Data Prefetch Mechanisms

STEVEN P. VANDERWIEL

IBM Server Group

AND

DAVID J. LILJA

University of Minnesota

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 access. 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 many 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 initiate 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 such as cache pollution and increased memory bandwidth requirements 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 that provides optimal performance. The following survey examines several alternative approaches, and discusses the design tradeoffs involved when implementing a data prefetch strategy.

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles—*Cache memories*; B.3 [**Hardware**]: Memory Structures

General Terms: Design, Performance

Additional Key Words and Phrases: Memory latency, prefetching

This work was supported in part by National Science Foundation grants MIP-9610379, CDA-9502979, CDA-9414015, the Minnesota Supercomputing Institute, and the University of Minnesota-IBM Shared University Research Project. Steve VanderWiel was partially supported by an IBM Graduate Research Fellowship during the preparation of this work.

Authors' addresses: S. P. VanderWiel, System Architecture, Performance & Design, IBM Server Group, 3605 Highway 52, North Rochester, MN 55901; email: svw@us.ibm.com; D. J. Lilja, Dept. of Electrical & Computer Engineering, University of Minnesota, 200 Union St. SE, Minneapolis, MN 55455; email: lilja@ece.umn.edu.

CONTENTS

1. Introduction
2. Background
3. Software Data Prefetching
4. Hardware Data Prefetching
   4.1 Sequential Prefetching
   4.2 Prefetching with Arbitrary Strides
5. Integrating Hardware and Software Prefetching
6. Prefetching in Multiprocessors
7. Conclusions

1. INTRODUCTION

By any metric, microprocessor performance has increased at a dramatic rate over the past decade. This trend has been sustained by continued architectural innovations and advances in microprocessor fabrication technology. In contrast, main memory dynamic RAM (DRAM) performance has increased at a much more leisurely rate, as shown in Figure 1.

Chief among latency reducing techniques is the use of cache memory hierarchies [Smith 1982]. The static RAM (SRAM) memories used in caches have managed to keep pace with processor memory request rates but continue to be too expensive for a main store technology. Although the use of large cache hierarchies has proven to be effective in reducing the average memory access penalty for programs that show a high degree of locality in their addressing patterns, it is still not uncommon for scientific and other data-intensive programs to spend more than half their run times stalled on memory requests [Mowry et al. 1992]. The large, dense matrix operations that form the basis of many such applications typically exhibit little data reuse and thus may defeat caching strategies.

The poor cache utilization of these 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 2(a) 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 a simple, in-order execution unit, 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.

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 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

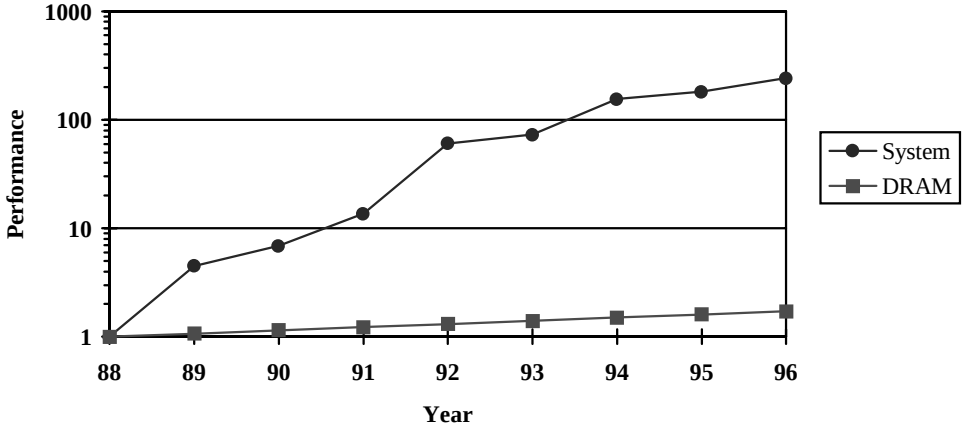

Figure 1. System and DRAM performance since 1988. System performance is measured by SPECfp92 and DRAM performance by row access times. All values are normalized to their 1988 equivalents (Source: Internet SPECtable, ftp://ftp.cs.toronto.edu/pub/jdd/spectable).

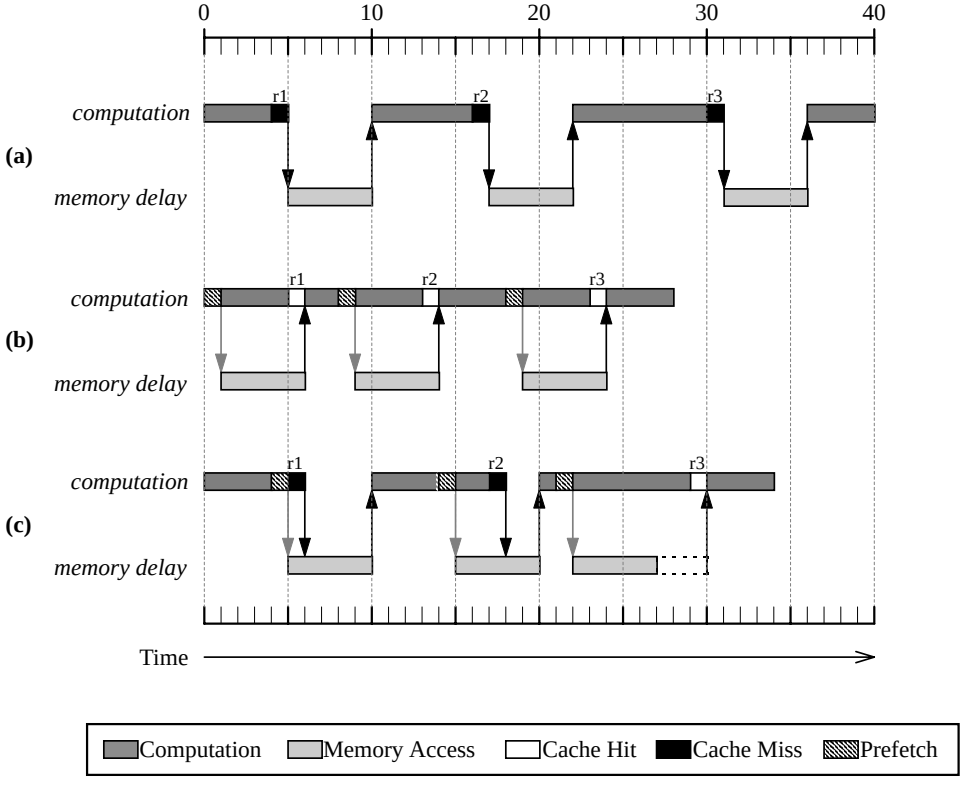

Figure 2. Execution diagram assuming (a) no prefetching, (b) perfect prefetching, and (c) degraded prefetching.

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

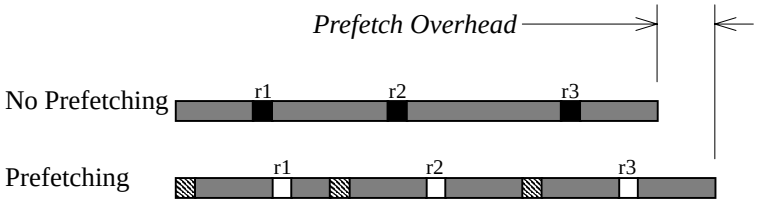

Figure 3. Software prefetching overhead.

without forcing 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 referenced word is not forwarded to the processor after it has been cached. Figure 2(b) shows how prefetching can be used to improve the execution time of the demand fetch case given in Figure 2(a). 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 2(c). 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. When this occurs, the prefetch is said to be *useless* because no performance benefit is derived from fetching the block early.

A prematurely prefetched block may also displace data in the cache that is currently in use by the processor, resulting in what is known as *cache pollution* [Casmira and Kaeli 1995]. 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 2(a) the three memory requests occur within the first 31 time units of program startup, whereas in Figure 2(b), 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 [Burger 1997]. This can be particularly true for multiprocessors where bus utilization is typically higher than single processor systems.

It is also interesting to note that software 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 that do not require the use of explicit fetch instructions. These techniques employ

special hardware that monitors the processor in an attempt to infer prefetching opportunities. Although hardware prefetching incurs no instruction overhead, it often generates more *unnecessary prefetches* than software prefetching. Unnecessary prefetches are more common in hardware schemes because they 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 unnecessary prefetches do not affect correct program behavior, they can result in cache pollution, and will consume memory 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 that 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; no single strategy that provides optimal performance has yet been proposed. 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-1960's. Early studies of cache design [Anacker and Wang 1967] 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 prefetching of separate cache blocks was later implemented in the IBM 370/168 and Amdahl 470V [Smith 1978]. Software techniques are more recent. Smith first alluded to this idea in his survey of cache memories [Smith 1982], but at that time doubted its usefulness. Later, Porterfield [1989] proposed the idea of a "cache load instruction" at approximately the same time that Motorola introduced the "touch load' instruction in the 88110, and Intel proposed the use of "dummy read" operations in the i486 [Fu et al. 1989]. Today, nearly all microprocessor ISAs contain explicit instructions designed to bring data into the processor cache hierarchy.

It should be noted that 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, see Young and Shekita [1993], or Patterson and Gibson [1994]. For brevity, only techniques that apply to data objects residing in memory will be considered here.

Nonblocking load instructions share many similarities with data prefetching. Like prefetches, these instructions are issued in advance of the data's 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. Nonblocking 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 nonblocking loads are not discussed further here, other forms of binding prefetches are 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. 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.

Due to this emphasis on prefetching in multiprocessor systems, many of the prefetching mechanisms discussed below have been studied either mostly 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 in such systems.

3. SOFTWARE DATA PREFETCHING

Most contemporary microprocessors support some form of `fetch` instruction that can be used to implement prefetching [Bernstein et al. 1995; Santhanam et al. 1997; Yeager 1996]. The Implementaation 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. `Fetches` are nonblocking memory operations and therefore require a lockup-free cache [Kroft 1981] 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 that does not affect program correctness or initiate large and potentially unnecessary overhead, such as page faults or other memory exceptions.

The hardware required to implement software 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 2(b). 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 scheduling prefetch instructions in a program.

`Fetch` instructions may be added by the programmer or by the compiler during an optimization pass. Unlike many optimizations that 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 indicate that adding just a few prefetch directives to a program can substantially improve performance [Mowry and Gupta 1991]. 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 4(a). 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 4(b). Note that this figure is a source code representation of the assembly code that would be generated by the compiler.

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 be done only 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 overhead of such an explicit *prefetch predicate*, however, would likely offset the benefits of prefetching, and therefore should 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 4(c), unrolling a loop involves replicating the loop body r times 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 4(c) 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 pipelining* techniques as shown in Figure 4(d). In this figure,

```
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 pipelining.

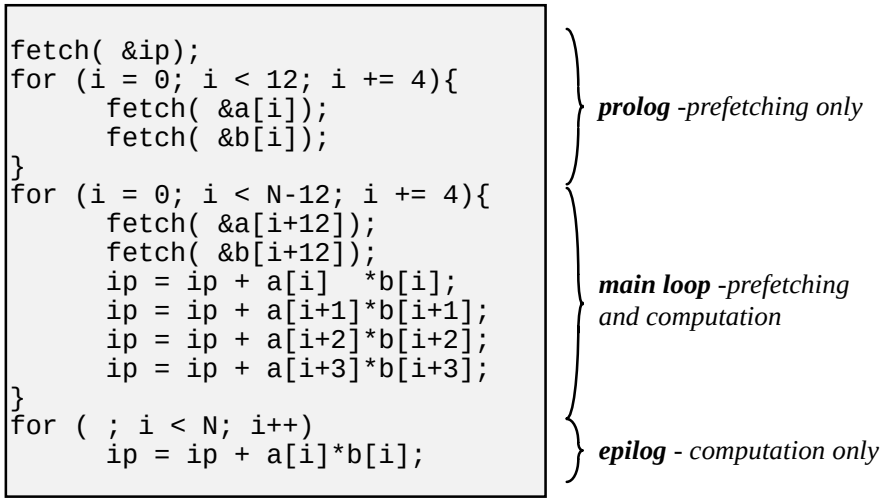

Figure 5. Final inner product loop transformation.

we 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 4 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 were 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. Although early studies [Callahan et al. 1991] were based on this assumption, Klaiber and Levy [1991] recognized that this was not a sufficiently general solution. When loops contain small computational bodies, it may be necessary to initiate prefetches

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

where l is the average memory 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, and thus increase the likelihood that prefetched data will be cached before it is requested by the processor.

Returning to the main loop in Figure 4(d), let us assume an average miss latency of 100 processor cycles and a loop iteration time of 45 cycles so that the final inner product loop transformation is as shown in Figure 5.

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, with varying degrees of success, to automatically add `fetch` instructions during an optimization pass of a compiler [Mowry et al. 1992]. Bernstein et al. [1995] measured the run-times of 12 scientific benchmarks both with and without the use of prefetching on a PowerPC 601-based system. Prefetching typically improved

run-times by less than 12%, although one benchmark ran 22% faster and three others actually ran slightly slower due to prefetch instruction overhead. Santhanam et al. [Kroft 1981] found that six of the ten SPECfp95 benchmark programs ran between 26% and 98% faster on a PA8000-based system when prefetching was enabled. Three of the four remaining SPECfp95 programs showed less than a 7% improvement in run-time and one program was slowed down by 12%.

Because a compiler must be able to reliably predict memory access patterns, prefetching is normally restricted to loops containing array accesses whose indices are linear functions of the loop indices. Such loops are relatively common in scientific codes, but far less so in general applications. Attempts at establishing similar software prefetching strategies for these applications are hampered by their irregular referencing patterns [Chen et al. 1991; Lipasti et al. 1995; Luk and Mowry 1996]. Given the complex control structures typical of general 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 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 manage variables "spilled" out to memory due to insufficient register space. The problem is exacerbated when 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 registers is 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 prefetching is done statically, it is unable to detect when a prefetched block has been prematurely evicted and needs to be refetched.

4. HARDWARE DATA PREFETCHING

Several hardware prefetching schemes have been proposed that 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 prefetching can also 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 effects. That is, as the cache block size increases, so does the amount of potentially 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* [Lilja 1993], 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 because it operates on whole cache blocks only. Hence the accesses are 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 because 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 differs from simply doubling the block size, in that the prefetched blocks are treated separately with regard to cache replacement and coherence policies. For example, a large block may contain one word that is frequently referenced and several other words that 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 is 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 [1982] 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 reduces 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.

The HP PA7200 [Chan et al. 1996] serves as an example of a contemporary microprocessor that uses OBL prefetch hardware. The PA7200 implements a tagged prefetch scheme using either a *directed* or an *undirected* mode. In the undirected mode, the next sequential line is prefetched. In the directed mode, the prefetch direction (forward or backward)

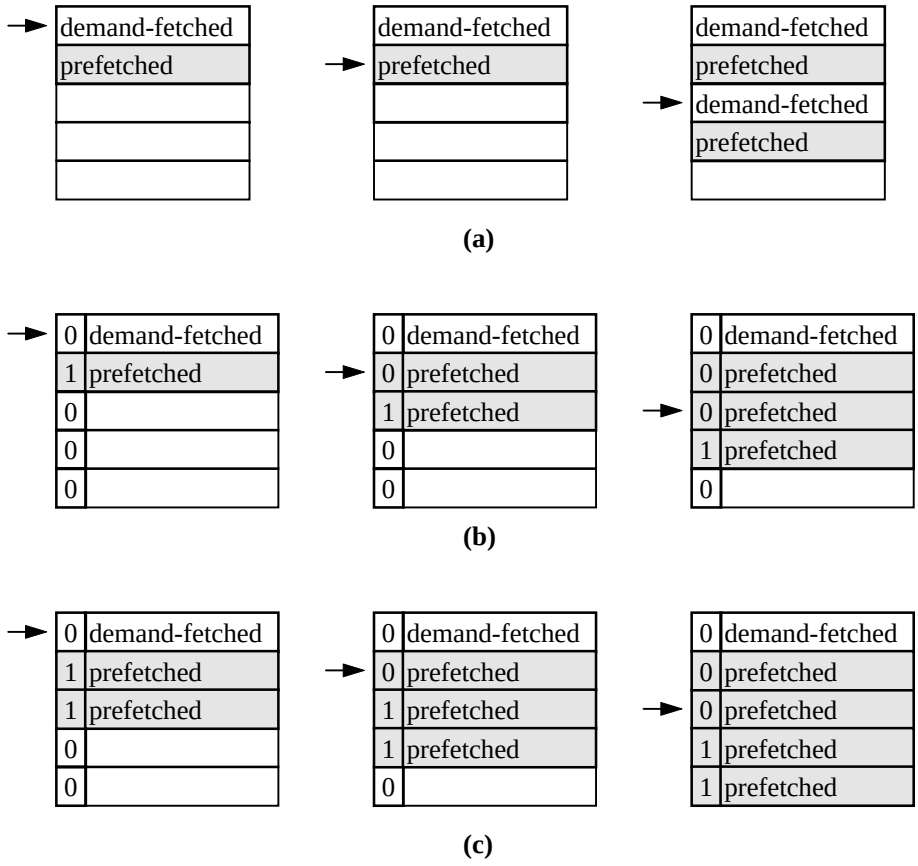

Figure 6. Three forms of sequential prefetching: (a) Prefetch on miss, (b) tagged prefetch, and (c) sequential prefetching with $K = 2$.

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. Compared to a base case with no prefetching, the PA7200 achieved run-time improvements in the range of 0% to 80% for 10 SPECfp95 benchmark programs [VanderWiel et al. 1997]. Although performance was found to be application-dependent, all but two of the programs ran more than 20% faster when prefetching was enabled.

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 lead time between the use of block b and the request for block $b + 1$. 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.

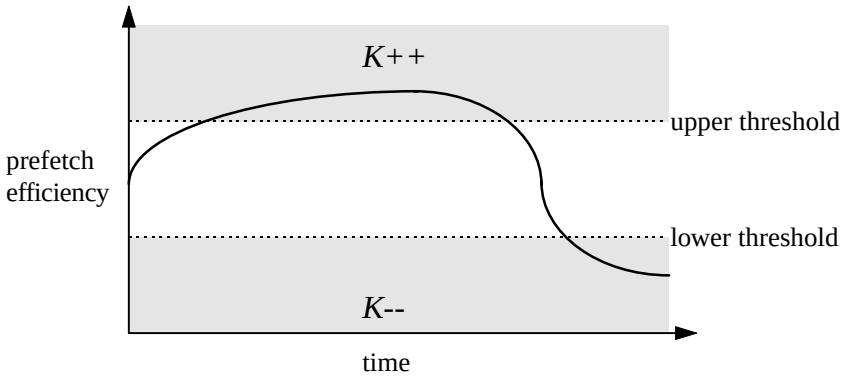

Figure 7. Sequential adaptive prefetching.

Note that when $K = 1$, this scheme is identical to tagged OBL prefetching.

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. Przybylski [1990] found that this overhead tends to make sequential prefetching unfeasible for values of K larger than one.

Dahlgren et al. [1993] 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 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 show only slight differences between the two schemes. The lower miss ratio of adaptive sequential prefetching was partially nullified by the associated overhead of increased memory traffic and contention.

Jouppi [1990] proposed an 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 scheme avoids any 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. Therefore, prefetched blocks must be accessed in the order they are brought into the buffer for

stream buffers to provide a performance benefit.

Palacharla and Kessler [1994] studied stream buffers as a cost-effective alternative to large secondary caches. When a primary cache miss occurs, one of several stream buffers is allocated to service the new reference stream. Stream buffers are allocated in LRU order and a newly allocated buffer immediately fetches the next K blocks following the missed block into the buffer. Palacharla and Kessler found that eight stream buffers and $K = 2$ provided adequate performance in their simulation study. With these parameters, stream buffer hit rates (the percentage of primary cache misses that are satisfied by the stream buffers) typically fell between 50% and 90%.

However, memory bandwidth requirements were found to increase sharply as a result of the large number of unnecessary prefetches generated by the stream buffers. To help mitigate this effect, a small history buffer is used to record the most recent primary cache misses. When this history buffer indicates that misses have occurred for both block b and block $b + 1$, a stream is allocated and blocks $b + 2$, …, $b + K + 1$ are prefetched into the buffer. Using this more selective stream allocation policy, bandwidth requirements were reduced at the expense of some slightly reduced stream buffer hit rates. The stream buffers described by Palacharla and Kessler were found to provide an economical alternative to large secondary caches, and were eventually incorporated into the Cray T3E multiprocessor [Oberlin et al. 1996].

In general, sequential prefetching techniques require no changes to existing executables and can be implemented with relatively simple hardware. However, compared to software prefetching, sequential hardware prefetching performs poorly when nonsequential memory access patterns are encountered. Scalar references or array accesses with large strides can result in unnecessary prefetches because these types of access patterns 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.

4.2 Prefetching with Arbitrary Strides

Several techniques have been proposed that employ special logic to monitor the processor's address referencing pattern to detect constant stride array references originating from looping structures [Baer and Chen 1991; Fu et al. 1992; Sklenar 1992]. This is accomplished by comparing successive addresses used by `load` or `store` instructions. Chen and Baer's [1995] technique is perhaps the most aggressive proposed 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. Prefetching 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 $A_n \neq a_n$.

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

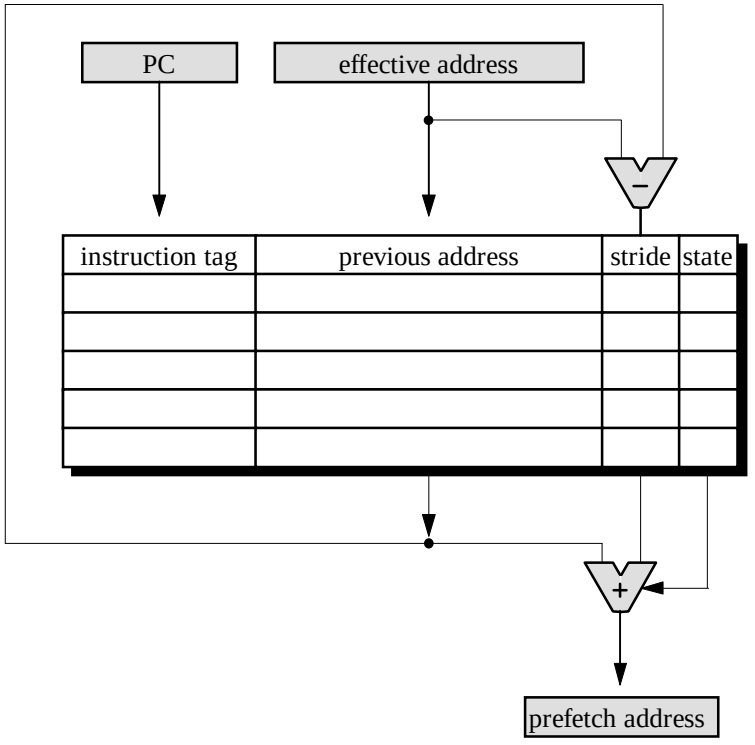

Figure 8. State transition graph for reference prediction table entries.

that have established a stride, and a state field that records the entry's current state. The state diagram for RPT entries is given in Figure 9.

The RPT 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 RPT is as given in Figure 10(b) where instruction addresses are represented by their pseudocode 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 10(c). 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

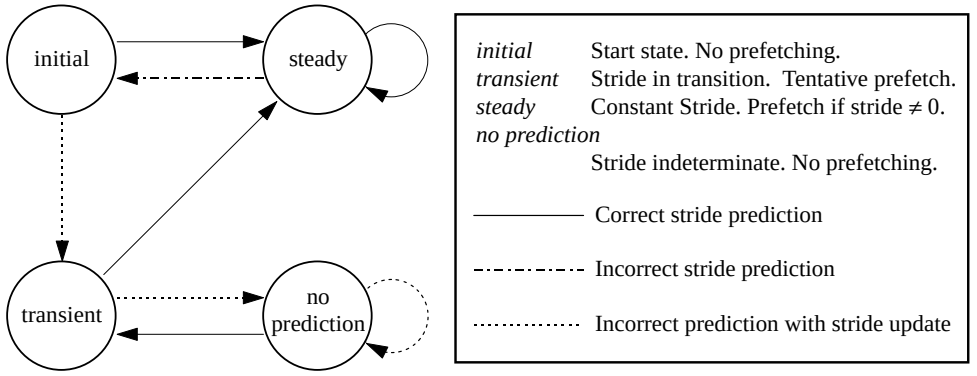

Figure 9. The RPT during execution of matrix multiply.

stride is zero, no prefetching will be issued for this instruction.

During the third iteration, the entries for array references b and c move to the *steady* state when the tentative strides computed in the previous 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; the interested reader is referred to Baer and Chen [1991].

Chen and Baer [1994] compared RPT prefetching to Mowry's software prefetching scheme [Mowry et al. 1992], and found that neither method showed consistently better performance on a simulated shared memory multiprocessor. Instead, it was found that performance depends on the individual program characteristics of the four benchmark programs upon which the study is based. Software prefetching was found to be more effective with certain irregular access patterns for which an indirect reference is used to calculate a prefetch address. The RPT may not be able to establish an access pattern for an instruction that uses an indirect address because the instruction may generate effective addresses that 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 note that it may take several iterations for the RPT to achieve a prefetch distance that completely masks memory latency when the LA-PC was used.

```
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. An access to the next field of a list element can prompt prefetching of the subsequent list element.

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 is able to dynamically adjust its prefetch distance to achieve a better overlap with memory latency than the software scheme for some array access patterns. Also, software prefetching incurs instruction overhead resulting from prefetch address calculation, fetch instruction execution, and spill code.

Dahlgren and Stenstrom [1995] 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 prefetching, which tends to perform as well or better for the most common memory access patterns. Dahlgren showed that most array strides are less than the block size, and therefore covered by the tagged prefetch policy. In addition, it was found that some scalar references show a limited amount of spatial locality that could be captured by the tagged prefetch policy but not by the RPT mechanism. If memory bandwidth

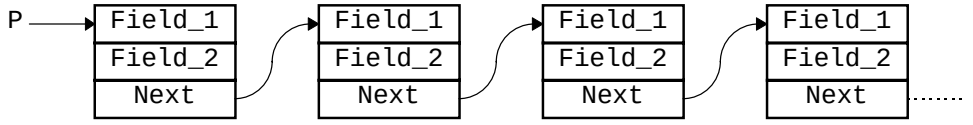

Figure 11. Block prefetching for a vector-matrix product calculation.

is limited, however, it is conjectured that the more conservative RPT prefetching mechanism may be preferable, since it tends to produce fewer useless prefetches.

As with software prefetching, the majority of hardware prefetching mechanisms focus on very regular array referencing patterns. There are some notable exceptions, however. For example, Harrison and Mehrotra [1994] propose extensions to the RPT mechanism that allow for the prefetching of data objects connected via pointers. This approach adds fields to the RPT that enable the detection of indirect reference strides arising from structures such as linked lists and sparse matrices.

Another hardware mechanism designed to prefetch data structures connected via pointers was described by Roth et al. [1998]. Like the RPT and its derivatives, this scheme uses a hardware table to log the most recently executed load instructions. However, this table is used to detect dependencies between load instructions rather than establishing reference patterns for single instructions. More specifically, this table records the data values loaded by the most recently executed load instructions and detects instances of these values being used as base addresses for subsequent loads. Such address dependencies are common in linked list processing, i.e., where the next pointer of one list element is used as the base address for the fields of the subsequent list element, as shown in Figure 11. Once the hardware table establishes these dependencies, prefetches are triggered by the execution of those load instructions that produce base addresses. For example, once the address of `p->next` is known, `p->next->field_1` and `p->next->field_2` can be prefetched. A prefetch of `p->next->next` can be used to initiate further prefetching. However, it was found that such aggressive prefetching is generally not useful because the relatively long processing time required for each pointer-connected element is sufficient to hide the latency of the prefetches.

Alexander and Kedem [1996] proposed a prefetch mechanism based on the observation that a cache miss to a given address tends to be followed by a miss that belongs to a relatively predictable subset of addresses. To take advantage of this property, a hardware table is used to associate the current cache miss address with a set of likely successor cache miss addresses. That is, when a cache miss occurs, the address of the missed block is used to index the table to find the most likely block addresses for the next cache miss, based on previous observations. Prefetches are then issued for these blocks. Rather than prefetching data into the processor cache, however, data blocks are prefetched into SRAM buffers integrated onto the DRAM memory chips. When a cache miss has been correctly predicted by the prefetch mechanism, the corresponding data can be read directly from the SRAM buffers, thereby avoiding the relatively time-consuming DRAM access. Data are not brought into the processor cache hierarchy because this scheme prefetches large amounts of data in order to exploit the high on-chip bandwidth that exists between the integrated DRAM arrays and SRAM buffers. Alexander and Kedem found that prefetch units in the range of 8 to 64 cache blocks gave the best performance for a benchmark suite of scientific and engineering applications. Prefetching

up to four such units on each cache miss yields an average prediction accuracy of 88%.

Using a similar approach, Joseph and Grunwald [1997] studied the use of a Markov predictor to drive a hardware data prefetch mechanism. This mechanism attempts to predict when a previous pattern of misses has begun to repeat itself by dynamically recording sequences of cache miss references in a hardware table similar to that used in Alexander and Kedem [1996]. When the current cache miss address is found in the table, prefetches for likely subsequent misses are issued to a prefetch request queue. To prevent cache pollution and wasted memory bandwidth, prefetch requests may be displaced from this queue by requests that belong to reference sequences with a higher probability of occurring in the near future. This probability is determined by the strength of the Markov chain associated with a given reference.

Rather than triggering data prefetches on memory instructions, some researchers [Chang et al. 1994; Lui and Kaeli 1996] proposed triggering on branch instructions stored in a branch target buffer (BTB) [Smith 1981]. When a branch instruction is executed, its entry in the BTB is inspected to predict not only the branch target address but also the data that will be consumed after the predicted branch is taken. One advantage of this approach is that several memory operations can be associated with a single branch target, thereby reducing the amount of tag space required to support prefetching. The entries in the BTB that control prefetching are similar to the contents of an RPT entry, with fields for the previous data address, a stride, and the state of the entry. These fields are updated in a fashion similar to the RPT if the branch prediction is correct and the corresponding memory instructions are executed. Studies of such branch-predicted prefetching indicate that no more than three prefetch entries per BTB entry are required to yield close to the maximum benefit of this approach. Simulations of a processor using branch-predicted prefetching with three prefetch entries per BTB entry found that miss rates improved by between 11.8% and 63.9% for six SPECint92 benchmarks.

5. INTEGRATING HARDWARE AND SOFTWARE PREFETCHING

Software prefetching relies exclusively on compile-time analysis to schedule fetch instructions within the user program. In contrast, the hardware techniques discussed thus far infer prefetching opportunities at run-time without any compiler or instruction set support. Noting that each of these approaches has its advantages, some researchers have proposed mechanisms that combine elements of both software and hardware prefetching.

Gornish and Veidenbaum [1994] describe a variation on tagged hardware prefetching in which the degree of prefetching (K) for a particular reference stream is calculated at compile time and passed on to the prefetch hardware. To implement this scheme, a *prefetching degree* (PD) field is associated with every cache entry. A special fetch instruction is provided that prefetches the specified block into the cache and then sets the tag bit and the value of the PD field of the cache entry holding the prefetched block. The first K blocks of a sequential reference stream are prefetched using this instruction. When a tagged block, b, is demand fetched, the value in its PD field, K_b, is added to the block address to calculate a prefetch address. The PD field of the newly prefetched block is then set to K_b and the tag bit is set. This insures that the appropriate value of K is propagated through the reference stream. Prefetching for nonsequential reference patterns is handled by ordinary fetch instructions.

Zhang and Torrellas [1995] suggest an integrated technique that enables prefetching for irregular data structures. This is accomplished by tagging

memory locations 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. Both array elements and data structures connected via pointers can therefore be prefetched. This approach relies on the compiler to initialize the tags in memory, but the actual prefetching is handled by hardware within the memory system.

The use of a programmable *prefetch engine* was proposed by Chen [1995] as an extension to the reference prediction table described in Section 4.2. Chen's prefetch engine differs from the RPT in that the tag, address, and stride information are supplied by the program rather than being dynamically established in hardware. Entries are inserted into the engine by the program before entering looping structures that can benefit from prefetching. Once programmed, the prefetch engine functions much like the RPT, with prefetches being initiated when the processor's program counter matches one of the tag fields in the prefetch engine.

VanderWiel and Lilja [1999] propose a prefetch engine that is external to the processor. The engine is a simple processor that executes its own program to prefetch data for the CPU. Through a shared second-level cache, a producer-consumer relationship is established in which the engine prefetches new data blocks into the cache, but only after previously prefetched data is accessed by the processor. The processor also partially directs the actions of the prefetch engine by writing control information to memory-mapped registers within the prefetch engine's support logic.

These integrated techniques are designed to take advantage of compile-time program information without introducing as much instruction overhead as pure software prefetching. Much of the speculation by pure hardware prefetching is also eliminated, resulting in fewer unnecessary prefetches. Although no commercial systems support this model of prefetching yet, the simulation studies used to evaluate the above techniques indicate that performance can be enhanced over pure software or hardware prefetch mechanisms.

6. PREFETCHING IN MULTIPROCESSORS

In addition to the prefetch mechanisms above, several multiprocessor-specific prefetching techniques have been proposed. Prefetching in these systems differs from uniprocessors for at least three reasons. First, multiprocessor applications are typically written using different programming paradigms than uniprocessors. These paradigms can provide additional array referencing information that enables more accurate prefetch mechanisms. Second, multiprocessor systems frequently contain additional memory hierarchies that provide different sources and destinations for prefetching. Finally, the performance implications of data prefetching can take on added significance in multiprocessors because these systems tend to have higher memory latencies and more sensitive memory interconnects.

Fu and Patel [1991] examined how data prefetching might improve the performance of vectorized multiprocessor applications. This study assumes vector operations are explicitly specified by the programmer and supported by the instruction set. Because the vectorized programs describe computations in terms of a series of vector and matrix operations, no compiler analysis or stride detection hardware is required to establish memory access patterns. Instead, the stride information encoded in vector references is made available to the processor caches and associated prefetch hardware.

Two prefetching policies were studied. The first is a variation upon the prefetch-on-miss policy, in which K consecutive blocks following a cache miss are fetched into the processor cache. This implementation of prefetch-on-miss differs from that presented earlier

in that prefetches are issued only for scalars and vector references with a stride less than or equal to the cache block size. The second prefetch policy, referred to as *vector prefetching* here, is similar to the first policy, with the exception that prefetches for vector references with large strides are also issued. If the vector reference for block b misses in the cache, then blocks b, $b + stride$, $b + (2 \times stride)$, . . . , $b + (K \times stride)$ are fetched.

Fu and Patel found both prefetch policies improve performance over the no prefetch on an Alliant FX/8 simulator. Speedups were more pronounced when smaller cache blocks were assumed, since small block sizes limit the amount of spatial locality a nonprefetching cache can capture, while prefetching caches can offset this disadvantage by simply prefetching more blocks. In contrast to other studies, Fu and Patel found both sequential prefetching policies were effective for values of K up to 32. This is in apparent conflict with earlier studies, which found sequential prefetching to degrade performance for $K > 1$. Much of this discrepancy may be explained by noting how vector instructions are exploited by the prefetching scheme used by Fu and Patel. In the case of prefetch-on-miss, prefetching is suppressed when a large stride is specified by the instruction. This avoids useless prefetches, which degraded the performance of the original policy. Although *vector prefetching* does issue prefetches for large stride referencing patterns, it is a more precise mechanism than other sequential schemes because it is able to take advantage of stride information provided by the program.

Comparing the two schemes, it was found that applications with large strides benefit the most from vector prefetching, as expected. For programs in which scalar and unit-stride references dominate, the prefetch-on-miss policy tends to perform slightly better. For these programs, the lower miss ratios resulting from the vector prefetching policy are offset by the corresponding increase in bus traffic.

Gornish et al. [1990] examined prefetching in a distributed memory multiprocessor where global and local memory are connected through a multistage interconnection network. Data are prefetched from global to local memory in large, asynchronous block transfers to achieve higher network bandwidth than would be possible with word-at-a-time transfers. Since large amounts of data are prefetched, the data are placed in local memory rather than the processor cache to avoid excessive cache pollution. Some form of software-controlled caching is assumed to be responsible for translating global array addresses to local addresses after the data been placed in local memory.

As with software prefetching in single-processor systems, loop transformations are performed by the compiler to insert prefetch operations into the user code. However, rather than inserting `fetch` instructions for individual words within the loop body, entire blocks of memory are prefetched before the loop is entered. Figure 12 shows how this block prefetching may be used with a vector-matrix product calculation. In Figure 12(b), the iterations of the original loop (Figure 12(a)) are partitioned among `NPROC` processors of the multiprocessor system so that each processor iterates over $1/NPROC$th of `a` and `c`. Also note that the array `c` is prefetched a row at a time. Although it is possible to *pull out* the prefetch for `c` so that the entire array is fetched into local memory before entering the outermost loop, it is assumed here that `c` is very large and a prefetch of the entire array would occupy more local memory than is available.

The block fetches given in Figure 12(b) add processor overhead to the original computation in a manner similar to the software prefetching scheme described earlier. Although the block-oriented prefetch operations require size and stride information, significantly less

```
for( i=0; i < N; i++){
      for( j=0; j < N; j++)
            a[i] = a[i] + b[j]*c[i][j];
}
```

(a)

```
nrows = N/NPROC;

fetch( b[0:N-1]);
for( i=0; i < nrows; i++){
      fetch( c[i][0:nrows-1]);
      for( j=0; j < N; j++)
            a[i] = a[i] + b[j]*c[i][j];
}
```

(b)

Figure 12. Block prefetching for a vector-matrix product calculation.

overhead is incurred than with the word-oriented scheme, since fewer prefetch operations are needed. Assuming equal problem sizes and ignoring prefetches for a, the loop given in Figure 12 generates $N + 1$ block prefetches, as compared to the $1/2(N + N^2)$ prefetches that would result from applying a word-oriented prefetching scheme.

Although a single bulk data transfer is more efficient than dividing the transfer into several smaller messages, the former approach tends to increase network congestion when several such messages are being transferred at once. Combined with the increased request rate that prefetching induces, this network contention can lead to significantly higher average memory latencies. For a set of six numerical benchmark programs, Gornish notes that prefetching increases average memory latency by a factor between 5.3 and 12.7 over the no prefetch case.

An implication of prefetching into the local memory rather than the cache is that the array a in Figure 12 cannot be prefetched. In general, this scheme requires that all data must be read-only between prefetch and use because no coherence mechanism is provided that allows writes by one processor to be seen by the other processors. Data transfers are also restricted by control dependencies within the loop bodies. If an array reference is predicated by a conditional statement, no prefetching is initiated for the array. This is done for two reasons. First, the conditional may only test true for a subset of the array references, and initiating a prefetch of the entire array would result in the unnecessary transfer of a potentially large amount of data. Second, the conditional may guard against referencing nonexistent data, and initiating a prefetch for such data could result in unpredictable behavior.

Honoring the above data and control dependencies limits the amount of data that can be prefetched. On average, 42% of loop memory references for the six benchmark programs used by Gornish could not be prefetched due to these constraints. Together with the increased average memory latencies, the suppression of these prefetches limited the speedup due to prefetching to less than 1.1 for five of the six benchmark programs.

Mowry and Gupta [1991] studied the effectiveness of software prefetching for the DASH DSM multiprocessor architecture. In this study, two alternative designs are considered. The first design places prefetched data in a *remote access*

cache (RAC), which lies between the interconnection network and the processor cache hierarchy of each node in the system. The second design alternative simply prefetched data from remote memory directly into the primary processor cache. In both cases, the unit of transfer is a cache block.

The use of a separate prefetch cache such as the RAC was motivated by a desire to reduce contention for the primary data cache. By separating prefetched data from demand-fetched data, a prefetch cache avoids polluting the processor cache and provides more overall cache space. This approach also avoids processor stalls that can result from waiting for prefetched data to be placed in the cache. However, in the case of a remote access cache, only remote memory operations benefit from prefetching, since the RAC is placed on the system bus and access times are approximately equal to those of main memory.

Simulation runs of three scientific benchmarks found that prefetching directly into the primary cache offered the most benefit with an average speedup of 1.94 compared to an average of 1.70 when the RAC was used. Despite significantly increasing cache contention and reducing overall cache space, prefetching into the primary cache resulted in higher cache hit rates, which proved to be the dominant performance factor. As with software prefetching in single processor systems, the benefit of prefetching was application-specific. Speedups for two array-based programs achieved speedups over the non-prefetch case of 2.53 and 1.99 while the third, less regular, program showed a speedup of 1.30.

7. 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 prefetched?

When Prefetches can be initiated either by an explicit fetch operation within a program, by logic that monitors the processor's referencing pattern to infer prefetching opportunities, or by a combination of these approaches. However they are initiated, prefetches must be issued in a timely manner. 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. Software prefetching issues fetches only for data that is likely to be used, while hardware schemes tend to prefetch data in a more speculative manner.

Where 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. Other schemes place prefetched data in dedicated buffers to protect the data from premature cache evictions and prevent cache pollution. When prefetched data are placed into named locations, such as processor registers or memory, 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, which must be taken into consideration, into the memory hierarchy.

What Data can be prefetched in units of single words, cache blocks, contiguous blocks of memory, or program data objects. The amount of data fetched is also 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 compare it to other schemes. Unfortunately, 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 thus benefit more from prefetching than general applications do. As a result, automatic techniques that are effective for general programs have received comparatively little attention.

Within the context of array-based referencing patterns, prefetch mechanisms provide varying degrees of coverage, depending on the flexibility of the mechanism. Because unit- or small-stride array referencing patterns are the most common and easily detected, all prefetching schemes capture this type of access pattern. Sequential prefetching techniques concentrate exclusively on such patterns. Although less frequent, large-stride array referencing patterns can result in very poor cache utilization. The design of the RPT is motivated by the desire to capture all constant-stride referencing patterns, including those with large strides. Array referencing patterns that do not have a constant stride or frequently change strides typically cannot be covered by pure hardware techniques. Software prefetching can provide coverage for any arbitrary referencing pattern if the pattern can be detected at compile-time. The array-based integrated techniques discussed thus far are designed to augment existing hardware techniques with software support, but their coverage is limited by the underlying hardware mechanisms.

Flexibility also tends to lend accuracy to a prefetch scheme. Software and integrated techniques avoid many of the unnecessary prefetches characteristic of pure hardware mechanisms, which are more constrained in the prefetch streams they can produce. These unnecessary prefetches can displace active data within the cache with data that is never used by the processor. In addition to causing cache pollution, unnecessary prefetches needlessly expend memory bandwidth, which may already be limited due to the added pressure prefetching naturally places on the memory system.

Prefetch mechanisms also introduce some degree of hardware overhead. All techniques rely on cache hardware that supports a prefetch operation. Most sequential schemes introduce additional cache logic, with the exception of stream buffers, which require hardware that is external to the cache. Some techniques require that logic be added to the processor. Pure software prefetching requires the

inclusion of a fetch instruction in the processors instruction set, while the RPT and its derivatives add a comparatively large amount of logic overhead into the processor.

Although software prefetching has minimal hardware requirements, this technique introduces a significant amount of instruction overhead into the user program. Integrated schemes attempt to strike a balance between pure hardware and software schemes by reducing instruction overhead while still offering better prefetch coverage than pure hardware techniques.

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 by removing processor stall cycles. This effectively increases the memory reference request rate of the processor which, in turn, can introduce congestion within the memory system. This can be a problem, particularly in multiprocessor systems where buses and interconnect networks are shared by several processors.

Despite the many application and system constraints, data prefetching has demonstrated the ability to reduce overall program execution time both in simulation studies and in real 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 increasing memory access penalties arising from both the widening gap between microprocessor and memory performance and the use of more complex memory hierarchies.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their many useful suggestions.

REFERENCES

ALEXANDER, T. AND KEDEM, G. 1996. Distributed prefetch-buffer/cache design for high performance memory systems. In *Proceedings of 2nd IEEE Symposium on High-Performance Computer Architecture*. IEEE Press, Piscataway, NJ, 254–263.

ANACKER, W. AND WANG, C. P. 1967. Performance evaluation of computing systems with memory hierarchies. *IEEE Trans. Comput. 16*, 6, 764–773.

BAER, J.-L. AND CHEN, T.-F. 1991. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of the 1991 Conference on Supercomputing* (Albuquerque, NM, Nov. 18–22), J. L. Martin, Chair. ACM Press, New York, NY, 176–186.

BERNSTEIN, D., COHEN, D., AND FREUND, A. 1995. Compiler techniques for data prefetching on the PowerPC. In *Proceedings of the IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques* (PACT '95, Limassol, Cyprus, June 27–29), L. Bic, P. Evripidou, W. Böhm, and J.-L. Gaudiot, Chairs. IFIP Working Group on Algol, Manchester, UK, 19–26.

BURGER, D., GOODMAN, J. R., AND KGI, A. 1997. Limited bandwidth to affect processor design. *IEEE Micro 17*, 6, 55–62.

CALLAHAN, D., KENNEDY, K., AND PORTERFIELD, A. 1991. Software prefetching. *SIGARCH Comput. Arch. News 19*, 2 (Apr.), 40–52.

CASMIRA, J. P. AND KAELI, D. R. 1995. Modeling cache pollution. In *Proceedings of the Second IASTED Conference on Modeling and Simulation*. 123–126.

CHAN, K. K. 1996. Design of the HP PA 7200 CPU. *Hewlett-Packard J. 47*, 1, 25–33.

CHANG, P. Y., KAELI, D., AND LIU, Y. 1994. Branch-directed data cache prefetching. In *Proceedings of Second Workshop on Shared-Memory Multiprocessing Systems*.

CHEN, T.-F. 1995. An effective programmable prefetch engine for on-chip caches. In *Proceedings of the 28th Annual International Symposium on Microarchitecture* (Ann Arbor, MI, Nov. 29 - Dec. 1), T. Mudge and K. Ebcioğlu, Chairs. IEEE Computer Society Press, Los Alamitos, CA, 237–242.

CHEN, T.-F. AND BAER, J. L. 1994. A performance study of software and hardware data prefetching schemes. In *Proceedings of 21st International Symposium on Computer Architecture* (Chicago, IL, Apr.). 223–232.

CHEN, T.-F. AND BEAR, J. L. 1995. Effective hardware-based data prefetching for high-performance processors. *IEEE Trans. Comput. 44*, 5 (May), 609–623.

CHEN, W. Y., MAHLKE, S. A., CHANG, P. P., AND HWU, W.-M. W. 1991. Data access microarchitectures for superscalar processors with compiler-assisted data prefetching. In *Proceedings of the 24th Annual International*

Symposium on Microarchitecture (MICRO 24, Albuquerque, NM, Nov. 18–20), Y. K. Malaiya, Chair. ACM Press, New York, NY, 69–73.

DAHLGREN, F. AND STENSTROM, P. 1995. Effectiveness of hardware-based stride and sequential prefetching in shared-memory multiprocessors. In *Proceedings of 1st IEEE Symposium on High-Performance Computer Architecture* (Raleigh, NC, Jan.). IEEE Press, Piscataway, NJ, 68–77.

DAHLGREN, F., DUBOIS, M., AND STENSTROM, P. 1993. Fixed and adaptive sequential prefetching in shared-memory multiprocessors. In *Proceedings of the International Conference on Parallel Processing* (St. Charles, IL). 56–63.

FU, B., SAINI, A., AND GELSINGER, P. P. 1989. Performance and microarchitecture of the i486 processor. In *Proceedings of the IEEE International Conference on Computer Design* (Cambridge, MA). 182–187.

FU, J. W. C. AND PATEL, J. H. 1991. Data prefetching in multiprocessor vector cache memories. In *Proceedings of 18th International Symposium on Computer Architecture* (Toronto, Ont., Canada). 54–63.

FU, J. W. C., PATEL, J. H., AND JANSSENS, B. L. 1992. Stride directed prefetching in scalar processors. In *Proceedings of the 25th Annual International Symposium on Microarchitecture* (MICRO 25, Portland, OR, Dec. 1–4), W.-m. Hwu, Chair. IEEE Computer Society Press, Los Alamitos, CA, 102–110.

GORNISH, E. H. AND VEIDENBAUM, A. V. 1994. An integrated hardware/software scheme for shared-memory multiprocessors. In *Proceedings of International Conference on Parallel Processing* (St. Charles, IL). 281–284.

GORNISH, E. H., GRANSTON, E. D., AND GRANSTON, A. V. 1990. Compiler-directed data prefetching in multiprocessors with memory hierarchies. In *Proceedings of the 1990 ACM International Conference on Supercomputing* (ICS '90, Amsterdam, The Netherlands, June 11–15), A. Sameh and H. van der Vorst, Chairs. ACM Press, New York, NY, 354–368.

HARRISON, L. AND MEHROTRA, S. 1994. A data prefetch mechanism for accelerating general computation. 1351. University of Illinois at Urbana-Champaign, Champaign, IL.

JOSEPH, D. AND GRUNWALD, D. 1997. Prefetching using Markov predictors. In *Proceedings of the 24th International Symposium on Computer Architecture* (ISCA '97, Denver, CO, June 2–4), A. R. Pleszkun and T. Mudge, Chairs. ACM Press, New York, NY, 252–263.

JOUPPI, N. 1990. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th International Symposium on Computer Architecture* (ISCA '90, Seattle, WA, May). IEEE Press, Piscataway, NJ, 364–373.

KLAIBER, A. C. AND LEVY, H. M. 1991. An architecture for software-controlled data prefetching. *SIGARCH Comput. Arch. News 19*, 3 (May), 43–53.

KROFT, D. 1981. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th International Symposium on Computer Architecture* (Minneapolis, MN, June). ACM Press, New York, NY, 81–85.

LILJA, D. J. 1993. Cache coherence in large-scale shared-memory multiprocessors: Issues and comparisons. *ACM Comput. Surv. 25*, 3 (Sept.), 303–338.

LIPASTI, M. H., SCHMIDT, W. J., KUNKEL, S. R., AND ROEDIGER, R. R. 1995. SPAID: Software prefetching in pointer- and call-intensive environments. In *Proceedings of the 28th Annual International Symposium on Microarchitecture* (Ann Arbor, MI, Nov. 29–Dec. 1), T. Mudge and K. Ebcioğlu, Chairs. IEEE Computer Society Press, Los Alamitos, CA, 231–236.

LUI, Y. AND KAELI, D. R. 1996. Branch-directed and stride-based data cache prefetching. In *Proceedings of International Conference on Computer Design* (ICCD'96, Austin, TX). IEEE Computer Society Press, Los Alamitos, CA, 255–230.

LUK, C.-K. AND MOWRY, T. C. 1996. Compiler-based prefetching for recursive data structures. *ACM SIGOPS Oper. Syst. Rev. 30*, 5, 222–233.

MOWRY, T. AND GUPTA, A. 1991. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *J. Parallel Distrib. Comput. 12*, 2 (June), 87–106.

MOWRY, T. C., LAM, M. S., AND GUPTA, A. 1992. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS-V, Boston, MA, Oct. 12–15), S. Eggers, Chair. ACM Press, New York, NY, 62–73.

OBERLIN, S., KESSLER, R., SCOTT, S., AND THORSON, G. 1996. Cray T3E architecture overview. Cray Supercomputers, Chippewa Falls, MN.

PALACHARLA, S. AND KESSLER, R. E. 1994. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of 21st International Symposium on Computer Architecture* (Chicago, IL, Apr.).

PATTERSON, R. H. AND GIBSON, G. A. 1994. Exposing I/O concurrency with informed prefetching. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems* (Austin, TX). 7–16.

PORTERFIELD, A. K. 1989. Software methods for improvement of cache performance on supercomputer applications. Ph.D thesis,. Ph.D. Dissertation. Rice University, Houston, TX.

PRZYBYLSKI, S. 1990. The performance impact of block sizes and fetch strategies. In *Proceedings of the 17th International Symposium on*

Computer Architecture (Seattle, WA). 160–169.

ROTH, A., MOSHOVOS, A., AND SOHI, G. 1998. Dependance based prefetching for linked data structures. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS-VIII, San Jose, CA, Oct. 3–7), D. Bhandarkar and A. Agarwal, Chairs. ACM Press, New York, NY.

SANTHANAM, V., GORNISH, E. H., AND HSU, W.-C. 1997. Data prefetching on the HP PA-8000. In *Proceedings of the 24th International Symposium on Computer Architecture* (ISCA '97, Denver, CO, June 2–4), A. R. Pleszkun and T. Mudge, Chairs. ACM Press, New York, NY, 264–273.

SKLENAR, I. 1992. Prefetch unit for vector operations on scalar computers. In *Proceedings of the 19th International Symposium on Computer Architecture* (Gold Coast, Qld., Australia). 31–37.

SMITH, A. J. 1978. Sequential program prefetching in memory hierarchies. *IEEE Computer 11*, 12, 7–21.

SMITH, A. J. 1982. Cache memories. *ACM Comput. Surv. 14*, 3 (Sept.), 473–530.

SMITH, J. E. 1981. A study of branch prediction techniques. In *Proceedings of the 8th International Symposium on Computer Architecture* (Minneapolis, MN, June). ACM Press, New York, NY, 135–147.

VANDERWIEL, S. P. AND LILJA, D. J. 1999. A compiler-assisted data prefetch controller. In *Proceedings of International Conference on Computer Design* (ICCD '99, Austin TX).

VANDERWIEL, S. P., HSU, W. C., AND LILJA, D. J. 1997. When caches are not enough: Data prefetching techniques. *IEEE Computer 30*, 7, 23–27.

YEAGER, K. C. 1996. The MIPS R10000 superscalar microprocessor. *IEEE Micro 16*, 2 (Apr.), 28–40.

YOUNG, H. C. AND SHEKITA, E. J. 1993. An intelligent I-cache prefetch mechanism. In *Proceedings of the International Conference on Computer Design* (ICCD'93, Cambridge, MA). IEEE Computer Society Press, Los Alamitos, CA, 44–49.

ZHANG, Z. AND TORRELLAS, J. 1995. Speeding up irregular applications in shared-memory multiprocessors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture* (ISCA '95, Santa Margherita Ligure, Italy, June 22–24), D. A. Patterson, Chair. ACM Press, New York, NY.

Received: January 1998; revised: March 1999; accepted: April 1999