of the 18th International Symposium on Computer Architecture*, Toronto, Ont., Canada, May 1991, p. 254-263.

12. Harrison, L. and S. Mehrotra, "A Data Prefetch Mechanism for Accelerating General Computation," Technical Report 1351, CSRD, University of Illinois at Urbana-Champaign, Dept. of Computer Science, Urbana, IL, May 1994.

13. Jouppi, N.P., "Improving Direct-mapped Cache Performance by the Addition of a Small Fully-associative Cache and Prefetch Buffers," *Proc. 17th Annual International Symposium on Computer Architecture*, Seattle, WA, May 1990, p. 364-373.

14. Klaiber, A.C. and Levy, H.M., "An Architecture for Software-Controlled Data Prefetching," *Proc. 18th International Symposium on Computer Architecture*, Toronto, Ont., Canada, May 1991, p. 43-53.

15. Kroft, D., “Lockup-free Instruction Fetch/prefetch Cache Organization,” *Proc. of the 8th Annual International Symposium on Computer Architecture*, Minneapolis, MN, May 1981, p. 81-85.

16. Lee, R.L., P.-C. Yew and D.H. Lawrie, “Data Prefetching in Shared Memory Multiprocessors,” *Proc. of the 1987 International Conference on Parallel Processing*, University Park, PA, USA, Aug. 1987, p. 28-31

17. Lipasti, M. H., W. J. Schmidt, S. R. Kunkel and R. R. Roediger, “SPAID: Software Prefetching in Pointer and Call-Intensive Environments,” *Proc. 28th Annual International Symposium on Microarchitecture*, Ann Arbor, MI, November 1995, p. 231-236.

18. Mowry, T. and A. Gupta, “Tolerating Latency through Software-controlled Prefetching in Shared-memory Multiprocessors,” *Journal of Parallel and Distributed Computing*, Vol.12, No.2, June 1991, p. 87-106.

19. Mowry, T.C., Lam, S. and Gupta, A., “Design and Evaluation of a Compiler Algorithm for Prefetching,” *Proc. Fifth International Conf. on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, Sept. 1992, p. 62-73.

20. Patterson, R.H 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

21. Porterfield, A.K., *Software Methods for Improvement of Cache Performance on Supercomputer Applications.* Ph.D. Thesis, Rice University, May 1989.

22. Przybylski, S., “The Performance Impact of Block Sizes and Fetch Strategies,” *Proc. of the17th Annual International Symposium on Computer Architecture*, Seattle, WA, May 1990, p. 160-169.

23. Smith, A.J., “Cache Memories,” *Computing Surveys*, Vol.14, No.3, Sept. 1982, p. 473-530.

24. Young, H.C. and E.J. Shekita, “An intelligent I-cache prefetch mechanism,” *Proc. IEEE International Conference on Computer Design ICCD'93*, Cambridge, MA, October 1993, p. 44-49.

25. Zhang, Z. and J. Torrellas, “Speeding up Irregular Applications in Shared-Memory Multiprocessors: Memory Binding and Group Prefetching,” *Proc. 22th Annual International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, June 1995, p. 188-199.