When Caches Aren't Enough: Data Prefetching Techniques

With data prefetching, memory systems call data into the cache before the processor needs it, thereby reducing memory-access latency. Using the most suitable techniques is critical to maximizing data prefetching's effectiveness.

Steven P. VanderWiel and David J. Lilja
University of Minnesota

The past decade has seen enormous strides in microprocessor fabrication technology and design methodology. As a result, CPU performance has outpaced that of dynamic RAM, the primary component of main memory. This expanding gap has required developers to use increasingly aggressive techniques to reduce or hide the large latency of main-memory accesses.

The use of cache memory hierarchies has been the chief technique to reduce this latency. The more expensive static RAM (SRAM) memories used in caches have kept pace with processor memory-request rates, and large cache hierarchies have reduced the latency experienced when accessing the most frequently used data. However, it is still not uncommon for scientific programs to spend more than half their runtimes stalled on memory requests.[1] The large, dense matrix operations that frequently form the basis of such applications typically exhibit little data reuse and therefore can defeat caching strategies.

Scientific applications' poor cache utilization is partially a result of the memory fetch policy of most caches. Under this policy, the cache controller fetches data from the main memory only after the processor requests the data and only if it is not subsequently found in the cache. Processor stall cycles are likely to occur because the computation cannot resume until the main memory supplies the requested data.

This policy will always result in a cache miss (a *cold start* or *compulsory miss*) the first time an application tries to access a particular data block because only previously accessed data is in the cache.

Also, if the referenced data is part of a large array operation, the cache will evict less recently used data to make room for new array elements being streamed into the cache. If the same data block is needed later, the processor will have to fetch it again from main memory. This situation is known as a *capacity miss*.

System designers can avoid many of these cache misses by adding a *data prefetch* operation to the cache. Data prefetching anticipates cache misses and fetches data from the memory system before the processor needs the data. The prefetch occurs while processor computation takes place, which gives the memory system time to transfer the desired data to the cache. Ideally, the prefetch would complete just in time for the processor to access the needed data, thereby avoiding stall cycles.

Prefetching can nearly double the performance of some scientific applications running on commercial systems. Comparative performance evaluations[2,3] suggest that no single approach to data prefetching provides better performance in all situations, as shown in the sidebar "Prefetching in the HP PA-RISC Family."

We will review three popular prefetching techniques: software-initiated prefetching, sequential hardware-

It is increasingly common for designers to implement data prefetching by including a fetch instruction in a processor's instruction set.

initiated prefetching, and prefetching via reference prediction tables.

SOFTWARE PREFETCHING

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

Prefetch scheduling

The hardware required to implement software-initiated prefetching is modest compared to other prefetching strategies. Most of the approach's complexity lies in the judicious placement of fetch instructions within the application. Choosing where to place a fetch instruction relative to the corresponding `load` or `store` instruction is known as *prefetch scheduling*. Software prefetching can often take advantage of compile-time information to schedule prefetches more accurately than hardware techniques.

In practice, it is not possible to predict exactly when to schedule a prefetch so that data arrives in the cache exactly when it will be requested by the processor. The execution time between the prefetch and the matching `load` or `store` may vary, as will memory latencies. These uncertainties must be considered when deciding where in the program to place fetch instructions.

If the compiler schedules fetches too late, the data will not be in the cache when the processor needs it. If the compiler schedules the fetches too early, the cache may evict the data before it can be used to make room for data that the controller fetches later. Early prefetches also might displace data in the cache that the processor is using at the time. This situation, in which there is a miss that would not have occurred without prefetching, is called *cache pollution*.

Prefetching for loops

Fetch instructions may be hand-coded by the programmer or added by a compiler during an optimization pass. In either case, prefetching is most often used within loops responsible for large array calculations. Such loops, which are common in scientific code, provide excellent prefetching opportunities because they exhibit poor cache utilization and often have predictable memory-referencing patterns.

Simple prefetching. The code segment in Figure 1a is an example of loop-based prefetching.

This loop sums the elements of array a. If we assume a four-word cache block, this code segment will cause a cache miss every fourth iteration. We can try to avoid these cache misses by using the prefetch directives added in Figure 1b. The prefetch of the array element to be used in the next loop iteration is scheduled just before computation for the current iteration begins.

However, prefetching every iteration of this loop is unnecessary because each fetch actually brings four array elements into the cache. Prefetching should thus be done only every fourth iteration. One solution is to surround the `fetch` directives with an `if` condition that tests when `(i modulo 4) = 0` is true. The computational overhead of introducing such an explicit *prefetch predicate*, however, would probably offset the benefits of prefetching and should thus be avoided.

Unrolling the loop. Unrolling the loop by a factor of r (where r is equal to the number of words to be prefetched per cache block) is more effective than using an explicit prefetch predicate. As Figure 1c shows, unrolling a loop involves replicating the loop body r times and increasing the loop increment stride from one to r. In the process the compiler doesn't replicate fetch directives, but it does change the *array index*, which it uses to calculate the prefetch address, from $i + 1$ to $i + r$.

Nonetheless, cache misses will occur during the loop's first iteration because prefetches are never issued for this iteration. In addition, unnecessary prefetches will occur in the unrolled loop's last iteration, in which the fetch command tries to access data past array boundaries.

Software pipelining. Software pipelining techniques, shown in Figure 1d, can solve these problems. In this figure, we have extracted some code segments from the loop body and placed them on either side of the original loop. We have added a *loop prologue* consisting of fetch statements to the beginning of the main loop to prefetch data for the first iteration. We added an epilogue to the end of the main loop to execute the final computations without initiating unnecessary prefetch instructions.

The code in Figure 1d covers all loop references because a prefetch precedes each reference. However, one final refinement may be necessary to make sure the compiler schedules the prefetches early enough. We wrote the examples in the Figures 1a through 1d assuming that prefetching data one iteration early would provide enough time to bring the requested data from the main memory to the cache. However, 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, the *prefetch distance*,[1] is

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

Here, *l* is the average cache miss latency, measured in processor cycles, and *s* is the estimated number of cycles in the shortest possible execution path through one loop iteration, including any prefetch overhead. (A compiler can automatically determine the shortest path without much trouble.) Using the ceiling operator, which rounds the result to the next highest whole number, and calculating the prefetch distance by using the shortest execution path means that data will be prefetched as far in advance as is generally necessary, which will maximize the likelihood that the prefetched data will be cached before it is requested by the processor.

In Figure 1d, if we assume an average miss latency of 100 processor cycles and a minimum loop iteration time of 45 cycles, δ will be 3. Figure 1e shows the final version of the code segment, altered to handle a prefetch distance of three. Now that we are prefetching three iterations in advance, we have expanded the prologue to include a loop that prefetches several cache blocks for the main loop's first three iterations. In addition, the main loop has been shortened to stop prefetching three iterations before the end of the computation. No changes are necessary for the epilogue, which carries out the remaining iterations with no prefetching. This basic approach could also be applied, with some refinements, to nested loops.

Applying software prefetching

Sophisticated compiler algorithms based on this approach have been developed to automatically add prefetching during a compiler's optimization pass, with varying degrees of success.[1,4] Because the compiler must be able to reliably predict memory access patterns, prefetching is normally restricted to loops with relatively simple array-referencing patterns. Such loops are relatively common in scientific programs but much less common in general applications.

Attempts to establish similar software prefetching strategies for general applications with irregular data structures have met with limited success.[5,6] Because such applications do not follow a predictable execution path, it is difficult to anticipate their memory-referencing patterns.

General applications also show much more data reuse than scientific applications. This reuse often leads to high cache utilization, which diminishes prefetching's benefits.

Performance penalties

Using explicit fetch instructions exacts a performance penalty that must be considered when using software-initiated prefetching. The execution of fetch instructions increases a processor's total execution time. In addition, processors must calculate and store fetch instructions' source addresses. Software prefetching also results in significant code expansion.

```
for (i = 0; i < N; i++)
       sum = a[i] + sum;
```

(a)

```
for (i = 0; i < N; i++){
       fetch( &a[i+1]);    Prefetch for array a
       sum = a[i] + sum;
  }
```

(b)

```
for (i = 0; i < N; i+=4){
       fetch( &a[i+4]);
       sum = a[i] + sum;   Unroll loop four times
       sum = a[i+1] + sum;
       sum = a[i+2] + sum;
       sum = a[i+3] + sum;
  }
```

(c)

```
fetch( &sum);               Prologue prefetches data
fetch( &a[0]);                 for first iteration

for (i = 0; i < N-4; i+=4){
       fetch( &a[i+4]);
       sum = a[i] + sum;
       sum = a[i+1] + sum;
       sum = a[i+2] + sum;
       sum = a[i+3] + sum;
}
for ( ; i < N; i++)        Epilogue eliminates
       sum = a[i] + sum;   unnecessary prefetches
                           in last iteration
```

(d)

```
fetch( &sum);
for (i = 0; i < 12; i += 4)
       fetch( &a[i]);

for (i = 0; i < N-12; i += 4){
       fetch( &a[i+12]);   Increase prefetch
       sum = a[i] + sum;   distance to three
       sum = a[i+1] + sum;
       sum = a[i+2] + sum;
       sum = a[i+3] + sum;
}
for ( ; i < N; i++)
       sum = a[i] + sum;
```

(e)

Figure 1. Prefetching for loops. In the segment in (a), the loop sums the elements of array a, but there will be a cache miss every fourth iteration. The segment in (b) uses prefetch directives (shown in blue) to eliminate the cache misses. However, this causes unnecessary prefetch operations, which introduce prefetch overhead and degrade performance. The segment in (c) unrolls the loop. This prevents redundant prefetches. However, cache misses still occur during the loop's first iteration and unnecessary prefetches occur in the last iteration. The segment in (d) eliminates these problems by using software pipelining, which uses loop prologues and epilogues. The programs in (a) through (d) assume that prefetching data one iteration in advance will hide latency. In situations where that isn't the case, the prefetch distance must be increased to prefetch data further in advance, as the segment in (e) shows.

Hardware-based prefetching techniques do not incur the instruction overhead associated with the use of explicit fetch instructions.

HARDWARE PREFETCHING

Hardware-based prefetching techniques do not incur the instruction overhead associated with the use of explicit fetch instructions. These techniques do not require changes to existing executables, so there is no need for programmer or compiler intervention. However, without the benefit of compile-time information, hardware prefetching relies on speculation about future memory-access patterns based on previous patterns. Incorrect speculation will cause the memory system to bring unnecessary blocks into the cache. These unnecessary prefetches do not affect correct program behavior, but they can cause cache pollution and consume memory bandwidth.

Sequential prefetching

The effectiveness of using large cache blocks to prefetch data is limited by cache pollution.[7] As the cache block's size increases, so does the amount of useful data displaced from the cache to make room for the block.

In shared memory multiprocessors with private caches, increasing the size of cache blocks increases the likelihood that multiple processors will want data from the same block. This would increase *false-sharing effects*, which occur when multiple processors try to access different words from the same cache block and at least one of the accesses is a `store`.

The cache hardware operates only on whole cache blocks, so the accesses are treated as operations applied to a single object. Cache coherence traffic is thus generated to ensure that the changes made to a block by a `store` operation are seen by all processors caching the block. However, in false sharing, this traffic is unnecessary because only the processor that executes the `store` references the word being written.

Sequential prefetching can take advantage of spatial locality by prefetching consecutive smaller cache blocks, without introducing some of the problems associated with large cache blocks. Sequential prefetching can be implemented with relatively simple hardware.

OBL approach. The simplest sequential prefetching schemes are variations upon the *one-block-lookahead* (OBL) approach, which automatically initiates a prefetch for block b + 1 when block b is accessed. This approach differs from simply doubling the block size because the demand-fetched block and the prefetched block are considered separate items for cache replacement and coherency purposes. The use of separate, smaller blocks means the computer does not have to evict large amounts of data each time it replaces items in the cache. Smaller blocks also reduce the chance of false sharing.

OBL implementations differ depending on what type of access to block b initiates the prefetch of b + 1:[8]

- The *prefetch-on-miss* algorithm initiates a prefetch for block b + 1 whenever an access for block b results in a cache miss. If b + 1 is already cached, no memory access is initiated.
- The *tagged prefetch* algorithm associates a tag bit with every cache block. This bit detects when a block is fetched or when a prefetched block is referenced for the first time. In either case, the next block in memory is fetched.

Tagged prefetching is more expensive to implement because of the addition of the tag bits to the cache and the need for a more complex cache controller design. However, tagged prefetching eliminated misses in a unified (data and instruction) cache more than twice as well as the prefetch-on-miss approach.[8] For example, with a strictly sequential referencing pattern, the prefetch-on-miss algorithm will result in a cache miss for every other cache block because a prefetch for block b + 1 is not issued until the miss for b occurs. Tagged prefetching, on the other hand, can initiate a "domino effect" that avoids all but the first miss. So, when the prefetched block b + 1 is accessed, a prefetch for b + 2 is initiated, and when b + 2 is accessed, b + 3 is prefetched. The process continues until the sequential access stream terminates.

A shortcoming of OBL schemes is that the memory system may not initiate a prefetch for data far enough in advance of the data's actual use to avoid a processor memory stall. A sequential access stream resulting from a tight loop, for example, may not allow sufficient time between the use of blocks b and b + 1 to completely hide the memory latency.

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*. As the memory system accesses each prefetched block, b, for the first time, it interrogates the cache to determine whether blocks b + 1, . . . , b + K are present. If not, it fetches the missing blocks from memory.

However, this approach also generates memory traffic and cache pollution during program phases that have little spatial locality, which can make it impractical for large values of K.[7]

Adaptive sequential prefetching. An adaptive sequential prefetching policy[9] lets K's value vary during program execution to match the program's degree of spatial locality at a given time. The cache periodically calculates a *prefetch efficiency* metric to determine a program's current spatial locality characteristics.

Prefetch efficiency is the ratio of the number of times a prefetched block results in a cache hit to the total number of prefetches. The value of K is initialized at one, increases when the prefetch efficiency exceeds a predetermined upper threshold, and decreases when

Prefetching in the HP PA-RISC Family

Steven VanderWiel and David Lilja, University of Minnesota
Wei Hsu, Hewlett-Packard Corp.

We ran trials of two recent superscalar implementations of Hewlett-Packard's PA-RISC architecture. The trials served as examples of how hardware and software data prefetching are supported in contemporary microprocessors. The PA7200 and PA8000 both used the same high-performance multiprocessor bus.

The PA7200[1] was configured with a 256-Kbyte data cache and was clocked at 120 MHz. The PA7200 implemented a tagged, hardware-initiated prefetch scheme using either a directed or an undirected mode. In the undirected mode, the cache automatically prefetches the next sequential line. In the directed mode, the processor determines a prefetch direction (forward or backward) and distance via an autoincrement amount encoded in the `load` or `store` instructions. The processor then passes this information on to the data cache. In other words, when an address register's contents are autoincremented, the cache block associated with the newly computed address is prefetched.

The PA8000[2] test system contained a 1-Mbyte data cache and ran at 180 MHz. The PA8000 used explicit prefetch instructions inserted by the HP-PA compiler.

Figure A shows data prefetching's effect on the performance of the PA7200 and PA8000. The effect on performance is calculated by dividing the runtime of each SPECfp95 benchmark achieved without prefetching by the runtime achieved with prefetching.

Of the 20 trials, 16 showed improved performances, three showed little or no change in runtime, and one showed degraded performance. Both the PA7200 and PA8000 showed significant performance improvements for applications that typically exhibit poor cache utilization, such as tomcatv, swim, su2cor, hydro2d, mgrid, and applu.

Of the remaining programs, fpppp and wave5 showed little or no performance improvement because of their inherently better cache utilization. The turb3d and apsi applications contain complex memory-referencing patterns that inhibit prefetching. The PA7200's hardware prefetching scheme's low overhead performed slightly better for such programs. In fact, software prefetching and its higher overhead actually degraded performance for the turb3d benchmark.

Although the degree to which prefetching helps a particular program depends on application characteristics, data from the trials show that prefetching generally improves performance. In the case of software prefetching, performance lost due to the addition of prefetch instructions can be regained by recompiling the program so that prefetching is disabled.

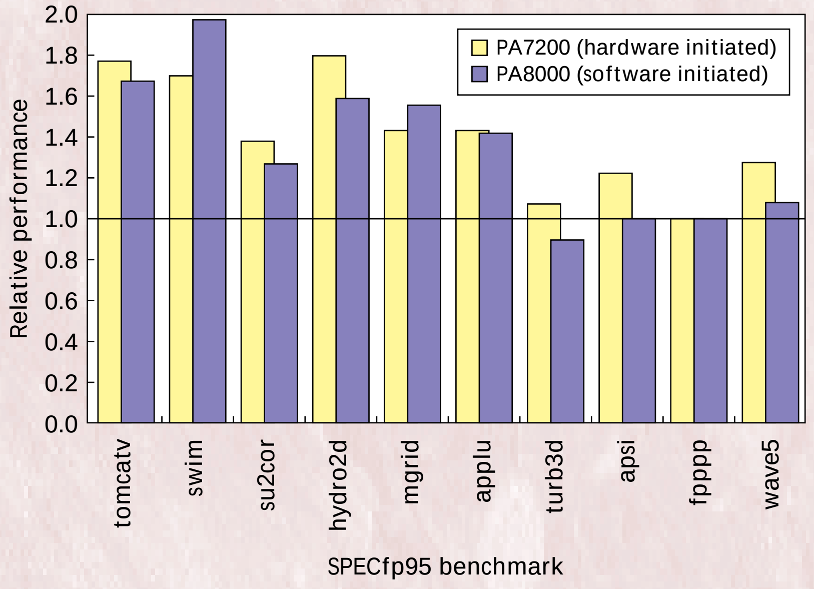

Figure A. The effects of prefetching on the performance of the HP PA7200 and PA8000 processors calculated for each of 10 SPECfp95 benchmarks by dividing the runtime achieved without prefetching by the runtime achieved with prefetching. In the graph, the 16 results that are greater than one reflect speedup. The one result that is less than one reflects a slowdown. The other three results reflect little or no performance change.

References

1. K.K. Chan et al., "Design of the HP PA 7200 CPU," *Hewlett-Packard J.*, Feb. 1996, pp. 25-33.
2. D. Hunt, "Advanced Performance Features of the 64-bit PA8000," *Proc. 40th IEEE Int'l Computer Conf.*, IEEE CS Press, Los Alamitos, Calif., 1995, pp. 123-128.

the efficiency drops below a lower threshold. If K is reduced to zero, prefetching ends. At this point, the prefetch hardware begins to determine how frequently a cache miss to block b occurs while block b – 1 is cached. The hardware restarts prefetching if the frequency exceeds the designated lower threshold.

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

Prefetching with arbitrary strides

When the processor's referencing pattern strides through nonconsecutive memory blocks, sequential prefetching will cause needless prefetches and will thus become ineffective. You need more elaborate prefetch-

A separate cache, called an RPT, stores information that lets it predict when to prefetch data, which is very helpful when processors have data-referencing patterns that are irregular or not consecutive.

ing techniques to take advantage of both small and large strided array-referencing patterns while ignoring references that are not array-based.

One such technique employs special prefetch hardware that monitors the processor's address-referencing pattern and infers prefetching opportunities by comparing successive addresses used by `load` or `store` instructions. If the prefetch hardware detects that a particular `load` or `store` is generating a predictable memory-addressing pattern, it will automatically issue prefetches for that instruction.

To illustrate one aggressive scheme,[10] assume that a memory instruction, m_i, references addresses a_1, a_2, and a_3 during three successive loop iterations. The prefetch hardware initiates a prefetch for m_i if $(a_2 - a_1) = \Delta \neq 0$, where Δ is 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 a_3. Prefetching continues in this way until the equality $A_n = a_n$ is no longer true. At this point, prefetching for instruction m_i ends.

REFERENCE PREDICTION TABLE

To implement this approach, it is necessary to store the previous address used by a memory instruction along with the last detected stride, if there has been one. It is clearly impossible to record the reference histories of every memory instruction. Instead, a separate cache called the *reference prediction table* (RPT) holds this information for most of the recently used memory instructions. Table entries contain a memory instruction's address, the previous address accessed by the instruction, a stride value for entries that have established a stride, and a field that records the entry's current state.

The table is indexed by the CPU's program counter. When the CPU executes memory instruction m_i for the first time, it enters the instruction in the RPT with its state set to initial. This shows that the RPT has not initiated prefetching for this instruction. If m_i is executed again before its RPT entry has been evicted, the RPT calculates a stride value by subtracting the instruction's most recent address stored in the RPT from the current address.

How an RPT works

Figure 2 illustrates how an RPT would work during the execution of a matrix multiply loop.

For simplicity, we consider only the `load` instructions for arrays a, b, and c, and we assume that each cache block contains one word. We also assume that arrays a, b, and c begin at addresses 100,000, 200,000, and 300,000, respectively.

Figure 2b shows the RPT's state after the inner loop's first iteration. Instruction addresses are represented by their pseudocode mnemonics. Since the processor has not executed any of the memory instructions yet, each entry is in an initial state, each stride is zero, and all references to the instructions result in a cache miss.

Figure 2c shows how this changes after the second iteration. We assume the `load` instruction for array a occurs outside the innermost loop, so its RPT entry remains unchanged. The instructions for arrays b and c are in a transient state because they have new addresses and strides. This indicates that an instruction's referencing pattern may be in transition. As the RPT has not yet issued any prefetches, references to arrays b and c will again result in cache misses.

However, the RPT will now issue tentative prefetches for the `load` instructions for arrays b and c based on their newly computed strides. The RPT will calculate the prefetch addresses by adding the previous address field to the stride field for each entry. For example, with the `load` for array b in Figure 2c, the RPT would issue a prefetch for the block at address 200,008, which is 200,004 (the previous address) plus four (the stride).

During the third iteration, shown in Figure 2d, the `load` instructions for arrays b and c move to the steady state when the RPT finds that the tentative strides computed in the second iteration have stayed the same. The tentative prefetches issued during the second iteration have already fetched the requested blocks into the cache, resulting in cache hits for the references to arrays b and c.

The remaining iterations of the inner loop proceed without cache misses. However, during the last iteration, the prefetch issued for array c results in an incorrect prediction because the next element of this array is actually at a lower address than the one predicted by the RPT. This occurs because of the way the arrays are stored in memory.

The RPT entry for array c will return to the initial state when the RPT detects that an incorrect prediction has occurred, as Figure 2e shows, until a reference pattern can again be established. A cache miss will thus result when the processor issues a `load` instruction for array c.

RPT limitations

We can see that the RPT improves upon sequential policies by correctly handling large strided array references. However, the RPT will suffer initial misses while a reference pattern is being established. The RPT will also issue unnecessary prefetches at the end of a sequential reference stream or when the reference pattern is discontinuous.

In the basic scheme, the RPT prefetches only one array stride ahead of the current address. If this prefetch distance is insufficient to hide the latency of the main-memory accesses, processor stall cycles will occur.

However, it is possible to adjust the RPT to use longer prefetch distances by adding a distance field to each RPT entry. The prefetch address for an entry can then be calculated by adding the product of the entry's stride and distance fields to the previous address field. There are techniques[10] for establishing an appropriate value for the distance field, although they add significantly to the RPT's complexity.

The degree to which prefetching can improve runtime performance depends on the application and computer architecture. Applications that already exhibit good cache performance or that produce highly irregular memory-referencing patterns do not typically benefit from prefetching. However, applications that iterate over large arrays can show marked improvements in runtimes with the addition of prefetch hardware or prefetch instructions. Prefetching is also more beneficial to systems with large latencies in main-memory accesses because memory stall cycles in these systems represent a large fraction of overall program runtime.

Researchers are exploring ways to improve prefetching techniques and extend them to a greater variety of architectures and applications. For example, techniques that combine hardware prefetching's low overhead and software prefetching's accuracy are promising.

Researchers also must look for ways to use prefetching to reduce the many memory stall cycles that occur when running applications that process large but irregular data structures. In addition, systems with complex memory hierarchies, such as distributed, shared memory multiprocessors, will also need novel prefetching mechanisms to mask their long memory latencies. ❖

Acknowledgments

This work was supported in part by the US Army Intelligence Center and Fort Huachuca under Contract No. DABT63-95-C-0127 and ARPA Order No. D 346r. VanderWiel's work was supported in part by an IBM Graduate Fellowship.

References

1. T.C. Mowry, S. Lam, and A. Gupta, "Design and Evaluation of a Compiler Algorithm for Prefetching," *Proc. Fifth Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, ACM Press, New York, 1992, pp. 62-73.
2. T.F. Chen and J.-L. Baer, "A Performance Study of Software and Hardware Data Prefetching Schemes," *Proc. 21st Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1994, pp. 223-232.
3. F. Dahlgren and P. Stenström, "Effectiveness of Hard-

```
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)

Address tag	Previous address	Stride	State
ld a[i] [j]	100,000	0	Initial
ld b[i] [k]	200,000	0	Initial
ld c[k] [j]	300,000	0	Initial

i = 0, j = 0, k = 1

(b)

Address tag	Previous address	Stride	State
ld a[i] [j]	100,000	0	Initial
ld b[i] [k]	200,004	4	Transient
ld c[k] [j]	300,400	400	Transient

i = 0, j = 0, k = 2

(c)

Address tag	Previous address	Stride	State
ld a[i] [j]	100,000	0	Initial
ld b[i] [k]	200,008	4	Steady
ld c[k] [j]	300,800	400	Steady

i = 0, j = 0, k = 3

(d)

⋮

Address tag	Previous address	Stride	State
ld a[i] [j]	100,004	4	Transient
ld b[i] [k]	200,008	4	Steady
ld c[k] [j]	300,004	0	Initial

i = 0, j = 1, k = 1

(e)

Figure 2. A reference prediction table (RPT) during the execution of a matrix multiply loop, shown in (a). When dealing with large and small array-indexing patterns, a separate cache, the RPT, can monitor the array addresses a processor references and infer future prefetching opportunities. In (b), after the inner loop's first iteration, the processor has not executed any memory instructions. So the entries for the instructions for arrays a, b, and c (top to bottom in the RPT) are in an initial state, and each stride is zero. In (c), after the second iteration, the instructions for arrays b and c have new addresses and strides, so they are in a transient state, indicating that their referencing patterns are in transition. In (d), after the third iteration, the instructions for arrays b and c have the same strides they had after the second iteration, so they are in a steady state, and a referencing pattern has been established. This lets the RPT predict when to issue prefetches for data until—as (e) shows for array c's instruction—an entry's stride and referencing pattern changes. When an entry changes from a steady state, it returns to an initial state.

ware-based Stride and Sequential Prefetching in Shared-memory Multiprocessors," *Proc. First IEEE Symp. High-Performance Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1995, pp. 68-77.
4. D. Bernstein, C. Doron, and A. Freund, "Compiler Techniques for Data Prefetching on the PowerPC," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques*, ACM Press, New York, pp. 16-19.
5. M.H. Lipasti et al., "SPAID: Software Prefetching in Pointer and Call-Intensive Environments," *Proc. 28th Int'l Symp. Microarchitecture*, IEEE CS Press, Los Alamitos, Calif., 1995, pp. 231-236.
6. C.K. Luk and T. C. Mowry, "Compiler-Based Prefetching for Recursive Data Structures," *Proc. Ninth Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, ACM Press, New York, 1996, pp. 222-233.
7. S. Przybylski, "The Performance Impact of Block Sizes and Fetch Strategies," *Proc. 17th Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1990, pp. 160-169.
8. A.J. Smith, "Cache Memories," *Computing Surveys*, Sept. 1982, pp. 473-530.
9. F. Dahlgren, M. Dubois, and P. Stenström, "Fixed and Adaptive Sequential Prefetching in Shared-memory Multiprocessors," *Proc. 1993 Int'l Conf. Parallel Processing*, CRC Press, Boca Raton, Fla., 1993, pp. I56-I63.
10. T.F. Chen and J.-L. Baer, "Effective Hardware-Based Data Prefetching for High Performance Processors," *IEEE Trans. Computers*, May 1995, pp. 609-623.

Steven P. VanderWiel is a PhD candidate and IBM Research Fellow in the Department of Electrical Engineering at the University of Minnesota. His research interests include computer architecture and parallel processing. VanderWiel received a BS and an MS, both in computer engineering, from Iowa State University. He is a member of the IEEE Computer Society and the IEEE.

David J. Lilja is an associate professor and the director of graduate studies in computer engineering in the Department of Electrical Engineering at the University of Minnesota. His main research interests are computer architecture, parallel processing, high-performance computing, and the interaction of compilation technology and computer architecture. Lilja received a BS in computer engineering from Iowa State University and an MS and a PhD, both in electrical engineering, from the University of Illinois, Urbana-Champaign. He is a senior member of the IEEE and a member of the ACM.

Contact VanderWiel and Lilja at {svw,lilja}@ee.umn.edu.