Improving performance of linear algebra algorithms for dense matrices, using algorithmic prefetch

by R. C. Agarwal
F. G. Gustavson
M. Zubair

In this paper, we introduce a concept called *algorithmic prefetching*, for exploiting some of the features of the IBM RISC System/6000® computer. Algorithmic prefetching denotes changing algorithm A to algorithm B, which contains additional steps to move data from slower levels of memory to faster levels, with the aim that algorithm B outperform algorithm A. The objective of algorithmic prefetching is to minimize any penalty due to cache misses in the innermost loop of an algorithm. This concept, along with "cache blocking," can be exploited to improve the performance of linear algebra algorithms for dense matrices. We experimentally demonstrated the impact of prefetching on two dense-matrix operations. For one operation, the performance was improved from 74% of peak to 89% of peak by algorithmic prefetching; for the second operation, it was improved from 73% to 87% of the peak performance.

Introduction

To achieve good performance on high-performance workstations, it is essential that the underlying algorithms be restructured to *match* the underlying architectures of the workstations. In this paper, we restrict our discussion to IBM RISC System/6000® (RS/6000) workstations (note that we use RS/6000 to indicate IBM POWER models);

The work of M. Zubair was done while he was visiting the IBM Thomas J. Watson Research Center from Old Dominion University during the summer of 1992.

however, the ideas presented for enhancing the performance of numerically intensive computations (NIC)[1] are general and can be applied to other workstations possessing similar hardware characteristics—for example, IBM POWER2™ models. These high-performance workstations are similar, in some sense, to vector pipeline machines. It has been shown by Dongarra, Gustavson, and Karp [1] that it is possible to restructure linear algebra algorithms to match the architecture of vector pipeline machines. A major objective of restructuring is to reuse data (keep data in cache) and thereby reduce references to main memory. In [1], the focus was on Cray-1-type machines, which have a single level of memory. With the advent of the IBM 3090 vector facility, Cray-2-type machines, and other similar machines, memory hierarchies, most featuring caches, became a very important consideration for overall NIC performance [2]. In January 1987, J. Dongarra hosted a meeting[2] at the Argonne National Laboratory in which the Level-3 basic linear algebra subroutines (BLAS) [3], a subroutine library usually incorporating features for efficient cache usage, were proposed to computer vendors and numerical analysts, with the view that they, like the Level-1 and the Level-2 BLAS, become an industry standard. The IBM Engineering and Scientific Subroutine Library (ESSL) [4] recognized at an early stage the importance of Level-3 BLAS and included a version of the double-precision general matrix-multiply routine (DGEMM), which takes advantage of memory-hierarchy features, even before the standard DGEMM was accepted.

Agarwal and Gustavson [5] have developed high-performance, cache-based "blocking" algorithms for Level-3 BLAS. The key feature of these schemes is to bring a block of data into cache once and use it several times before replacing it with a new block.

The architecture of the IBM 3090 machines with vector facility does not support chaining, a feature for overlapping phases of several vector operations. (For a complete description of chaining, see [1].) To overcome the absence of this feature, the 3090 vector facility introduced into its architecture two compound instructions that simulated the same performance as vector chaining, when data were in vector registers. These instructions are the vector multiply add (DAXPY) and vector multiply accumulate (dot product). A major aim was to keep data in vector registers and to store the results only after many operations were computed on those data.

- *Prefetching*

The term *prefetching* denotes accessing a memory word that is not in cache several cycles ahead of its usage. The objective of prefetching is to hide memory-access latencies due to cache misses. There is a lot of interest in prefetching in the research area of architectural support for programming languages, for example [6–9]. Most of this effort is directed toward the highly desirable goal of incorporating prefetching into the compiler, so that no changes are required in the user program. To implement prefetch instructions, this approach requires additional hardware which is not currently available on commercial workstations. It will take some time, however, before such compilers, along with the required architecture/hardware support, are available. To this effect, Callahan, Kennedy, and Porterfield state in the conclusion of [6], "... it remains to be established in practice whether the advanced design of new high-performance microprocessors will reduce the prefetching overhead sufficiently to realize the potential gains." References [6–9] demonstrate the potential gains due to prefetching. So far, however, all of their results are obtained from simulations in which existing architectures are modified to support prefetching.

To be effective, the prefetch instructions should be used judiciously—only for those arrays that are likely to cause cache misses. The use of prefetch instructions for all arrays may overwhelm the system and may actually degrade the overall performance, as was demonstrated by some of the simulations in [6–9]. In compiler-initiated prefetching (with or without special hardware), the compiler must analyze the data-access pattern in the program. It also must know the dimensions of the various arrays and the repetition counts for the various loops. By combining this information with the relevant cache-size parameters, the compiler can estimate which data accesses are likely to result in cache misses. For simple programs, the compiler is likely to do a reasonable job of prefetching, but for more complex programs and when the array dimensions and loop-repetition counts are not known at compile time, it may be very difficult for the compiler to judiciously insert prefetch instructions.

- *Algorithmic prefetching*

As an alternative to compiler prefetching, we propose and demonstrate the usefulness of a concept, which we refer to as *algorithmic* prefetching, that can be implemented on existing workstations with standard compilers. In algorithmic prefetching, we transform a given algorithm A to algorithm B, where B implements A yet has additional steps that move data from slower levels of memory to faster levels (e.g., cache, registers). We then present algorithm B to a compiler that need not have a prefetching capability. We have demonstrated that the additional source statements in algorithm B enhance the overall performance; i.e., algorithm B outperforms algorithm A. (We have found, however, that the XLF compiler [10] sometimes "gets confused" when trying to optimize

[1] A list of the acronyms used in this paper can be found in the Appendix.
[2] Preliminary meeting on BLAS 3 adoption, Argonne National Laboratory, Argonne, IL, January 27–29, 1987.

algorithm B and therefore produces code that performs poorly.)

An application program writer understands his algorithm very well and can visualize the data-access pattern; if necessary, he can modify the algorithm and the resulting data-access pattern to better match the memory and cache parameters of the machine. It is unrealistic to expect this degree of sophistication from today's compilers. Algorithmic prefetching requires additional human effort, and this is certainly worthwhile for heavily used library programs such as the BLAS.

In algorithmic prefetching, the user anticipates cache misses that may occur in accessing an array and issues an ordinary load instruction for a single doubleword[3] from the array, which is likely to result in a cache miss. This prefetch load is done sufficiently in advance of the actual use of data from the missing cache line. This brings the missing cache line into the cache before it is actually needed and hides (overlaps) the cache-miss penalty with some useful work. If the prefetch load does not result in a cache miss, it is like an ordinary load instruction. In fact, if the user is accessing two arrays, and only one of them (he does not know which one) is likely to result in a cache miss, he issues two loads, and only one of them results in a cache miss requiring servicing from the hardware.

We now mention some specifics of algorithmic prefetching for the underlying architecture of the RS/6000 family of workstations and the XLF FORTRAN compiler [10] for the RS/6000 family. This workstation architecture does not have a prefetching feature, nor does the XLF FORTRAN compiler support prefetching. We have been able to accomplish algorithmic prefetching by writing code in FORTRAN and by using loop unrolling, replacing inner-loop code code that is iterated n times with m replications of that inner-loop code, iterated n/m times. A scalar variable is assigned the value of an array doubleword that is likely to be out of cache; i.e., the assignment will cause a cache miss. This scalar variable may be either actually employed for a useful computation or simply disregarded. If the variable is actually used in the loop, the number of loads in the loop remains the same, and *there is no additional overhead for prefetching.* As pointed out earlier, for the prefetching to be fully effective, the load must be done several cycles in advance, equal in time to the cache-miss latency. (For the RS/6000, this latency is 11–16 cycles.) Therefore, a large degree of unrolling may be necessary in order to use the prefetched variable efficiently. Some partial benefit from prefetching can be obtained, even when prefetching is not done sufficiently in advance to hide the entire cache-miss latency. If the prefetched variable is not used in the loop, the number of loads in the loop will be one more than necessary. In some situations, this may degrade performance slightly. Also, since the prefetched variable is not used, an optimizing compiler might eliminate that load and therefore thwart the attempt to use algorithmic prefetching. Since this situation occurred for the XLF compiler, we used a dummy computation involving the prefetched variable outside the loop so that the compiler would not remove it.

We now mention some fine points about algorithmic prefetching. For those loops in which the loop performance is limited by the number of loads and for which the data are actually in cache, performance is degraded slightly because of any extra prefetch load instruction. If the prefetched load actually results in a cache miss, however, the overall performance generally improves in spite of the additional load. Our use of algorithmic prefetching is implemented via ordinary load instructions. The final iteration(s) of the loop may have to be treated specially; otherwise, the inner loop involving such iteration(s) might attempt to access doublewords beyond the array boundary. This is easily accomplished by reducing the loop count by one (or more) and doing the last iteration(s) outside the loop, without prefetching.

The algorithmic prefetching concept, in addition to cache blocking, can be exploited to improve the performance of linear algebra algorithms for dense matrices (matrices not specially treated as "sparse"). Alpern and Carter applied a type of algorithmic prefetching to the DGEMM computation on the RS/6000 workstation.[4] For Level-3 BLAS, however, the use of cache blocking provides similar results. Algorithmic prefetching works best in the context of Level-2 computations, where cache blocking cannot help, since there is no reuse of data. As is shown later, on RS/6000 it is possible to apply algorithmic prefetching for only those algorithms that have more floating-point operations than load and store operations in the innermost loop.

- *Experimental results*

We have experimentally demonstrated the impact of algorithmic prefetching on two dense-matrix operations, multiplication of a matrix by two vectors, and multiplication of a complex matrix by a complex vector. The experiments were done on the RS/6000 Model 530. All coding was done in standard FORTRAN. The performance of the first operation with algorithmic prefetching improves from 74% (37 MFLOPS) to 89% (44.5 MFLOPS) of the peak performance; and for the second operation, from 73% (36.5 MFLOPS) to 87% (43.5 MFLOPS) of the peak performance of the machine.

The rest of the paper is organized as follows. In the next section, we briefly discuss the hardware features of the RS/6000 workstation. The following section discusses the

[3] We use the term *doubleword* (eight bytes, the amount of memory required to store a double-precision floating-point number) in this paper to signify a data item.

[4] Private communication, J. Lawrence Carter, IBM Thomas J. Watson Research Center, January 1990.

general matrix vector (GEMV) computation and the general algorithmic prefetching concept. In the next section, we give details of two dense-matrix operations in which the algorithmic prefetching concept can be used. Experimental results are discussed in the following section. Finally, we give some conclusions in the last section.

RS/6000 architecture

In this section, we give a brief description of the RS/6000 architecture. Key architectural features of the RS/6000 computers that greatly influence their performance are

- A large set of registers, typically 32.
- A memory hierarchy.
- The parallel execution of the branch, fixed-point, and floating-point functional units.
- A pipelined floating-point unit, which produces one multiply–add operation per cycle.

For a detailed review of the hardware features of the RS/6000 computer, one can refer to [11].

- *Data-cache unit*

We consider the RS/6000 systems that have four-way set-associative, 64KB caches. Each set has 16 KB of fast memory and consists of 128 lines of 128 bytes each. In the following, we summarize some features of the cache that have an impact on the performance:

- A load instruction takes one cycle to execute if the word is in cache and eight cycles if the word is not in cache and all previous cache misses have been completely processed.
- Following the eight-cycle penalty, two consecutive doublewords in the same cache line can arrive in the cache during each following cycle. Thus, the 16 doublewords in a line arrive in 15 cycles.
- In case of a cache miss, doublewords of the appropriate cache line are brought from the main memory in the order $d_j, d_{j+1}, \cdots, d_{lsize}, d_1, d_2, \cdots, d_{j-1}$, where $d_1, d_2, d_3, \cdots, d_{lsize}$ is the order of the consecutive doublewords corresponding to a cache line in the main memory and d_j is the requested data item (not necessarily on the cache-line boundary). We refer to this order as the *requested-word-first* order.

- *Translation lookaside buffer*

The user program and data are located in a virtual address space, which is translated to a real address space. The translation lookaside buffer (TLB) contains the translation information for virtual pages (4 KB each). The RS/6000 TLB has 128 entries in a two-way set-associative table; i.e., at any time, up to 128 virtual page addresses can be translated. If the translation information for a virtual page is not in the TLB, a TLB miss occurs, and it can take up to 40 cycles to bring the translation information into the TLB. Therefore, in addition to minimizing cache misses, it is imperative to block the problem in order to minimize TLB misses.

- *Superscalar features*

Some of the salient features that enable the RS/6000 to give very good floating-point performance are summarized below:

- When data and instructions remain in cache and TLB,[5] the RS/6000 is capable of simultaneously executing four instructions per machine cycle: a branch, a condition-code-logic instruction, a fixed-point instruction, and a floating-point instruction. The floating-point instruction can be a compound multiply–add instruction (FMA).
- All arithmetic is done between registers. There are 32 fixed-point and 32 floating-point registers.
- The FMA instruction is executed in a two-stage pipeline. The delay in each stage is one cycle; that is, after the first cycle, a second FMA instruction may start, provided that it does not need the output of the first FMA instruction.
- A floating-point load instruction is done by the fixed-point instruction unit, and can be done concurrently with a floating-point-arithmetic instruction that does not use the value being loaded.

Level-2-like computation

- *GEMV: Multiplication of a real matrix with a vector*

Suppose we wish to perform a Level-2 computation, say GEMV: $\mathbf{y} = \mathbf{y} + \mathbf{Ax}$, where vector $\mathbf{y}$ has dimension M; vector $\mathbf{x}$ has dimension N; and the matrix $\mathbf{A}$ has dimension $M \times N$. Assume that $\mathbf{A}$ is so large that it does not fit into the cache. (In this context, fitting $\mathbf{x}$ and $\mathbf{y}$ is a negligible problem.) While a column of $\mathbf{A}$ is processed, for every sixteen doublewords of $\mathbf{A}$ (128 bytes), a cache miss of eight cycles occurs (a so-called "hiccup") during which processing stops. After the first doubleword arrives, the remaining fifteen doublewords arrive at a rate of two doublewords per cycle, so that all sixteen doublewords arrive in cache in 15–16 cycles. Note that here and throughout our paper we assume FORTRAN-like storage for the arrays (see below for a description of how FORTRAN stores arrays).

The innermost loops of the GEMV computation, which perform all necessary calculations for a submatrix of $\mathbf{A}$, can be described by the pseudocode in **Figure 1**. Vector $\mathbf{y}$ has dimension M; vector $\mathbf{x}$ has dimension N; and the

[5] By this, we mean that the data and instructions remain in the memory space associated with the 128 pages whose translation information currently resides in the TLB.

submatrix of **A** has dimension $q \times n$, where $q \leq M$, $n \leq N$. The symbol f*k* represents floating-point register k. After the first execution of the inner loop of Figure 1, the values of the first q $y(i)$ are updated by adding $\sum_{j=1}^{n} a(i,j)x(j)$. After each successive iteration of the inner loop, q more of the $y(i)$ values are similarly updated, until all of the M values of **y** have been updated. This completes the multiplication of **A** by the first n elements of **x**. In order to complete the multiplication, all of the code of Figure 1 must be repeated for successive groups of n elements of **x** (and n columns of **A**) until all N elements of **x** have been multiplied.

Note that $q + 2$ floating-point registers are required to compute q doublewords of **y** (registers f0 and f31 are used as working registers). Thus, $q \leq 30$. We wish to form a $q \times n$ submatrix of **A** that fits into cache and TLB. We call this submatrix a block of the matrix or, simply, a block (see ahead). When n is large, the q loads and stores of **y** in the outer loop of Figure 1 require much less time than the nq FMAs in the inner loop. The inner loop has $q + 1$ loads and q FMAs. Since q of the loads are executed simultaneously with the q FMAs, the inner loop runs at $q/(q + 1)$ times the peak performance if no cache miss occurs. If $q \geq 16$, a cache miss occurs at least once every time the inner loop is reentered, since each new column of **A** contains an untouched line. (We neglect cache misses caused by accessing **x**, because they are much less frequent.) In ESSL [4], we have chosen q to be $\approx$20.

The location in memory of matrix **A** is significant. Typically, we declare a two-dimensional array in FORTRAN as $A(LDA, *)$ where $LDA \geq M$. Here, LDA refers to the leading dimension of the array **A**. In what follows, a matrix stored this way is called a dense matrix. If the (1, 1) doubleword of matrix **A** is stored in memory location α, the FORTRAN convention of storing **A** places the (i, j)th doubleword of **A** in memory location $\alpha + i - 1 + (j - 1) \times LDA$. (We assume that this address and α are in units of doublewords.) It is important to realize that the value of LDA can influence how well blocks of **A** fit into cache and TLB. For $LDA \geq 512$, each column of **A** begins on a different page. In order to avoid a TLB miss, n must be chosen so that translation information for n pages fits comfortably into the TLB. We have determined experimentally that $n = 80$ is a good choice for the two-way set-associative TLB with 128 entries. Because $q \leq 30$ (there are 32 floating-point registers), we have $2qn \leq 4800$ doublewords; this is the number of doublewords needed to process two consecutive blocks of the matrix. We wish to keep two consecutive blocks in cache in order to keep all lines that span two blocks in cache until they are fully processed. To keep two consecutive blocks (for $q \leq 16$, see the last paragraph of the subsection on DGEMV2, below) in cache, we may need to keep up to three cache lines per column of **A** in cache. These cache lines remain

```
DO i = 1,M,q
   f1 = y(i)
   ...
   fq = y(i+q-1)
   DO  j = 1,n
      f31 = x(j)
      f0  = a(i,j)
      f1  = f1 + f0*f31
      f0  = a(i+1,j)
      f2  = f2 + f0*f31
      ...
      f0  = a(i+q-1,j)
      fq  = fq + f0*f31
   END DO
   y(i) = f1
   ...
   y(i+q-1) = fq
END DO
```

Figure 1

Pseudocode for the innermost loops of GEMV: $\mathbf{y} = \mathbf{y} + \mathbf{Ax}$, for the first n columns of **A** and first n components of **x**.

in cache for those matrices that do not have "bad" LDA.[6] Most LDA values are "good."

We cannot use the algorithmic prefetching concept for the GEMV computation, because the inner loop requires more floating-point load instructions (FPLs) than FMAs. In order for algorithmic prefetching to work, the inner loop must have sufficiently more FMAs than FPLs that a new line can be loaded into cache while excess FMAs are being executed. Thus, the GEMV computation must halt for eight or more cycles whenever a new line is encountered. Then, computing for the 16 doublewords in this line proceeds at a near-peak rate. The ESSL double-precision general matrix vector (DGEMV) algorithm is based on the algorithm described above; its computation rate on the Model 530 computer was 25–28 MFLOPS (50–56% of peak) for data in memory. For the DGEMV computation on the Model 530, Dongarra, Mayes, and Radicati di Brozolo [12] report 24 MFLOPS (48% of peak) for datain cache and 12.3 MFLOPS (24.6% of peak) for data in memory.

[6] In a four-way set-associative cache, a particular line from memory can go into any one of only four lines of cache. Usually, access patterns to memory are such that, as the cache fills, the groups of four lines are filled roughly equally. However, for data accessed with an LDA (difference between the address of successively accessed data items) that is a factor of 16 KB or a multiple of 16 KB, the effective cache size is reduced. For the worst case of an LDA of 16 KB or a multiple of 16 KB, the effective size of the cache is a mere four lines (512 bytes). The best general advice is to avoid values of LDA (strides) that are powers of 2.

```
FPL 1
FPA 1
FPL 2
FPA 2
...
FPL q
FPA q
FPA q+1
FPA q+2
...
FPA p
```

Figure 2

Unmodified inner loop of algorithm.

```
FPL 1
FPA 1
FPL 2
FPA 2
...
FPL q
FPA q
FPL 1     (for the next iteration)
FPA q+1
FPA q+2
...
FPA p
```

Figure 3

Inner loop of algorithm modified by addition of prefetch instruction.

• *Algorithmic prefetching*

As stated earlier, the idea of algorithmic prefetching is to hide the latency in accessing a doubleword that is not in cache. For this to be fully effective, at least eight instructions subsequent to the prefetch should not be fixed-point or branching instructions. To understand this better, consider **Figure 2**, which shows the unmodified innermost loop of an algorithm with p floating-point arithmetic (FPA) instructions and q FPLs, where $p \geq q$. We assume that the q loads are for consecutive words in the memory, and we load a different set of consecutive words in every iteration of the innermost loop. The q loads can be overlapped with the execution of the first q FPAs, as shown in Figure 2. Recall that an FPL can be done concurrently with an FPA.

Because of our assumption that **A** does not fit in cache, the first load will most likely miss the cache; hence, there will be a penalty of eight cycles in the innermost loop. To avoid this, we execute the first floating-point load of the $(i + 1)$th iteration during the ith iteration, immediately after the qth floating-point instruction, as is shown in **Figure 3**. This brings into cache all of the data necessary for the next iteration. Now the eight-cycle penalty due to this load is overlapped with the next $p - q$ floating-point arithmetic instructions. For full overlapping, it is necessary that $p - q \geq 8$.

This basic idea of algorithmic prefetching must be refined for dense-matrix processing and also to take into account cache misses that occur for words which may not reside on the cache line boundary. (In Figure 3, we assumed that the FPL 1 prefetch brought in all data necessary for iteration $i + 1$.) To understand this, consider processing a dense matrix **A** of size $M \times N$. The matrix is partitioned into horizontal blocks of size $q \times N$, as shown in **Figure 4**. The value of q is determined by the number of available floating-point registers and also by the nature of the underlying algorithm. Here, we consider only horizontal blocking; for large N, it may also be necessary to do vertical blocking, as for the GEMV multiplication in Figure 1. The vertical-blocking parameter is based on TLB considerations. For our discussion below, we assume that 1) two blocks can reside in cache simultaneously, and 2) in the innermost loop of the algorithm, a column of size q of a block is accessed and p floating-point operations are performed, where $p - q > 8$. We first consider the case in which the matrix is processed in natural order (block 1 followed by block 2, and so on).

• *Algorithmic prefetching with block processing in natural order*

We assume that when a block is processed, the columns are also accessed in natural order. The algorithmic

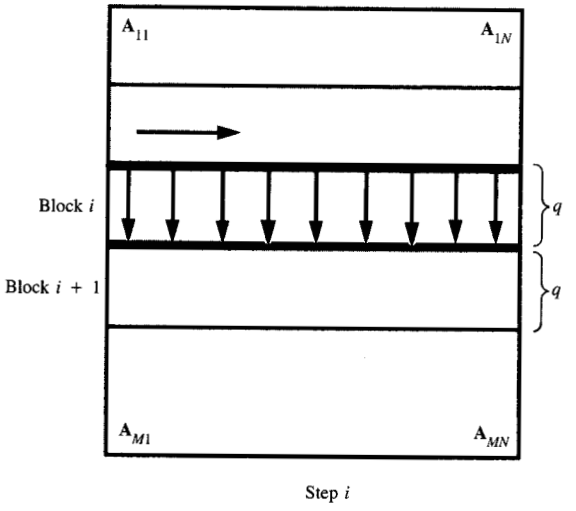

Figure 4

Processing matrix **A** blocks in natural order. The horizontal and vertical arrows indicate the order in which row and column elements are processed.

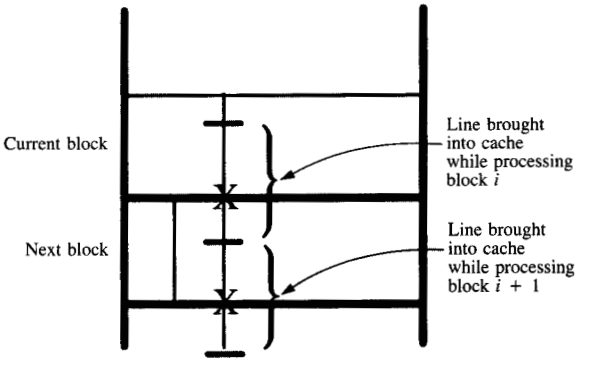

Figure 5

Prefetching while processing matrix **A** blocks in natural order. X designates prefetched doubleword.

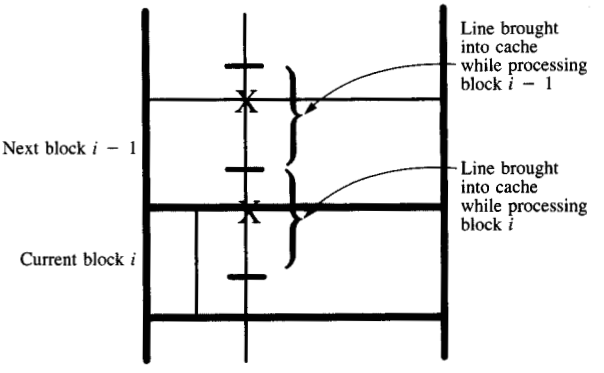

Figure 6

Algorithmic prefetching, processing blocks in reverse order; X designates prefetched doubleword.

prefetching scheme in this environment is illustrated in **Figure 5**. The prefetched doubleword is indicated by X. Notice that while working on a column of a block, we prefetch the *last* doubleword of the next column. We do not prefetch the *first* doubleword of the next column, as it is almost always already in cache, having been brought into cache during processing of the previous block. There is a problem with this algorithmic prefetching scheme, however, which arises because of the *requested-word-first* order in which doublewords corresponding to a cache line are brought into cache in case of a cache miss: If a cache miss occurs for the prefetched doubleword, the doublewords brought into the cache first are the ones that are not used in the processing of the current block; they belong to the next block. In other words, we are fetching doublewords into the cache that are not immediately required. This may stall the CPU for several cycles while waiting for the required doublewords in the current block. To overcome this problem, we prefetch with block processing in the reverse order.

Algorithmic prefetching with block processing in reverse order

In reverse-order processing, the last block of the matrix is accessed first, then the next-to-last block, and so on. The columns of a block are processed in natural order. Reverse-order processing generates output identical to that of natural-order processing. The algorithmic prefetching scheme in this environment is illustrated in **Figure 6**.

While working on a column of a block, we prefetch the first doubleword of the next column, as opposed to the last doubleword. The reason for doing this is similar to that for the natural-order case. Observe that the doublewords of a cache line are now brought from the main memory in the desired order when access of the doubleword causes a cache miss. Thus, the doublewords that come into the cache first are the ones needed next.

```
FOR  i = nb,1,−1 (nb: number of blocks)
  load q doublewords of y1
  load q doublewords of y2
  FOR  j = 1, n
       execute q+1 FPLs concurrently with q+1 FMAs
       execute an FPL for the first doubleword of (j+1)th column (prefetch)
       execute q−1 FMAs
  END FOR
  store q doublewords of y1
  store q doublewords of y2
END FOR
```

Figure 7

Cache-blocking DGEMV2 with prefetch.

Dense-matrix algorithms

In this section, we look at two of the dense-matrix algorithms in which the algorithmic prefetching concept developed in the previous section can be used for improving performance on the RS/6000 workstation. The first algorithm, DGEMV2, is for the multiplication of a dense real matrix by two real vectors. The DGEMV2 algorithm is used in a number of applications, such as linear programming and general matrix factorization. The second algorithm, ZGEMV, is for the multiplication of a dense complex matrix by a vector. Algorithmic prefetching for both of these algorithms can be used for matrix **A** as well as $\mathbf{A}^T$; however, we describe the algorithm only for the normal case—for matrix **A**.

• *DGEMV2: Multiplication of a matrix by two vectors*

Consider the following problem:

$$\mathbf{y}_1 = \mathbf{y}_1 + \mathbf{A}\mathbf{x}_1 ,$$

$$\mathbf{y}_2 = \mathbf{y}_2 + \mathbf{A}\mathbf{x}_2 ,$$

where **A** is an $M \times N$ matrix of real numbers, $\mathbf{x}_1$ and $\mathbf{x}_2$ are two real vectors of size N, and $\mathbf{y}_1$ and $\mathbf{y}_2$ are resultant vectors of size M. We now describe a cache-blocking algorithm with algorithmic prefetching to compute $\mathbf{y}_1$ and $\mathbf{y}_2$.

Partition matrix **A** into horizontal blocks of $q \times N$ elements. For simplicity of presentation, we again assume that three cache lines for each of the N columns can fit in the cache, so no vertical blocking is required. The algorithm processes a block at a time, and within a block it processes a column at a time. The doublewords of $\mathbf{y}_1$ and $\mathbf{y}_2$ are loaded into registers in the outer loop. In the inner loop, the required doublewords of $\mathbf{x}_1$ and $\mathbf{x}_2$ and a column of the block of **A** being processed are loaded into registers. In the inner loop, the FMA is used to do a DAXPY operation; $2q$ FMAs and $q + 2$ FPLs are needed.

Figure 7 gives a high-level description of the algorithm. As our idea is to highlight the algorithmic prefetching concept, we go into only enough detail to describe it. Also, to keep the description clear, exceptional cases, such as the first and last column of a block, are not treated separately. In our description, the destination for a load is a register, and a store refers to moving data from a register to a memory location.

In the inner loop, we first have $q + 1$ FPLs: two are for the jth doubleword of $\mathbf{x}_1$ and $\mathbf{x}_2$, and the remaining $q - 1$ are for the jth column of block i of the matrix **A**. These FPLs are executed concurrently with $q + 1$ FMAs, as in Figure 1. The next single FPL is the prefetch load. After this load, the remaining $q - 1$ FMAs are executed, during which most of the doublewords in the cache line of the prefetched load arrive in cache. Recall that for prefetch to be fully effective, we should have

$$2q - (q + 2) > 8, \text{ or } q > 10.$$

An upper constraint on the value of q is due to the number of available registers. A value of $q = 11$ was determined by numerical experimentation as a good value for the RS/6000, which has 32 floating-point registers. Another constraint is $q \leq 16$, in order that the column length be no greater than the cache line size; otherwise, additional cache misses can occur in the middle of a column.

```
FOR  i = nb,1,−1 (nb: number of blocks)
   execute 2q loads for q elements (2q doublewords) of y
   FOR j = 1, n
        execute 2q+1 FPLs concurrently with  2q+1 FMAs
        execute an FPL for the first real doubleword of (j+1)th column (prefetch)
        execute 2q−1 FMAs
   END FOR
   execute 2q stores for q elements of y
END FOR
```

Figure 8

Cache-blocking ZGEMV with prefetch.

• *ZGEMV: Multiplication of a complex matrix by a complex vector*

Consider the following problem:

$\mathbf{y} = \mathbf{y} + \mathbf{Ax}$,

where **A** is a complex $M \times N$ matrix, and **x** and **y** are complex vectors of size N and M, respectively. In FORTRAN, complex numbers are always stored as a contiguous pair of real numbers. We now give an informal description of the algorithm with algorithmic prefetching.

Partition matrix **A** into horizontal blocks of $q \times N$ each. For simplicity of presentation, we again assume that three cache lines for each of the N columns can fit in the cache of the machine, so no vertical blocking is required. The algorithm processes a block at a time, and within a block it processes a column at a time. The algorithm consists of two nested loops. In the outer loop, pairs of doublewords of **y** are loaded into a set of registers. In the inner loop, we load the required pair of doublewords of **x**, and a column of the block being processed is loaded into registers. The computation in the inner loop is of the form $y(i) = y(i) + a(i, j) \times x(j)$. A pair of doublewords of **y**, consisting of a real and an imaginary part, can be computed using three FMA instructions and a floating-point multiply–subtract (FMS) instruction. For accounting purposes, we make no distinction between an FMA and an FMS. Thus, to compute q pairs of doublewords of **y**, we need $4q$ FMAs. The number of FPLs required in the inner loop is $2(q + 1)$. Of these, $2q$ loads are for loading q pairs of doublewords of a column of **A**, and 2 for loading a pair of doublewords of **x**. Note that a complex pair of doublewords occupies two floating-point registers.

Figure 8 gives a high-level description of the algorithm. Here also, to keep the description clear, exceptional cases are not treated.

In the inner loop, we first have $2(q + 1)$ FPLs, which are executed concurrently with $2(q + 1)$ FMAs, as outlined in the above subsection on algorithmic prefetching. The last single load is the prefetch step. For this problem, we selected $q = 7$, which ensures that

$2q - 1 > 8$.

Note that there are more FMAs available to hide the cache penalty in this problem $(2q - 1)$ than in DGEMV2 $(q - 1)$. For $q = 7$, the number of FMAs available to hide the cache latency (13) is sufficiently greater than 8 that the reverse processing of blocks (see the section on Level-2-like computation, above) is not necessary.

Experimental results

We have implemented two versions each of DGEMV2 and ZGEMV on the RS/6000 Model 530, one with algorithmic prefetching and the other without. All coding was done using standard FORTRAN 77. The performance of the two versions was compared for both of the problems. To be sure that none of the data were in cache, we flushed cache before executing the algorithm. Our results are summarized in **Figures 9** and **10**. The performance is defined as the number of floating-point operations necessary to compute the answers ($4mn$ for DGEMV2 and $8mn$ for ZGEMV) divided by the total execution time in microseconds. Figures 9 and 10 show this performance as a function of the matrix/array dimension M, with N constant ($N = 140$ for DGEMV2 and 120 for ZGEMV).

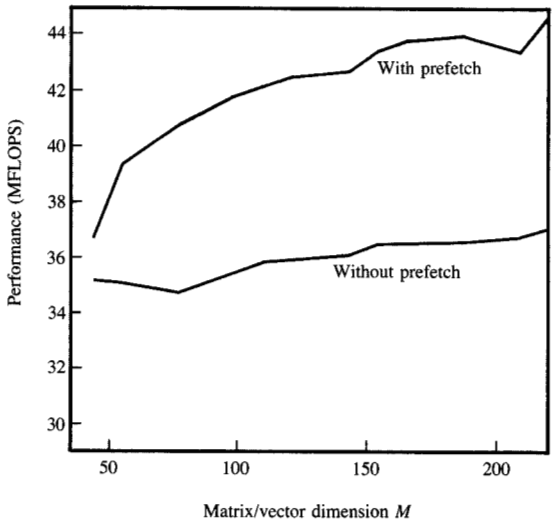

Figure 9

Performance of DGEMV2 with and without algorithmic prefetching ($N = 140$).

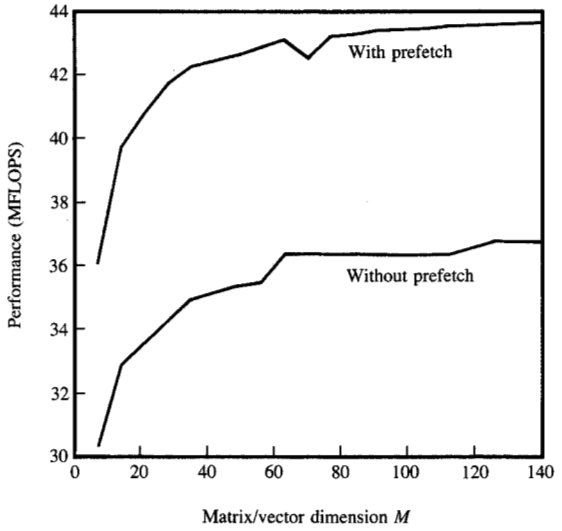

Figure 10

Performance of ZGEMV with and without algorithmic prefetching ($N = 120$).

The Model 530 has a peak performance of 50 MFLOPS. We chose $LDA = M$ for most of our data points, since this allowed us to use a larger value of N. At one point we observed a bad LDA, resulting in about 10% performance degradation. We restored the 10% performance loss for this LDA value by setting $LDA = M + 1$ for that point.

In Figure 9, we plot the performance of DGEMV2 with and without algorithmic prefetching. The performance without algorithmic prefetching saturates at 74% (37 MFLOPS) of the peak performance (despite the appearance of rising near $M = 225$), while the performance with algorithmic prefetching goes up to 89% (44.5 MFLOPS) of the peak performance. Similar behavior can be observed for ZGEMV (see Figure 10). Both Figures 9 and 10 are plotted for a discrete set of points. In two cases there are drops (dips) in the curves corresponding to bad LDA points.

Conclusion

In this paper, we have introduced a new concept, called algorithmic prefetching, which can be exploited to improve NIC performance. In particular, we have demonstrated that algorithmic prefetching can improve the performance of DGEMV2 and ZGEMV. The algorithmic prefetching concept is general, however, and can be applied to other dense-matrix operations on architectures similar to RS/6000.

Appendix: Glossary of acronyms

BLAS	basic linear algebra subroutines
DAXPY	double-precision $\mathbf{a} \cdot \mathbf{x} + \mathbf{y}$ (Level-1 BLAS)
DGEMM	double-precision general matrix multiply (Level-3 BLAS)
DGEMV	double-precision general matrix vector (Level-2 BLAS)
DGEMV2	double-precision general matrix vector 2 (Level-2 BLAS)
ESSL	Engineering and Scientific Subroutine Library
FMA	floating-point multiply–add instruction
FMS	floating-point multiply–subtract instruction
FPA	floating-point arithmetic instruction
FPL	floating-point load instruction
GEMV	general matrix vector (generic Level-2 BLAS)
LDA	leading dimension of array
NIC	numerically intensive computations
TLB	translation lookaside buffer
ZGEMV	double-precision complex general matrix vector (Level-2 BLAS)

Acknowledgments

The authors wish to thank Aparna Patil, who performed the initial implementation of this cache-prefetching scheme in computing symmetric, packed-format, complex BLAS-2. We also thank Dr. Anne Rogers for making us aware of the extensive literature on software prefetching.

References

1. J. J. Dongarra, F. G. Gustavson, and A. Karp, "Implementing Linear Algebra Algorithms for Dense Matrices on a Vector Pipeline Machine," *SIAM Rev.* **26**, 91–112 (1984).
2. K. Gallivan, W. Jalby, U. Meier, and A. Sameh, "Impact of Hierarchical Memory Systems on Linear Algebra Algorithms Design," *Int. J. Supercomputer Appl.*, pp. 12–48 (1988).
3. J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling, "Algorithms 679: A Set of Level 3 Basic Linear Algebra Subprograms," *ACM Trans. Math. Soft.* **16**, 18–28 (1990).
4. *IBM Engineering and Scientific Subroutine Library, Guide and Reference*, Order No. SC23-0526-00, 1992; available through IBM branch offices.
5. R. C. Agarwal and F. G. Gustavson, "A Parallel Implementation of Matrix Multiplication and LU Factorization on the IBM 3090," *Proceedings of the IFIP WG 2.5 Working Conference on Aspects of Computation on Asynchronous Parallel Processors*, 1989, pp. 217–221.
6. D. Callahan, K. Kennedy, and A. Porterfield, "Software Prefetching," *Proceedings of ASPLOS-IV*, 1991, pp. 40–52.
7. A. Rogers and K. Li, "Software Support for Speculative Loads," *Proceedings of ASPLOS-V*, 1992, pp. 38–50.
8. T. Chen and J. Baer, "Reducing Memory Latency Via Non-Blocking and Prefetching Caches," *Proceedings of ASPLOS-V*, 1992, pp. 51–61.
9. T. C. Mowry, M. S. Lam, and A. Gupta, "Design and Evaluation of a Compiler Algorithm for Prefetching," *Proceedings of ASPLOS-V*, 1992, pp. 62–75.
10. *IBM AIX XL FORTRAN Compiler/6000 Version 2.1*, Order No. SC09-1353-00, 1991; available through IBM branch offices.
11. *IBM RISC System/6000 Technology*, Order No. SA23-2619, 1990; available through IBM branch offices.
12. J. Dongarra, P. Mayes, and G. Radicati di Brozolo, "The IBM RISC System 6000 and Linear Algebra Operations," *Technical Report CS-90-122*, Computer Science Department, University of Tennessee, Knoxville, December 1990.

Received August 2, 1993; accepted for publication February 4, 1994

Ramesh C. Agarwal *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (AGARWAL at YKTVMV, agarwal@watson.ibm.com).* Dr. Agarwal received a B.Tech. (Hons.) degree from the Indian Institute of Technology (IIT), Bombay. He was the recipient of The President of India Gold Medal while there. He received M.S. and Ph.D. degrees from Rice University and was awarded the Sigma Xi Award for best Ph.D. thesis in electrical engineering. He has been a member of the Mathematical Sciences Department at the IBM Thomas J. Watson Research Center since 1983. Dr. Agarwal has done research in many areas of engineering, science, and mathematics and has published over 60 papers in various journals. Currently, his primary research interest is in the area of algorithms and architecture for high-performance computing on workstations and scalable parallel machines. In 1974, Dr. Agarwal received the Senior Award from the IEEE Acoustics, Speech, and Signal Processing (ASSP) group, for best papers. He has received several Outstanding Achievement Awards and a Corporate Award from IBM. Dr. Agarwal is a Fellow of the IEEE and a member of the IBM Academy of Technology.

Fred G. Gustavson *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (GUSTAV at YKTVMV, gustav@watson.ibm.com).* Dr. Gustavson is manager of Algorithms and Architectures in the Mathematical Sciences Department at the IBM Thomas J. Watson Research Center. He received his B.S. in physics, and his M.S. and Ph.D. degrees in applied mathematics, all from Rensselaer Polytechnic Institute. He joined IBM Research in 1963. One of his primary interests has been in developing theory and programming techniques for exploiting the sparseness inherent in large systems of linear equations. Dr. Gustavson has worked in the areas of nonlinear differential equations, linear algebra, symbolic computation, computer-aided design of networks, design and analysis of algorithms, and programming applications. He and his group are currently engaged in activities that are aimed at exploiting the novel features of the IBM family of RISC processors. These include hardware design for divide and square root, new algorithms for POWER2™ for the Engineering and Scientific Subroutine Library (ESSL) and for other math kernels, and parallel algorithms for distributed memory processors. Dr. Gustavson has received an IBM Outstanding Contribution Award, an IBM Outstanding Innovation Award, an IBM Outstanding Invention Award, two IBM Outstanding Technical Achievement Awards, two IBM Corporate Technical Recognition Awards, and a Research Division Technical Group Award.

Mohammad Zubair *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (ZUBAIR at YKTVMV, zubair@watson.ibm.com).* Dr. Zubair received his Ph.D. degree in 1987 from the Indian Institute of Technology (IIT), New Delhi. From 1981 to 1987, he was at the Center for Applied Research in Electronics, IIT Delhi. In 1987, he became an Assistant Professor at Old Dominion University, Norfolk, VA, and in 1993 he became an Associate Professor. He joined IBM Research in 1994. Dr. Zubair's primary research interest is in the algorithm and architecture aspects of large-scale scientific computing. He has published over 30 papers in various journals and conference proceedings.